Technische Universität München

TUM School of Computation, Information and Technology

Geometric Machine Learning-

Structure-Preserving Neural Networks for Scientific Computing

Benedikt Brantner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

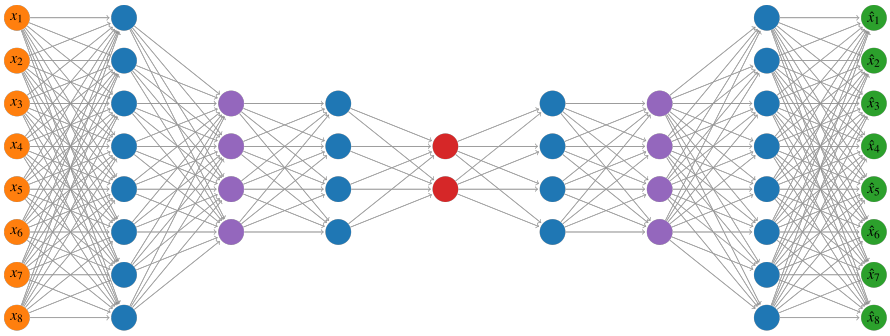
Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitz:	Prof. Dr. Oliver Junge
Prüfende der Dissertation:	1. Prof. Dr. Eric Sonnendrücker
	2. Prof. Dr. Tatjana Stykel
	3. Assistant Prof. Dr. Joshua Burby

Die Dissertation wurde am 25.09.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 11.02.2025 angenommen.

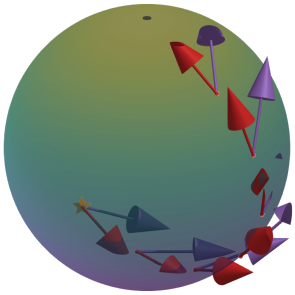
MAX-PLANCK-INSTITUT
FÜR PLASMAPHYSIK



Geometric Machine Learning

Benedikt Brantner

September 24, 2024



Preliminary Remarks and Acknowledgements

During the time I conducted my PhD the TUM School of Computation, Information, and Technology (CIT) replaced the former departments of Mathematics, Informatics, and Electrical and Computer Engineering. Its mission statement reads: "[CIT unites] a wide spectrum of different competencies:

- from theoretical foundations to implementation in different contexts of application,
- from elementary components via hardware and software architectures to holistic technical systems,
- from mathematical abstraction to technological engineering processes,
- from formal specification to partially automated implementation."

Although not touching on all these aspects, I tried to make my dissertation fit into this framework: it is closely linked to the development of a software package that can be used "in different contexts of application" and contains algorithms that emerged as a result of "mathematical abstraction". In the spirit of the CIT this text contains both code snippets demonstrating the software package and mathematical theorems (including proofs).

At this point I want to acknowledge several people who helped me professionally and personally. My foremost thanks go to my scientific mentor Michael Kraus who throughout my PhD provided me with guidance, nudged me in the right direction when I needed it, and was extremely understanding whenever progress was slow. He almost always gave me the maximum amount of freedom to pursue things I found interesting. The only instance I can think of, where this freedom was not granted, was his insistence that all his students use Julia, a demand that sped up getting results immensely.

I am also very grateful to Eric Sonnendrücker, the head of our department, who gave me many opportunities to present my work at international conferences which in turn enabled me to meet new people (very often the faces behind the papers I read) and exchange ideas. My thanks also go to all my colleagues, especially those I regularly had lunch with, for stimulating discussions on work and everything else in the world.

On a more personal note I want to thank my parents for their patience, understanding and relentless encouragement; I often think back to them telling me: "Irgendwie geht's immer."

Lastly I want to thank Lejie for her love and support, remarks that very often helped me organize thoughts in my head, and her extreme understanding when I opened my laptop for the umpteenth time for a late-night programming session. 路漫漫其修远兮，我们将上下而求索。我永远爱你。

Summary

Scientific computing has become an indispensible tool for many disciplines like biology, engineering and physics. It is, for example, used to (i) establishing connections between theories and experiments, (ii) making predictions and (iii) aiding in optimizing parameters in the design of large-scale projects such as fusion reactors.

In practice scientific computing involves solving partial differential equations, usually on supercomputers; and this can be very expensive. Scientists have long tried to reduce the cost required to solve these equations by various methods. In this work we focus on data-driven reduced order modeling to do so. This approach in practice often means employing machine learning techniques such as neural networks to process data that come from simulations.

Neural networks have facilitated breakthroughs in various fields, but their application in reduced order modeling is still in its infancy. A largely neglected aspect is the adherence to properties in their algorithms that have been observed to be important in traditional scientific computing. These properties often pertain to the structure of the solution of the differential equations.

From theoretical and empirical results we know such structure is often indispensible when performing simulations and should be accounted for, and structure preservation serves as the main motivation of this dissertation. To this end we design new neural network architectures that preserve structure. *Geometric* has traditionally been used as a synonym for structure-preserving and we therefore adopt the name *geometric machine learning* as an umbrella term for the ideas introduced in this work.

Geometric machine learning and neural networks are not an entirely novel concept and other researches have proposed architectures that preserve structure before. There are however many new instances of geometric neural networks presented here that are original work. This dissertation is structured into four main parts.

In Part I we give background information that does not constitute novel work, but lays the foundation for the coming chapters. This first part includes a basic introduction into the theory of Riemannian manifolds, a basic discussion of structure preservation and a short explanation of data-driven reduced order modeling.

Part II discusses a novel optimization framework that generalizes existing neural network optimizers, such as the Adam optimizer and the BFGS optimizer, to manifolds. These new optimizers were necessary to enable the training of a new neural network architecture which we call *symplectic autoencoders* (SAEs).

Part III finally introduces various special neural network layers and architectures. Some of them, like *SympNets* and the *multihead attention layer*, are well-known, but others like SAEs, *volume-preserving attention* and the *linear symplectic transformer* are new.

In Part IV we give some results based on the new architectures and show how they compare to existing approaches. Most of these applications pertain to applications from physics; but to demonstrate the efficacy of the new optimizers we resort to a classical problem from image classification to show that geometric machine learning can also find applications in fields outside of scientific computing. For all examples that we show, our new architectures exhibit a clear improvement in terms of speed or accuracy over existing architectures. We show one example where we obtain a speed-up of a factor of 1000 with SAEs and transformers.

Zusammenfassung

Wissenschaftliches Rechnen ist ein unverzichtbares Werkzeug für viele Disziplinen wie Biologie, Ingenieurwesen und Physik. Es ist wichtig z.B. für (i) die Verknüpfung zwischen Theorien und empirischen Beobachtungen, (ii) das Treffen von Vorhersagen ohne physische Experimente und (iii) das Design von Apparaturen wie Fusionsreaktoren. In der Praxis bedeutet wissenschaftliches Rechnen fast immer, dass partielle Differentialgleichungen gelöst werden müssen, meist auf Supercomputern; und das ist gewöhnlich sehr teuer.

Wissenschaftler versuchen seit langem die Kosten für das Lösen dieser Gleichungen zu senken, indem sie z. B. Modelle vereinfachen oder reduzierte Darstellungen konstruieren. Einer dieser Ansätze ist *Data-Driven Reduced Order Modeling* (DDROM), das in den letzten Jahren aufgrund seiner Eignung für moderne Hardware und Algorithmen an Bedeutung gewonnen hat. In der Praxis bedeutet das oft die Anwendung von Techniken des maschinellen Lernens wie neuronale Netze.

Neuronale Netze haben in verschiedenen Anwendungen ein enormes Potenzial gezeigt. Für DDROM sind die entsprechenden Ergebnisse oft allerdings noch nicht sehr zufriedenstellend. Wissenschaftler haben bei ihrer Anwendung von neuronalen Netzen oft Eigenschaften vernachlässigt, die sich im traditionellen wissenschaftlichen Rechnen als sehr wichtig erwiesen haben. In dieser Arbeit beziehen sich diese Eigenschaften auf die Struktur der Differentialgleichungen; diese ist bei der Durchführung von Simulationen oft unverzichtbar um Stabilität zu gewährleisten.

In dieser Arbeit bezeichnen wir Methoden des maschinellen Lernens, die auf die spezifische Struktur einer Differentialgleichung zugeschnitten sind, als *geometrisches maschinelles Lernen*. Der Begriff *geometrisch* wird in diesem Zusammenhang traditionell auch verwendet und ist als Synonym des Wortes *strukturerhaltend* zu verstehen. Die Idee neuronale Netze *geometrisch* zu machen ist nicht neu; viele der Netzwerke die wir hier präsentieren sind es aber und bilden einen wichtigen Teil dieser Dissertation. Diese Arbeit teilt sich in vier Teile.

In Teil I geben wir Hintergrundinformationen wieder, die keine neue Arbeit darstellen, aber die Grundlage für die folgenden Kapitel bilden. Dieser erste Teil enthält eine grundlegende Einführung in die Theorie der Riemannschen Mannigfaltigkeiten, eine Diskussion über Strukturerhaltung und eine kurze Erläuterung von DDROM.

In Teil II wird ein neues Optimierungsframework eingeführt, das bestehende Optimierer für neuronale Netze auf Mannigfaltigkeiten verallgemeinert. Beispiele hierfür sind der Adam-Optimierer und der BFGS-Optimierer. Diese neuen Optimierer waren notwendig, um das Training einer neuen neuronalen Netzwerkarchitektur zu ermöglichen, die wir *symplektische Autoenkoder* (SAE) nennen.

In Teil III werden schließlich verschiedene spezielle neuronale Netzwerkarchitekturen vorgestellt. Einige von ihnen, wie *SympNets* und *Multi-Head Attention*, stellen keine Neuheiten dar, aber andere, wie SAEs, *Volume-Preserving Attention* und der *lineare symplektische Transformer*, sind originell.

In Teil IV geben wir einige Ergebnisse an, die auf den neuen Architekturen basieren. Die meisten dieser Anwendungen beziehen sich auf Anwendungen aus der Physik; um jedoch die neuen Optimierer zu demonstrieren, greifen wir auf ein klassisches Problem aus der Bildklassifikation zurück. Wir wollen damit zeigen dass geometrisches maschinelles Lernen auch in Bereichen außerhalb des wissenschaftlichen Rechnens Anwendung finden kann. In allen behandelten Problemen ist zu sehen dass unsere Modelle genauer oder schneller als vergleichbare Architekturen sind. In einem Beispiel

zeigen wir wie ein SAE-reduziertes Model die Auswertung eines Problems um einen Faktor 1000 beschleunigt.

Contents

Chapter 1

Introduction and Outline

One may argue that *structure-preserving machine learning* is a contradiction. One of the most popular books on neural networks [1] introduces the term *machine learning* the following way: "The difficulties faced by systems relying on hard-coded knowledge suggest that [artificial intelligence] systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as machine learning." The many success stories of deep neural networks such as ChatGPT [2] have shown that abandoning hard-coded knowledge in favour of extracting patterns can yield enormous improvement for many applications. In scientific computing the story is a different one however. Hard coding of certain properties into an algorithm has proved indispensible for many numerical applications. The introductory chapter to one of the canonical references on geometric numerical integration [3] contains the sentence: "It turned out that the preservation of geometric properties of the flow not only produces an improved qualitative behaviour, but also allows for a more accurate long-time integration than with general-purpose methods." Here "preservation of geometric properties" means *hard-coding physical information* into an algorithm.

Despite the allure of neglecting hard-coded knowledge in an "era of big data" [4] many researchers have very early realized that systems that work for image recognition, natural language processing and other purely data-driven tasks may not be suitable to treat problems from physics [5]. Scientific machine learning [6], which in this work refers to the application of machine learning techniques for the solution of differential equations from science and engineering, has however much too often neglected the preservation of geometric properties that has proved to be so important in traditional numerics. An ostensible solution is offered by so-called physics-informed neural networks (PINNS), whose eponymous paper [7] is one of the most-cited in scientific machine learning. The authors write: "Coming to our rescue, for many cases pertaining to the modeling of physical and biological systems, there exists a vast amount of prior knowledge that is currently not being utilized in modern machine learning practice." It is stated that PINNS "[enrich] deep learning with the longstanding developments in mathematical physics;" one should however add that they also ignore *longstanding developments in numerics*, like preserving the geometric properties which are observed to be crucial in [3].

What this work aims at doing is not "to set the foundations for a new paradigm" [7], but rather to show that in many cases it is advantageous to imbue neural networks with specific structure and one should to do this whenever possible. In this regard this work is much more closely related to traditional numerics than to neural network research as we try to design problem-specific algorithms rather than "universal approximators" [8]. The *structure-preserving neural networks* in this work are never fundamentally new architectures but build on existing neural network designs [9, 10] or more classical methods [11]. We design neural networks that have a specific structure encoded in them (modeling part) and then make their behavior reflect information found in data (machine learning part). We refer to this as *geometric machine learning*.

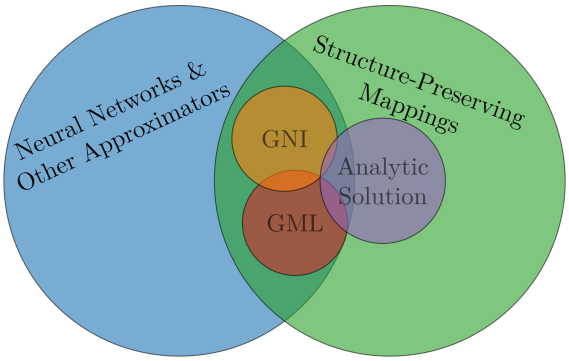


Figure 1.1: Geometric machine learning (GML) like traditional geometric numerical integration (GNI) and other structure-preserving numerical methods aims at building models that share properties with the analytic solution of a differential equation.

In the picture above we visualize that geometric machine learning aims at constructing so-called structure-preserving mappings that are ideally close to the analytic solution and perform better than classical methods (e.g. GNI). *Structure-preserving* here means that the model shares properties with the analytic solution of the underlying differential equation. In this work the most important of these properties are *symplecticity* and *volume preservation*, but this may extend to others such as the null space of certain operators [12] and symmetries encoded into a differential equation [13, 14].

For us the biggest motivation for geometric machine learning comes from *data-driven reduced order modeling*. There we want to find *reduced representations* of so-called *full order models* of which we have data available; such reduced representation ideally have much lower computational complexity then the full order model. Data-driven reduced order modeling is especially useful when solving parametric partial differential equations (PPDEs). In this case we can solve the full order model for a few parameter instances and then build a cheaper representation of the full model (a so-called *reduced model*) with neural networks. This can bring dramatic speed-ups in performance.

Closely linked to the research presented here is the development of a software package written in `Julia` called `GeometricMachineLearning` [15]. Throughout this work we will demonstrate concepts such as neural network architecture and (Riemannian) optimization by using `GeometricMachineLearning`[1]. Most sections contain a subsection **Library Functions** that explains types and functions in `GeometricMachineLearning` that pertain to the text in that section (they are generated as so-called docstrings [16]). We show an example here:

`GeometricMachineLearning.GradientCache` – Type.

```
GradientCache(Y)
```

Do not store anything.

The cache for the `GradientOptimizer` does not consider past information.

source

So the docstring shows the name of the type or method, in most cases how to call it and then gives some information explaining what it does and potentially hyperlinks to other similar docstrings

[1] This document was produced with `GeometricMachineLearning v0.3`. It may be that the interface will slightly change in future versions, but efforts will be made to keep these changes as small as possible.

(`GradientOptimizer` in this case); all of this information is indented by a tab. Docstrings may include other information under subheaders **Arguments** (showing the arguments the method can be supplied with), **Examples** (giving more detailed examples (including results) of how to use the method) and **Implementation** (giving details on how the method is implemented). When we reference a docstring it is always printed in blue (e.g. `GradientOptimizer`), indicating a hyperlink. In addition there is an *index of docstrings* showing all docstrings in chronological order with the associated page number.

Similar to **Library Functions**, which is included in most sections, almost every chapter concludes with a section **Chapter Summary** and an additional section **References** that shows further related reading material. The **Chapter Summary** recaps the important aspects of the corresponding chapter, states again what is new (this may be mathematical or software aspects) and gives information to what other parts of the dissertation the contents of the present chapter are relevant.

All the code necessary to reproduce the results is included in the text and does not have any specific hardware requirements. Except for training some of the neural networks (which was done on an NVIDA Geforce RTX 4090 [17]) all the code snippets were run on CPU (via GitHub runners [18]). All plots have been generated with `Makie` [19].

This dissertation is structures into four main parts: (i) background information, (ii) an explanation of the optimizer framework used for training neural networks, (iii) a detailed explanation of the various neural network layers and architectures that we use and (iv) a number of examples of how to use `GeometricMachineLearning`. We explain the content of these parts in some detail[2].

1.1 Background Information

The background material, which does not include any original work, covers all the prerequisites for introducing our new optimizers in Part II. In addition it introduces some basic functionality of `GeometricMachineLearning`. It contains (among others) the following sections:

- Concepts from general topology (see Chapter 2.1): here we introduce topological spaces, closedness, compactness, countability and Hausdorffness amongst others. These concepts are prerequisites for defining manifolds.
- General theory on manifolds (see Chapter 2.5): we introduce manifolds, the preimage theorem and submersion theorem. These theorems will be used to construct manifolds; the preimage theorem is used to give structure to the Stiefel (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10), and the immersion theorem gives structure to the solution manifold (see Chapter 4.1) which is used in reduced order modeling.
- Riemannian manifolds (see Chapter 2.7): for optimizing on manifolds we need to define a metric on them, which leads to *Riemannian manifolds*. We introduce *geodesics* and the *Riemannian gradient* here.
- Homogeneous spaces (see Chapter 2.8): homogeneous spaces are a special class of manifolds to which our *generalized optimizer framework* can be applied. They trivially include all Lie groups and spaces like the Stiefel manifold (see Chapter 2.9), the Grassmann manifold (see Chapter 2.10) and the "homogeneous space of positions and orientations" [20].
- Global tangent spaces (see Chapter 2.11): homogeneous spaces allow for identifying for an invariant representation of all tangent spaces which we call *global tangent spaces*[3]. We explain this concept in this section.

[2] In addition there is also an appendix that provides more implementation details.

- Geometric structure (see Chapter 3.1): structure preservation takes a prominent role in this dissertation. In general *structure* refers to some property that the analytic solution of a differential equation also has and that we want to preserve when modeling the system. Here we discuss *symplecticity* and volume preservation (see Chapter 3.2) in detail. We also introduce neural networks in this chapter and give a definition of geometric neural networks (see Chapter 3.3).
- Reduced order modeling (see Chapter 4.1): reduced order modeling serves as a motivation for most of the architectures introduced here. In this section we introduce the basic idea behind reduced order modeling, show a typical workflow and explain what structure preservation looks like in this context (see Chapter 4.5).

1.2 The Optimizer Framework

One of the central parts of this dissertation is an *optimizer framework* that allows the generalization of existing optimizers such as Adam [23] and BFGS [24, Chapter 6.1] to homogeneous spaces in a consistent way[4]. This part contains the following sections:

- Neural Network Optimizers (see Chapter 5.1): here we introduce the concept of a neural network optimizer and discuss the modifications we have to make in order to generalize them to homogeneous spaces.
- Retractions (see Chapter 5.2): an important concept in manifold optimization are retractions [25]. We introduce them in this section, discuss how they can be constructed for homogeneous spaces and show the two examples of the *geodesic retraction* and the *Cayley retraction*.
- Parallel Transport (see Chapter 5.3): whenever we have an optimizer that contains momentum terms (such as Adam for example) we need to *transport* these momenta. In this section we explain how this can be done straightforwardly when dealing with homogeneous spaces.
- Optimizer methods (see Chapter 6.1): in this section we introduce simple optimizers such as the *gradient optimizer*, the *momentum optimizer* and *Adam* and show how to generalize them to our setting. Due to its increased complexity the BFGS optimizer gets its own section (see Chapter 6.2).

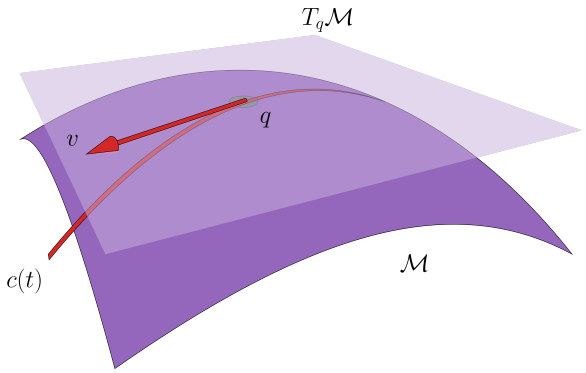


Figure 1.2: Weights can be put on manifolds to achieve structure preservation or improved stability.

[3] These spaces are also discussed in [21, 22].

[4] The optimizer framework was introduced in [26].

1.3 Special Neural Network Layers and Architectures

In here we first discuss specific neural network layers and then architectures. A neural network architecture is always a composition of many neural network layers that is designed for a specific task.

Special neural network layers include:

- SympNet layers (see Chapter 7.1): *symplectic neural networks* (SympNets) [10] are special neural networks that are *universal approximators in the class of canonical symplectic maps.* SympNet layers comprise three different types: linear layers, activation layers and gradient layers. All these are introduced here.
- Volume-preserving layers (see Chapter 7.2): the volume-preserving layers presented here are inspired by linear and activation SympNet layers. They slightly differ from other approaches with the same aim [27].
- Attention layers (see Chapter 7.3): many fields in neural network research have seen big improvements due to *attention mechanisms* [9, 28, 29]. Here we introduce this mechanism (which is a neural network layer) and also discuss how to make it volume-preserving [30]. It serves as a basis for multihead attention (see Chapter 7.4) and linear symplectic attention (see Chapter 7.5).

Special neural network architectures include:

- Symplectic autoencoders (see Chapter 8.2): the *symplectic autoencoder* constitutes one of the central elements of this dissertation. It offers a way of flexibly performing nonlinear model order reduction for Hamiltonian systems. In this section we explain its architecture and how it is implemented in `GeometricMachineLearning` in detail.
- SympNet architectures (see Chapter 8.5): based on SympNet layers (which are simple building blocks) one can construct two main types of architectures which are called LA-SympNets and G-SympNets. We explain both here.
- Volume-preserving feedforward neural networks (see Chapter 8.6): based on LA-SympNets we build *volume-preserving feedforward neural networks* that can learn arbitrary volume-preserving maps.
- Transformers (see Chapter 8.7): transformer neural networks have revolutionized many fields in machine learning like natural language processing [9] and image recognition [29]. We discuss them here and further imbue them with structure-preserving properties to arrive at volume-preserving transformers (see Chapter 8.8) and linear symplectic transformers (see Chapter 8.9).

We note that SympNets, volume-preserving feedforward neural networks and the three transformer types presented here belong to a category of neural network integrators (see Chapter 8.3), which are used in the *online stage* of reduced order modeling. That makes them different from symplectic autoencoders which are used in the *offline stage*.

1.4 Examples

In this part we demonstrate the neural network architectures implemented in `GeometricMachineLearning` with a few examples:

- Symplectic autoencoders (see Chapter 9.1): here we show how to reduce the Toda lattice [31], which is a 400-dimensional Hamiltonian system in our case, to a two-dimensional Hamiltonian system with symplectic autoencoders.
- SympNets (see Chapter 10.1): this serves as an introductory example into using `GeometricMachineLearning` and does not contain any new results. It simply shows how to use SympNets to learn the flow of a harmonic oscillator.
- Image classification (see Chapter 11.1): Here we perform image classification for the MNIST dataset [32] with vision transformers and show that manifold optimization can enable convergence that would otherwise not be possible.
- The Grassmann manifold in neural networks (see Chapter 12.1): in this example we model a surface embedded in $\mathbb{R}^3$ with the help of the Grassmann manifold.
- Different volume-preserving attention mechanisms (see Chapter 11.3): the volume-preserving attention mechanism (see Chapter 7.3) in `GeometricMachineLearning` is based on computing correlations in the input sequence. These correlations can be constructed in two different ways. Here we compare these two.
- Linear Symplectic Transformer (see Chapter 10.2): the linear symplectic transformer is used to integrate the four-dimensional Hamiltonian system of the *coupled harmonic oscillator*. Here we compare the linear symplectic transformer to the standard transformer and SympNets.

1.5 Associated Papers and Contributions

The following papers have emerged in connection with the development of `GeometricMachineLearning`:

1. In [26] a new class of optimizers for *homogeneous spaces*, a category that includes the Stiefel manifold and the Grassmann manifold, is introduced. The results presented in this paper are reproduced in the examples (see Chapter 11.1).
2. In [33] we introduced a new neural network architectures that we call *symplectic autoencoders*. This is capable of performing non-linear Hamiltonian model reduction. During training of these symplectic autoencoders we use the optimizers introduced in [26]. Similar results to what is presented in the paper are reproduced as an example (see Chapter 9.1).
3. In [30] we introduce a new neural network architecture that we call *volume-preserving transformers*. This is a structure-preserving version of the *standard transformer* [9] for which all components have been made volume preserving. As application we foresee the *online phase* in reduced order modeling.

In addition there are new results presented in this work that have not been written up as a separate paper:

1. Similar to the volume-preserving transformer [30] we introduce a linear symplectic transformer (see Chapter 8.9) that preserves a symplectic product structure and is also foreseen to be used in reduced order modeling.
2. We show how the Grassmann manifold can be included into a neural network (see Chapter 12.1) and construct a loss based on the Wasserstein distance to approximate a nonlinear space from which we can then sample.

Part I

Background

Chapter 2

Manifolds

In this chapter we introduce basic concepts necessary to discuss manifolds and manifold optimization. We begin by discussing *topological vector spaces* and *topological metric spaces*, and several theorems important for developing a theory of manifolds such as the *implicit function theorem*. We then define manifolds and discuss the *preimage theorem* and the *immersion theorem* as tools to give general spaces the structure of a manifold. We then proceed with a discussion on *geodesics* and *Riemannian manifolds*. The chapter concludes with a presentation of *homogeneous spaces* and their *global tangent space representation* that will be crucial for generalizing neural network optimizers to the manifold setting.

2.1 Basic Concepts from General Topology

Here we discuss basic notions of topology that are necessary to define manifolds (see Chapter 2.5) and work with them. Here we largely omit concrete examples and only define concepts that are necessary for defining a manifold[1], namely the properties of being *Hausdorff* and *second countable*. For a detailed discussion of the theory and for a wide range of examples that illustrate this theory see e.g. [34]. The here-presented concepts are also (rudimentarily) covered in most differential geometry textbooks such as [35, 36].

We now start by giving all the definitions, theorem and corresponding proofs that are needed to define manifolds. Every manifold is a *topological space* which is why we give this definition first:

Definition 2.1.1. A **topological space** is a set $\mathcal{M}$ for which we are given a collection of subsets of $\mathcal{M}$, which we denote by $\mathcal{T}$ and call the *open subsets*. $\mathcal{T}$ further has to satisfy the following three conditions:

1. The empty set and $\mathcal{M}$ belong to $\mathcal{T}$.
2. Any union of an arbitrary number of elements of $\mathcal{T}$ again belongs to $\mathcal{T}$.
3. Any intersection of a finite number of elements of $\mathcal{T}$ again belongs to $\mathcal{T}$.

So an arbitrary union of open sets is again open and a finite intersection of open sets is again open.

Based on this definition of a topological space we can now define what it means to be *Hausdorff*:

Definition 2.1.2. A topological space $\mathcal{M}$ is said to be **Hausdorff** if for any two points $x, y \in \mathcal{M}$ we can find two open sets $U_x, U_y \in \mathcal{T}$ s.t. $x \in U_x, y \in U_y$ and $U_x \cap U_y = \{\}$.

[1] Some authors (see e.g. [35]) do not require these properties. But since they constitute very weak restrictions and are always satisfied by the manifolds relevant for our purposes we require them here.

We now give the second definition that we need for defining manifolds, that of *second countability*:

Definition 2.1.3. A topological space $\mathcal{M}$ is said to be **second-countable** if we can find a countable subcollection of $\mathcal{T}$ called $\mathcal{U}$ s.t. $\forall U \in \mathcal{T}$ and $x \in U$ we can find an element $V \in \mathcal{U}$ for which $x \in V \subset U$.

We now give a few definitions and results that are needed for the inverse function theorem (see Chapter 2.4) which is essential for practical applications of manifold theory. We start with the definition of *continuity*:

Definition 2.1.4. A mapping f between topological spaces $\mathcal{M}$ and $\mathcal{N}$ is called **continuous** if the preimage of every open set is again an open set, i.e. if $f^{-1}\{U\} \in \mathcal{T}$ for U open in $\mathcal{N}$ and $\mathcal{T}$ the topology on $\mathcal{M}$.

Continuity can also be formulated in terms of *closed sets* instead of doing it with *open sets*. The definition of closed sets is given below:

Definition 2.1.5. A **closed set** of a topological space $\mathcal{M}$ is one whose complement is an open set, i.e. F is closed if $F^c \in \mathcal{T}$, where the superscript ^{c} indicates the complement: $F^c := \{x \in \mathcal{M} : x \notin F\}$. For closed sets we thus have the following three properties:

1. The empty set and $\mathcal{M}$ are closed sets.
2. Any union of a finite number of closed sets is again closed.
3. Any intersection of an arbitrary number of closed sets is again closed.

So a finite union of closed sets is again closed and an arbitrary intersection of closed sets is again closed.

We now give the definition of continuity in terms of closed sets:

Theorem 2.1.6. *The definition of continuity in terms of open sets is equivalent to the following, second definition: $f : \mathcal{M} \to \mathcal{N}$ is continuous if $f^{-1}\{F\} \subset \mathcal{M}$ is a closed set for each closed set $F \subset \mathcal{N}$.*

Proof. First assume that f is continuous according to the first definition and not to the second. Then $f^{-1}\{F\}$ is not closed but $f^{-1}\{F^c\}$ is open. But $f^{-1}\{F^c\} = \{x \in \mathcal{M} : f(x) \notin \mathcal{N}\} = (f^{-1}\{F\})^c$ cannot be open, else $f^{-1}\{F\}$ would be closed. The implication of the first definition under assumption of the second can be shown analogously. □

The next theorem makes the rather abstract definition of *closed sets* more concrete; this definition is especially important for many practical proofs:

Theorem 2.1.7. *The property of a set F being closed is equivalent to the following statement: If a point y is such that for every open set U containing it we have $U \cap F \neq \{\}$ then this point is contained in F.*

Proof. We first proof that if a set is closed then the statement holds. Consider a closed set F and a point $y \notin F$ s.t. every open set containing y has nonempty intersection with F. But the complement F^c also is such a set, which is a clear contradiction. Now assume the above statement for a set F and further assume F is not closed. Its complement F^c is thus not open. Now consider the *interior* of this set: $\text{int}(F^c) := \cup\{U : U \subset F^c \text{ and } U \text{ open}\}$, i.e. the biggest open set contained within F^c.

Hence there must be a point y which is in F^c but is not in its interior, else F^c would be equal to its interior, i.e. would be open. We further must be able to find an open set U that contains y but is also contained in F^c, else y would be an element of F. A contradiction. □

Next we define *open covers*, a concept that is very important in developing a theory of manifolds:

Definition 2.1.8. An **open cover** of a topological space $\mathcal{M}$ is a (not necessarily countable) collection of open sets $\{U_i\}_{i\mathcal{I}}$ s.t. their union contains $\mathcal{M}$. A **finite open cover** is a finite collection of open sets that cover $\mathcal{M}$. We say that an open cover is **reducible** to a finite cover if we can find a finite number of elements in the open cover whose union still contains $\mathcal{M}$.

And connected to this definition we state what it means for a topological space to be *compact*. This is a rather strong property that some of the manifolds treated in here have, for example the Stiefel manifold (see Chapter 2.9).

Definition 2.1.9. A topological space $\mathcal{M}$ is called **compact** if every open cover is reducible to a finite cover.

A very important result from general topology is that continuous functions preserve compactness[2]:

Theorem 2.1.10. *Consider a continuous function $f : \mathcal{M} \to \mathcal{N}$ and a compact set $K \in \mathcal{M}$. Then $f(K)$ is also compact.*

Proof. Consider an open cover of $f(K)$: $\{U_i\}_{i\in\mathcal{I}}$. Then $\{f^{-1}\{U_i\}\}_{i\in\mathcal{I}}$ is an open cover of K and hence reducible to a finite cover $\{f^{-1}\{U_i\}\}_{i\in\{i_1,\dots,i_n\}}$. But then $\{U_i\}_{i\in\{i_1,\dots,i_n}$ also covers $f(K)$. □

Moreover compactness is a property that is *inherited* by closed subspaces:

Theorem 2.1.11. *A closed subset of a compact space is compact.*

Proof. Call the closed set F and consider an open cover of this set: $\{U\}_{i\in\mathcal{I}}$. Then this open cover combined with F^c is an open cover for the entire compact space, hence reducible to a finite cover. □

If a set is contained in a Hausdorff space and is also compact we have:

Theorem 2.1.12. *A compact subset of a Hausdorff space is closed.*

Proof. Consider a compact subset K. If K is not closed, then there has to be a point $y \notin K$ s.t. every open set containing y intersects K. Because the surrounding space is Hausdorff we can now find the following two collections of open sets: $\{(U_z, U_{z,y} : U_z \cap U_{z,y} = \{\})\}_{z\in K}$. The open cover $\{U_z\}_{z\in K}$ is then reducible to a finite cover $\{U_z\}_{z\in\{z_1,\dots,z_n\}}$. The intersection $\cap_{z\in z_1,\dots,z_n} U_{z,y}$ is then an open set that contains y but has no intersection with K. A contraction. □

This last theorem we will use in proofing the inverse function theorem (see Chapter 2.4):

Theorem 2.1.13. *If $\mathcal{M}$ is compact and $\mathcal{N}$ is Hausdorff, then the inverse of a continuous injective function $f : \mathcal{M} \to \mathcal{N}$ is again continuous, i.e. $f(V)$ is an open set in $\mathcal{N}$ for $V \in \mathcal{T}$.*

Proof. We can equivalently show that every closed set is mapped to a closed set. First consider the set $K \in \mathcal{M}$. Its image is again compact and hence closed because $\mathcal{N}$ is Hausdorff. □

[2]We also say that *compactness is a topological property* [34].

We further define what it means for a set to be *dense*:

Definition 2.1.14. A set U is called **dense in** D, where $U \subset D$ if the *closure of* U, i.e. the smallest closed set containing U, also contains D.

We will come back to the notion of *denseness* when talking about the universal approximation theorem for SympNets (see Chapter 8.5).

2.2 (Topological) Metric Spaces

A metric space is a certain class of a topological space where the topology is *induced through a metric*. We define this notion now:

Definition 2.2.1. A **metric** on a topological space $\mathcal{M}$ is a mapping $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ such that the following three conditions hold:

1. $d(x, y) = 0 \iff x = y$ for every $x, y \in \mathcal{M}$, i.e. the distance between two points is zero if and only if they are the same,
2. $d(x, y) = d(y, x)$,
3. $d(x, z) \leq d(x, y) + d(y, z)$.

The second condition is referred to as *symmetry* and the third condition is referred to as the *triangle inequality*.

We give some examples of metric spaces that are relevant for us:

Example 2.2.2. The real line $\mathbb{R}$ with the metric defined by the absolute distance between two points: $d(x, y) = |y - x|$.

Example 2.2.3. The vector space $\mathbb{R}^n$ with the *Euclidean distance* $d_2(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$.

Example 2.2.4. The space of continuous functions $\mathcal{C} = \{f : (-\epsilon, \epsilon) \to \mathbb{R}^n\}$ with the metric $d_\infty(f_1, f_2) = \sup_{t\in(-\epsilon,\epsilon)}|f_1(t) - f_2(t)|$.

Proof. We have to show the triangle inequality:

$$\begin{aligned} d_\infty(d_1, d_3) = \sup_{t\in(-\epsilon,\epsilon)}|f_1(t) - f_3(t)| &\leq \sup_{t\in(-\epsilon,\epsilon)}(|f_1(t) - f_2(t)| + |f_2(t) - f_3(t)|) \\ &\leq \sup_{t\in(-\epsilon,\epsilon)}|f_1(t) - f_2(t)| + \sup_{t\in(-\epsilon,\epsilon)}|f_1(t) - f_2(t)|. \end{aligned}$$

This shows that d_∞ is indeed a metric. □

Example 2.2.5. Any Riemannian manifold is a metric space.

This last example shows that *metric spaces need not be vector spaces*, i.e. spaces for which we can define a metric but not addition of two elements. This will be discussed in more detail in the section on Riemannian manifolds (see Chapter 2.7).

Complete Metric Spaces

To define *complete metric spaces* we first need the definition of a *Cauchy sequence*.

Definition 2.2.6. A **Cauchy sequence** is a sequence $(a_n)_{n\in\mathbb{N}}$ for which, given any $\epsilon > 0$, we can find an integer N such that $d(a_n, a_m) < \epsilon$ for all $n, m \geq N$.

Now we can give the definition of a *complete metric space*:

Definition 2.2.7. A **complete metric space** is one for which every Cauchy sequence converges.

Completeness of the real numbers is most often seen as an axiom and therefore stated without proof. This also implies completeness of $\mathbb{R}^n$ [37].

2.3 (Topological) Vector Spaces

Vector Spaces are, like metric spaces, topological spaces which we endow with additional structure.

Definition 2.3.1. A **vector space** $\mathcal{V}$ is a topological space for which we define an operation called *addition* and denoted by $+$ and an operation called *scalar multiplication* (by elements of $\mathbb{R}$) denoted by $x \mapsto ax$ for $x \in \mathcal{V}$ and $x \in \mathbb{R}$ for which the following hold for all $x, y, z \in \mathcal{V}$ and $a, b \in \mathbb{R}$:

1. $x + (y + z) = (x + y) + z$,
2. $x + y = y + x$,
3. $\exists 0 \in \mathcal{V}$such that $x + 0 = x$,
4. $\exists - x \in \mathcal{V}$ such that $x + (-x) = 0$,
5. $a(ax) = (ab)x$,
6. $1x = x$ for $1 \in \mathbb{R}$,
7. $a(x + y) = ax + ay$,
8. $(a + b)x = ax + bx$.

The first law is known as *associativity*, the second one as *commutativity* and the last two ones are known as *distributivity*.

The topological spaces $\mathbb{R}$ and $\mathbb{R}^n$ are (almost) trivially vector spaces. The same is true for many function spaces. One of the special aspects of `GeometricMachineLearning` is that it can deal with spaces that are not vector spaces, but manifolds. All vector spaces are however manifolds.

2.4 Foundational Theorems for Differential Manifolds

Here we state and proof all the theorems necessary to define differential manifolds (see Chapter 2.5). All these theorems (including proofs) can be found in e.g. [35].

The Fixed-Point Theorem

The fixed-point theorem will be used in the proof of the inverse function theorem below and the existence-and-uniqueness theorem (see Chapter 2.6).

Theorem 2.4.1 (Banach Fixed-Point Theorem)**.** *A function $f : U \to U$ defined on an open subset U of a complete metric vector space $\mathcal{V} \supset U$ that is contractive, i.e. $|f(z) - f(y)| \leq q|z - y|$ with $q < 1$, has a unique fixed point y^* such that $f(y^*) = y^*$. Further y^* can be found by taking any $y \in U$ through $y^* = \lim_{m\to\infty} f^m(y)$.*

Proof. Fix a point $y \in U$. We proof that the sequence $(f^m(y))_{m\in\mathbb{N}}$ is Cauchy and because $\mathcal{V}$ is a complete metric space, the limit of this sequence exists. Take $\tilde{m} > m$ and we have

$$\begin{aligned}
|f^{\tilde{m}}(y) - f^m(y)| &\leq \sum_{i=m}^{\tilde{m}-1} |f^{i+1}(y) - f^i(y)| \\
&\leq \sum_{i=m}^{\tilde{m}-1} q^i |f(y) - y| \\
&\leq \sum_{i=m}^{\infty} q^i |f(y) - y| = (f(y) - y)\left(\frac{q}{1-q} - \sum_{i=1}^{m-1} q^i\right) \\
&= (f(y) - y)\left(\frac{q}{1-q} - \frac{q - q^m}{q-1}\right) = (f(y) - y)\frac{q^{m+1}}{1-q}.
\end{aligned}$$

And the sequence is clearly Cauchy. □

Note that we stated the fixed-point theorem for arbitrary complete metric spaces here, not just for $\mathbb{R}^n$. For the section on manifolds (see Chapter 2.5) we only need the theorem for $\mathbb{R}^n$, but for the existence-and-uniqueness theorem (see Chapter 2.6) we need the statement for more general spaces.

The Inverse Function Theorem

The *inverse function theorem* gives a sufficient condition on a vector-valued function to be invertible in a neighborhood of a specific point. This theorem serves as a basis for the *implicit function theorem* and further for the preimage theorem (see Chapter 2.5) and is critical in developing a theory of manifolds (see Chapter 2.5). Here we first state the theorem and then give a proof.

Theorem 2.4.2 (Inverse function theorem)**.** *Consider a vector-valued differentiable function $F : \mathbb{R}^N \to \mathbb{R}^N$ and assume its Jacobian is non-degenerate at a point $x \in \mathbb{R}^N$. Then there exists a neighborhood U that contains $F(x)$ and on which F is invertible, i.e. $\exists H : U \to \mathbb{R}^N$ s.t. $\forall y \in U$, $F \circ H(y) = y$ and H is differentiable.*

Proof. Consider a mapping $F : \mathbb{R}^N \to \mathbb{R}^N$ and assume its Jacobian has full rank at point x, i.e. $\det F'(x) \neq 0$. We further assume that $F(x) = 0$, $F'(x) = \mathbb{I}$ and $x = 0$. Now consider a ball around x whose radius r we do not yet fix and two points y and z in that ball: $y, z \in B(r)$. We further introduce the function $G(y) := y - F(y)$. By the *mean value theorem* we have

$$|G(y)| = |G(y) - x| = |G(y) - G(x)| \leq |y - x| \sup_{0<t<1} ||G'(x + t(y - x))||,$$

where $||\cdot||$ is the *operator norm*. Because $t \mapsto G'(x + t(y - x))$ is continuous and $G'(x) = 0$ there must exist an r s.t. $\forall t \in [0, 1]$, $||G'(x + t(y - x))|| < 1/2$. We have for any element $y \in B(r)$: $|G(y)| \leq ||G'(y)|| \cdot |y| < |y|/2$, so $G(B(r)) \subset B(r/2)$. We further define $G_z(y) := z + G(y)$; this map is contractive on $B(r)$ (for $z \in B(r/2)$): $|G_z(y)| \leq |z| + |G(y) - x| < q < 1$ and therefore has a fixed point: $y^* = G_z(y^*) = z + y^* - F(y^*)$ and we obtain $z = F(y^*)$. The inverse (which we call $H : F(B(r/2)) \to B(r)$) is also continuous by the last theorem presented in the section on basic topological concepts. We now proof that the derivative of H at $F(x) = 0$ exists and that it is equal to $F'(H(z))^{-1}$. For this we let $\eta \in F(B(r/2))$ go to zero. We further define $\xi = F(z)$ and $h = H(\xi + \eta) - z$:

$$\begin{aligned} |H(\xi + \eta) - H(\xi) - F'(z)^{-1}\eta| &= |h - F'(x)^{-1}\xi| = |h - F'(z)^{-1}(F(z + h) - \xi)| \\ &\leq ||F'(z)^{-1}|| \cdot |F'(z)h - F(z + h) + \xi| \\ &\leq ||F'(z)^{-1}|| \cdot |h| \cdot \left| F'(z)\frac{h}{|h|} - \frac{F(z + h) - \xi}{|h|} \right|, \end{aligned}$$

and the rightmost expression is bounded because of the mean value theorem: $F(z + h) - F(z) \leq sup_{0<t<1}|h| \cdot ||F'(z + th)||$. □

The Implicit Function Theorem

This theorem is a direct consequence of the inverse function theorem.

Theorem 2.4.3 (Implicit Function Theorem)**.** *Given a function $f : \mathbb{R}^{n+m} \to \mathbb{R}^n$ whose derivative at $x \in \mathbb{R}^{n+m}$ has full rank, we can find a map $h : U \to \mathbb{R}^{n+m}$ for a neighborhood $U \ni (f(x), x_{n+1}, \ldots, x_{n+m})$ such that $f \circ h$ is a projection onto the first factor, i.e. $f(h(x_1, \ldots, x_{n+m})) = (x_1, \ldots, x_n)$.*

Proof. Consider the map $x = (x_1, \ldots, x_{n+m}) = (f(x), x_{n+1}, \ldots, x_{n+m})$. The derivative of this map is clearly of full rank if $f'(x)$ is of full rank and therefore invertible in a neighborhood around $(f(x), x_{n+1}, \ldots, x_{n+m})$. We call this inverse map h. We then see that $f \circ h$ is a projection. □

The implicit function will be used to proof the preimage theorem (see Chapter 2.5) which we use as a basis to construct all the manifolds in `GeometricMachineLearning`.

2.5 (Matrix) Manifolds

Manifolds are topological spaces that locally look like vector spaces. In the following we restrict ourselves to finite-dimensional smooth[1] manifolds. In this section we routinely denote points on a manifold by lower case letters like x, y and z if we speak about general properties and by upper case letters like A and B if we talk about specific examples of matrix manifolds. All manifolds that can be used to build neural networks in `GeometricMachineLearning`, such as the Stiefel manifold (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10) are matrix manifolds.

Definition 2.5.1. A **finite-dimensional smooth manifold** of dimension n is a second-countable Hausdorff space $\mathcal{M}$ for which $\forall x \in \mathcal{M}$ we can find a neighborhood U, that contains x, and a corresponding homeomorphism $\varphi_U : U \cong W \subset \mathbb{R}^n$ where W is an open subset. The homeomorphisms φ_U are referred to as *coordinate charts*. If two such coordinate charts overlap, i.e. if $U_1 \cap U_2 \neq \{\}$, then the map $\varphi_{U_2} \circ \varphi_{U_1}^{-1}$ has to be C^∞ on $varphi_{U_1}(U_1 \cap U_2)$. We call the collection of coordinate charts $\{\varphi_U\}_{U \subset \mathcal{M}}$ an **atlas** for $\mathcal{M}$.

[1] *Smooth* here means C^∞.

One example of a manifold that is also important for `GeometricMachineLearning` is the Lie group[2] of orthonormal matrices $SO(N)$. Before we can proof that $SO(N)$ is a manifold we first need the *preimage theorem*.

The Preimage Theorem

The preimage theorem is crucial for treating the specific manifolds that are part of `GeometricMachineLearning`; the preimage theorem gives spaces like the Stiefel manifold (see Chapter 2.9) the structure of a manifold. Before we can state the preimage theorem we need another definition:

Definition 2.5.2. Consider a smooth mapping $g : \mathcal{M} \to \mathcal{N}$ from one manifold to another. A point $y \in \mathcal{N}$ is called **regular point of** g if $\forall x \in g^{-1}\{y\}$ the map $T_x g : T_x\mathcal{M} \to T_y\mathcal{N}$ is surjective.

Remark 2.5.3. Here we already used the notation $T_y\mathcal{N}$ to denote the *tangent space to* $\mathcal{N}$ *at* y. We will explain what we mean by this precisely below. For now we simply view $T_y\mathcal{N}$ as *something that is homemorphic to* $\mathbb{R}^m$ and the *tangent map* $T_x g$ we will simply view as $(\psi \circ g \circ \varphi^{-1})'(varphi(x))$, where φ is a coordinate chart as x and ψ is a coordinate chart at y. In the examples we give below $\mathcal{M}$ and $\mathcal{N}$ will simply be vector spaces, and g will be differential map between vector spaces whose derivative at $x \in f^{-1}\{y\}$ (for a regular point y) is surjective. For a vector space $\mathcal{V}$ we furthermore have $T_x\mathcal{V} = \mathcal{V}$.

We now state the preimage theorem:

Theorem 2.5.4 (Preimage Theorem)**.** *Consider a smooth map $g : \mathcal{M} \to \mathcal{N}$ from one manifold to another (we assume the dimensions of the two manifolds to be $m+n$ and m respectively). Then the preimage of a regular point y of $\mathcal{N}$ is a submanifold of $\mathcal{M}$. Furthermore the codimension of $g^{-1}\{y\}$ is equal to the dimension of $\mathcal{N}$ and the tangent space $T_x(g^{-1}\{y\})$ is equal to the kernel of $T_x g$.*

Proof. Because $\mathcal{N}$ has manifold structure we can find a chart $\psi_U : U \to \mathbb{R}^m$ for some neighborhood U that contains y. We further consider a point $x \in g^{-1}\{y\}$ and a chart around it $\varphi_V : V \to \mathbb{R}^{m+n}$. By the implicit function theorem we can then find a mapping h that turns $\psi_U \circ g \circ \varphi_V^{-1}$ into a projection $(x_1, \ldots, x_{n+m}) \mapsto (x_{n+1}, \ldots, x_{n+m})$. We now consider the neighborhood $V_1 \times \{0\} = \varphi(V \cup f^{-1}\{y\})$ for $\varphi(V) = V_1 \times V_2$ with the coordinate chart $(x_1, \ldots, x_n) \mapsto \varphi(x_1, \ldots, x_n, 0, \ldots, 0)$. As this map is also smooth by the implicit function theorem this proofs our assertion. □

We now give some examples of manifolds that can be constructed this way:

Example 2.5.5. The group $SO(N)$ is a Lie group (i.e. has manifold structure). $SO(N)$ has dimension $N(N-1)/2$.

Proof. The vector space $\mathbb{R}^{N\times N}$ clearly has manifold structure. The group $SO(N)$ is equivalent to one of the level sets of the mapping: $g : \mathbb{R}^{N\times N} \to \mathcal{S}(N), A \mapsto A^TA - \mathbb{I}$, i.e. it is the component of $f^{-1}\{\mathbb{I}\}$ that contains $\mathbb{I}$; the image of f is contained in $\mathcal{S}(N)$, the symmetric matrices of size $N \times N$. We still need to proof that $\mathbb{I}$ is a regular point of g, i.e. that for $A \in SO(N)$ the mapping $T_A g$ is surjective. This means that $\forall B \in \mathcal{S}(N)\ \exists C \in \mathbb{R}^{N\times N}$ s.t. $C^TA + A^TC = B$. The element $C = \frac{1}{2}AB \in \mathcal{R}^{N\times N}$ satisfies this property. The dimension of $SO(N)$ is $N(N-1)/2$ as $\dim(\mathcal{S}(N)) = N(N+1)/2$. □

Similarly we can also proof:

[2] Lie groups are manifolds that also have a *group structure*, i.e. there is an operation $\mathcal{M} \times \mathcal{M} \to \mathcal{M}, (a, b) \mapsto ab$ s.t. $(ab)c = a(bc)$ and there exists a neutral element$e\mathcal{M}$ s.t. $ae = a\ \forall a \in \mathcal{M}$ as well as an (for every a) inverse element a^{-1} s.t. $a(a^{-1}) = e$. The neutral element e we refer to as $\mathbb{I}$ when dealing with matrix manifolds.

Example 2.5.6. The sphere $S^n := \{x \in \mathbb{R}^{n+1} : x^T x = 1\}$ is a manifold of dimension n.

Proof. Take $g(x) = x^T x - 1$ and proceed as in the case of $SO(N)$. □

Note that both these manifolds, $SO(N)$ and S^n are matrix manifolds, i.e. an element of $\mathcal{M}$ can be written as an element of $\mathbb{R}^{N\times N}$ in the first case and $\mathbb{R}^{(n+1)\times 1}$ in the second case. The additional conditions we impose on these manifolds are $A^T A = \mathbb{I}$ in the first case and $x^T x = 1$ in the second case. Both of these manifolds belong to the category of Stiefel manifolds (see Chapter 2.9) and therefore also to the bigger category of matrix manifolds, i.e. every element of $SO(N)$ and of S^n can be represented as a matrix on which further conditions are imposed (e.g orthogonality).

The Immersion Theorem

The immersion theorem, similarly to the preimage theorem, gives another way of constructing a manifold based on a non-linear function.

Theorem 2.5.7 (Immersion Theorem)**.** *Consider a differentiable function $\mathcal{R} : \mathcal{N} \to \mathcal{M}$ with $\dim(\mathcal{M}) = N > n = \dim(\mathcal{N})$ tangent mapping $T_x\mathcal{R}$ has full rank at every point $x \in \mathcal{N}$. Then $\mathcal{R}(\mathcal{N})$ is a manifold* immersed *in $\mathcal{M}$.*

The proof is again based on the inverse function theorem (see Chapter 2.4).

Proof. Consider a point $x \in \mathcal{N}$, a coordinate chart φ around x and a coordinate chart ψ around $f(x)$. We now define the function

$$F : (x_1, \dots, x_N) \mapsto (\psi \circ f \circ \varphi^{-1}(x_1, \dots, x_n), x_{n+1}, \dots, x_N).$$

By the inverse function theorem we can find an inverse of F for a neighborhood around the point $(x_1, \dots, x_n, 0, \dots, 0) \in \mathbb{R}^N$. We call this neighborhood $V = V_1 \times V_2$ and the inverse H. We now constrain V to the set $V_1 \times 0$, which is isomorphic to a neighborhood around x in $\mathbb{R}^n$. We then have in this neighborhood:

$$H(\psi \circ f \circ \varphi^{-1}(x_1, \dots, x_n), 0, \dots, 0) = (x_1, \dots, x_n, 0, \dots, 0),$$

And we can take

$$y \mapsto \pi \circ H(\psi(y), 0 \dots, 0)$$

as our coordinate chart. $\pi : \mathbb{R}^N \to \mathbb{R}^n$ is the projection onto the first n coordinates. □

We will use the immersion theorem when discussing the symplectic solution manifold (see Chapter 4.5).

Tangent Spaces

We already alluded to tangent spaces when talking about the preimage and the immersion theorems. Here we will give a precise definition. A tangent space can be seen as the *collection of all possible velocities a curve can take at a point on a manifold*. For this consider a manifold $\mathcal{M}$ and a point x on it and the collection of C^∞ curves through x:

Definition 2.5.8. A mapping $\gamma : (-\epsilon, \epsilon) \to \mathcal{M}$ that is C^∞ and for which we have $\gamma(0) = x$ is called a C^∞ **curve through** x.

The tangent space of $\mathcal{M}$ at x is the collection of the first derivatives of all γ:

Definition 2.5.9. The **tangent space** of $\mathcal{M}$ at x is the collection of all C^∞ curves at x modulo the equivalence class $\gamma_1 \sim \gamma_2 \iff \gamma_1'(0) = \gamma_2'(0)$. It is denoted by $T_x\mathcal{M}$.

As is customary we write $[\gamma]$ for the equivalence class of γ and this is by definition equivalent to $\gamma'(0)$. The tangent space $T_x\mathcal{M}$ can be shown to be homeomorphic[3] to $\mathbb{R}^n$ where n is the dimension of the manifold $\mathcal{M}$. If the homeomorphism is constructed through the coordinate chart (φ, U) we call it $\varphi'(x)$ or simply[4] φ'. If we are given a map $g : \mathcal{M} \to \mathcal{N}$ we further define $T_x g = (\varphi')^{-1} \circ (\varphi \circ g \circ \psi^{-1})' \circ \psi'$, i.e. a smooth map between two manifolds $\mathcal{M}$ and $\mathcal{N}$ induces a smooth map between the tangent spaces $T_x\mathcal{M}$ and $T_{g(x)}\mathcal{N}$.

We want to demonstrate this principle of constructing the tangent space from curves through the example of S^2. We consider the following curves:

1. $\gamma_1(t) = \begin{pmatrix} 0 \\ \sin(t) \\ \cos(t) \end{pmatrix}$,

2. $\gamma_2(t) = \begin{pmatrix} \sin(t) \\ 0 \\ \cos(t) \end{pmatrix}$,

3. $\gamma_3(t) = \begin{pmatrix} \exp(-t^2/2)t\sin(t) \\ \exp(-t^2/2)t\cos(t) \\ \sqrt{1-(t^2)\exp(-t^2)} \end{pmatrix}$.

We now plot the manifold S^2, the three curves described above and the associated tangent vectors (visualized as arrows). Note that the tangent vectors induced by γ_1 and γ_3 are the same; for these curves we have $\gamma_1 \sim \gamma_3$ and the tangent vectors of those two curves coincide:

[3] Note that we have not formally defined addition for $T_x\mathcal{M}$. This can be done through the definition $[\gamma]+[\beta] = [\alpha]$ where α is any C^∞ curve through x that satisfies $\alpha'(0) = \beta(0) + \gamma(0)$. Note that we can always find such an α by the existence and uniqueness theorem (see Chapter 2.6).

[4] We will further discuss this when we introduce the tangent bundle (see Chapter 2.5).

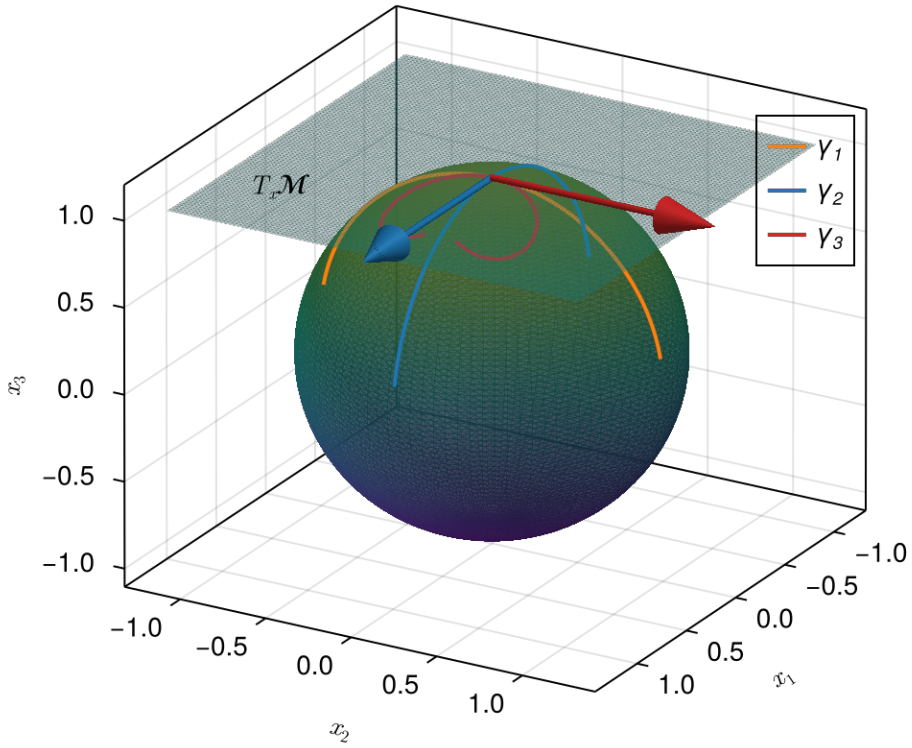


Figure 2.1: Visualization of how the tangent space is constructed.

The tangent space $T_x\mathcal{M}$ for

$$x = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

is also shown.

Vector Fields

A time-independent vector field[5] is an object that specifies a velocity for every point on a domain. We first give the definition of a vector field on the vector space $\mathbb{R}^n$ and limit ourselves here to C^∞ vector fields:

Definition 2.5.10. A **vector field** on $\mathbb{R}^n$ is a smooth map $X : \mathbb{R}^n \to \mathbb{R}^n$.

The definition of a vector field on a manifold is not much more complicated:

[5] Also called *ordinary differential equation* (ODE).

Definition 2.5.11. A **vector field** on $\mathcal{M}$ is a map X defined on $\mathcal{M}$ such that $X(x) \in T_x\mathcal{M}$ and $\varphi' \circ X \circ (\varphi)^{-1}$ is smooth for any coordinate chart (φ, U) that contains x.

In the section on the existence-and-uniqueness theorem (see Chapter 2.6) we show that every vector field has a unique solution given an initial condition; i.e. given a point $x \in \mathcal{M}$ and a vector field X we can find a curve γ such that $\gamma(0) = x$ and $\gamma'(t) = X(\gamma(t))$ for all t in some interval $(-\epsilon, \epsilon)$.

The Tangent Bundle

To each manifold $\mathcal{M}$ we can associate another manifold which we call the *tangent bundle* and denote by $T\mathcal{M}$. The points on this manifold are:

$$T\mathcal{M} = \{(x, v_x) : x \in \mathcal{M},\ v_x \in T_x\mathcal{M}\}.$$

Coordinate charts on this manifold can be constructed in a straightforward manner; for every coordinate chart φ_U the map $\varphi'_U(x)$ gives a homeomorphism between $T_x\mathcal{M}$ and $\mathbb{R}^n$ for any $x \in U$. We can then find a neighborhood of any point (x, v_x) by taking $\pi^{-1}(U) = \{(x, v_x) : x \in U, v_x \in T_x\mathcal{M}\}$ and this neighborhood is isomorphic to $\mathbb{R}^{2n}$ via $(x, v_x) \mapsto (\varphi_U(x), \varphi'(x)v_x)$. The geodesic spray (see Chapter 2.7) is an important vector field defined on $T\mathcal{M}$.

Library Functions

`GeometricMachineLearning.Manifold` – Type.

```
Manifold <: AbstractMatrix
```

A manifold in `GeometricMachineLearning` is a sutype of `AbstractMatrix`. All manifolds are matrix manifolds and therefore stored as matrices. More details can be found in the docstrings for the `StiefelManifold` and the `GrassmannManifold`.

source

`Base.rand` – Method.

```
rand(manifold_type, N, n)
```

Draw random elements from the Stiefel and the Grassmann manifold.

Because both of these manifolds are compact spaces we can sample them uniformly [38].

Examples

When we call ...

```
using GeometricMachineLearning
using GeometricMachineLearning: _round # hide
import Random
Random.seed!(123)

N, n = 5, 3
```

```
Y = rand(StiefelManifold{Float32}, N, n)
_round(Y; digits = 5) # hide

# output

5×3 StiefelManifold{Float32, Matrix{Float32}}:
 -0.27575   0.32991   0.77275
 -0.62485  -0.33224  -0.0686
 -0.69333   0.36724  -0.18988
 -0.09295  -0.73145   0.46064
  0.2102    0.33301   0.38717
```

... the sampling is done by first allocating a random matrix of size $N \times n$ via `Y = randn(Float32, N, n)`.

We then perform a QR decomposition `Q, R = qr(Y)` with the `qr` function from the `LinearAlgebra` package (this is using Householder reflections internally).

The final output are then the first `n` columns of the `Q` matrix.

source

`Base.rand` – Method.

```
rand(backend, manifold_type, N, n)
```

Draw random elements for a specific device.

Examples

Random elements of the manifold can be allocated on GPU. Call ...

```
rand(CUDABackend(), StiefelManifold{Float32}, N, n)
```

... for drawing elements on a `CUDA` device.

source

2.6 The Existence-And-Uniqueness Theorem

The *existence-and-uniqueness theorem*, also known as the *Picard-Lindelöf theorem*, *Picard's existence theorem* or the *Cauchy-Lipschitz theorem* gives a proof of the existence of solutions for ODEs. Here we state the existence-and-uniqueness theorem for manifolds as vector spaces are just a special case of this. Its proof relies on the Banach fixed-point theorem (see Chapter 2.4)[1].

Theorem 2.6.1 (Existence-And-Uniqueness Theorem)**.** *Let X a vector field on the manifold $\mathcal{M}$ that is differentiable at x. Then we can find an $\epsilon > 0$ and a unique curve $\gamma : (-\epsilon, \epsilon) \to \mathcal{M}$ such that $\gamma'(t) = X(\gamma(t))$.*

[1] It has to be noted that the proof given here is not entirely self-contained. The proof of the fundamental theorem of calculus, i.e. the proof of the existence of an antiderivative of a continuous function [37], is omitted for example.

Proof. We consider a ball around a point $x \in \mathcal{M}$ with radius r that we pick such that the ball $B(x,r)$ fits into the U of some coordinate chart φ_U; we further use X and $\varphi' \circ X \circ \varphi^{-1}$ interchangeably in this proof. We then define $L := \sup_{y,z \in B(x,r)} |X(y) - X(z)|/|y - z|$. Note that this L is always finite because X is bounded and differentiable. We now define the map $\Gamma : C^\infty((-\epsilon,\epsilon),\mathbb{R}^n) \to C^\infty((-\epsilon,\epsilon),\mathbb{R}^n)$ (for some ϵ that we do not yet fix) as

$$\Gamma\gamma(t) = x + \int_0^t X(\gamma(s))ds,$$

i.e. Γ maps C^∞ curves through x into C^∞ curves through x. We further have with the norm $||\gamma||_\infty = \sup_{t\in(-\epsilon,\epsilon)}|\gamma(t)|$:

$$\begin{aligned}
||\Gamma(\gamma_1 - \gamma_2)||_\infty &= \sup_{t\in(-\epsilon,\epsilon)} \left| \int_0^t (X(\gamma_1(s)) - X(\gamma_2(s)))ds \right| \\
&\leq \sup_{t\in(-\epsilon,\epsilon)} \int_0^t |X(\gamma_1(s)) - X(\gamma_2(s))|ds \\
&\leq \sup_{t\in(-\epsilon,\epsilon)} \int_0^t L|\gamma_1(s) - \gamma_2(s)|ds \\
&\leq \epsilon L \cdot \sup_{t\in(-\epsilon,\epsilon)} |\gamma_1(t) - \gamma_2(t)|,
\end{aligned}$$

and we see that Γ is a contractive mapping if we pick ϵ small enough and we can hence apply the fixed-point theorem. So there has to exist a C^∞ curve through x that we call γ^* such that

$$\gamma^*(t) = \int_0^t X(\gamma^*(s))ds,$$

and this γ^* is the curve we were looking for. Its uniqueness is guaranteed by the fixed-point theorem. □

For all the problems we discuss here we can extend the integral curves of X from the finite interval $(-\epsilon,\epsilon)$ to all of $\mathbb{R}$. The solution γ we call an *integral curve* or *flow* of the vector field (ODE).

Time-Dependent Vector Fields

We proved the theorem above for a time-independent vector field X, but it also holds for time-dependent vector fields, i.e. for mappings of the form:

$$X : [0,T] \times \mathcal{M} \to TM.$$

The proof for this case proceeds analogously to the case of the time-independent vector field; to apply the proof we simply have to *extend* the vector field to (here written for a specific coordinate chart φ_U):

$$\bar{X} : [0,T] \times \mathbb{R}^n \to \mathbb{R}^{n+1},\ (t, x_1, \ldots, x_n) \mapsto (1, X(x_1, \ldots, x_n)).$$

More details on this can be found in e.g. [35]. For `GeometricMachineLearning` time-dependent vector fields are important because many of the optimizers we are using (such as the Adam optimizer (see Chapter 6.1)) can be seen as approximating the flow of a time-dependent vector field.

2.7 Riemannian Manifolds

A Riemannian manifold is a manifold $\mathcal{M}$ that we endow with a mapping g that smoothly[1] assigns a metric (see Chapter 2.2) g_x to each tangent space $T_x\mathcal{M}$. By a slight abuse of notation we will also refer to this g as a *metric*.

After having defined a metric g we can *associate a length* to each curve $\gamma : [0, t] \to \mathcal{M}$ through:

$$L(\gamma) = \int_0^t \sqrt{g_{\gamma(s)}(\gamma'(s), \gamma'(s))} ds.$$

This L turns $\mathcal{M}$ into a metric space:

Definition 2.7.1. The **metric on a Riemannian manifold** $\mathcal{M}$ is

$$d(x, y) = \inf_{\substack{\gamma(0) = x \text{ and} \\ \gamma(t) = y}} L(\gamma),$$

where t can be chosen arbitrarily.

If a curve is minimal with respect to the function L we call it the *shortest curve* or a geodesic. So we say that a curve $\gamma : [0, t] \to \mathcal{M}$ is a geodesic if there is no shorter curve that can connect two points in $\gamma([0, t])$, i.e.

$$d(\gamma(t_i), \gamma(t_f)) = \int_{t_i}^{t_f} \sqrt{g_{\gamma(s)}(\gamma'(s), \gamma'(s))} ds,$$

for any $t_i, t_f \in [0, t]$.

An important result of Riemannian geometry states that there exists a vector field X on $T\mathcal{M}$, called the *geodesic spray*, whose integral curves are derivatives of geodesics. We formalize this statement as a theorem in the next section.

Geodesic Sprays and the Exponential Map

To every Riemannian manifold we can naturally associate a vector field called the *geodesic spray* or *geodesic equation*. For our purposes it is enough to state that this vector field is unique and well-defined [39].

The important property of the geodesic spray is

[1] Smooth here refers to the fact that $g : \mathcal{M} \to$ (Space of Metrics) has to be a smooth map. But in order to discuss this in detail we would have to define a topology on the space of metrics. A more detailed discussion can be found in [35, 36, 39].

Theorem 2.7.2. *Given an initial point x and an initial velocity v_x, an integral curve for the geodesic spray is of the form $t \mapsto (\gamma_{v_x}(t), \gamma'_{v_x}(t))$ where γ_{v_x} is a geodesic. We further have the property that the integral curve for the geodesic spray for an initial point x and an initial velocity $\eta \cdot v_x$ (where η is a scalar) is of the form $t \mapsto (\gamma_{\eta \cdot v_x}(t), \gamma'_{\eta \cdot v_x}(t)) = (\gamma_{v_x}(\eta \cdot t), \eta \cdot \gamma'_{v_x}(\eta \cdot t))$.*

It is therefore customary to introduce the *exponential map* $\exp : T_x\mathcal{M} \to \mathcal{M}$ as

$$\exp(v_x) := \gamma_{v_x}(1),$$

and we see that $\gamma_{v_x}(t) = \exp(t \cdot v_x)$. In `GeometricMachineLearning` we denote the exponential map by `geodesic` to avoid confusion with the matrix exponential map[2] which is called as `exp` in `Julia`. So we use the definition:

$$\texttt{geodesic}(x, v_x) \equiv \exp(v_x).$$

We give an example of using this function here:

```
Y = rand(StiefelManifold, 3, 1)

v = 2 * rand(3, 1)
Δ = v - Y * (v' * Y)

# plot a sphere with radius one and origin 0
surface!(ax, Main.sphere(1., [0., 0., 0.])...; alpha = .5, transparency = true)

point_vec = ([Y[1]], [Y[2]], [Y[3]])
scatter!(ax, point_vec...; color = morange, marker = :star5, markersize = 30)

arrow_vec = ([Δ[1]], [Δ[2]], [Δ[3]])
arrows!(ax, point_vec..., arrow_vec...; color = mred, linewidth = .02)
```

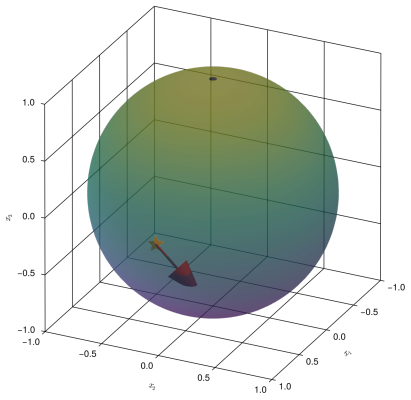


Figure 2.2: A tangent vector on $\mathcal{M}$ determines a direction.

[2]The Riemannian exponential map and the matrix exponential map coincide for many matrix Lie groups.

We now solve the geodesic spray for $\eta \cdot \Delta$ for $\eta = 0.1, 0.2, \ldots, 5.5$ with the function `geodesic` and plot the corresponding points:

```
Δ_increments = [Δ * η for η in 0.1 : 0.1 : 5.5]

Y_increments = [geodesic(Y, Δ_increment) for Δ_increment in Δ_increments]

for Y_increment in Y_increments
    scatter!(ax, [Y_increment[1]], [Y_increment[2]], [Y_increment[3]];
        color = mred)
end
```

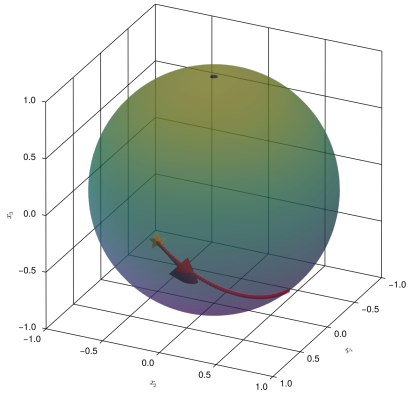


Figure 2.3: Solving the geodesic gives a path along the tangent vector, equivalent to a straight line in flat space.

A geodesic can be seen as the *equivalent of a straight line* on a manifold. Also note that we drew a random element form `StiefelManifold` here, and not from S^2. This is because the category of Stiefel manifolds (see Chapter 2.9) is more general than the category of spheres S^n: $St(1,3) \simeq S^2$.

The Riemannian Gradient

The *Riemannian gradient* is essential when talking about optimization on manifolds.

Definition 2.7.3. The Riemannian gradient of a function $L : \mathcal{M} \to \mathbb{R}$ is a vector field $\text{grad}^g L$ (or simply $\text{grad} L$) for which we have

$$g_x(\text{grad}^g L(x), v_x) = (\nabla_{\varphi_U(x)}(L \circ \varphi_U^{-1}))^T \varphi_U'(v_x),$$

for all $v_x \in T_x\mathcal{M}$. In the expression above φ_U is some coordinate chart defined in a neighborhood U around x.

In the definition above ∇ indicates the *Euclidean gradient*:

$$\nabla_x f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

We can also describe the Riemannian gradient through differential curves:

Definition 2.7.4. The Riemannian gradient of L is a vector field $\text{grad}^g L$ for which

$$g_x(\text{grad}^g L(x), \dot{\gamma}(0)) = \frac{d}{dt} L(\gamma(t)),$$

where γ is a C^∞ curve through x.

By the *non degeneracy* of g the Riemannian gradient always exists [36]. In the following we will also write $\text{grad}^g L(x) = \text{grad}^g_x L = \text{grad}_x L$. We will give specific examples of this when discussing the Stiefel manifold (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10).

Gradient Flows and Riemannian Optimization

In `GeometricMachineLearning` we can include weights in neural networks that are part of a manifold. Training such neural networks amounts to *Riemannian optimization* and hence solving the *gradient flow* equation. The gradient flow equation is given by

$$X(x) = -\text{grad}_x L.$$

Solving this gradient flow equation will then lead us to a local minimum on $\mathcal{M}$. This will be elaborated on when talking about optimizers (see Chapter 5.1). In practice we cannot solve the gradient flow equation directly and have to rely on approximations. The most straightforward approximation (and one that serves as a basis for all the optimization algorithms in `GeometricMachineLearning`) is to take the point $(x, X(x))$ as an initial condition for the geodesic spray and then solve the ODE for a small time step. Such an update rule, i.e.

$$x^{(t)} \leftarrow \gamma_{X(x^{(t-1)})}(\Delta t) \text{ with } \Delta t \text{ the time step},$$

we call the *gradient optimization scheme*.

Library Functions

`GeometricMachineLearning.geodesic` – Method.

```
geodesic(Y::Manifold, Δ)
```

Take as input an element of a manifold `Y` and a tangent vector in `Δ` in the corresponding tangent space and compute the geodesic (exponential map).

In different notation: take as input an element x of $\mathcal{M}$ and an element of $T_x\mathcal{M}$ and return `geodesic`$(x, v_x) = \exp(v_x)$.

Examples

```
using GeometricMachineLearning

Y = StiefelManifold([1. 0. 0.;]' |> Matrix)
Δ = [0. .5 0.;]' |> Matrix
Y₂ = geodesic(Y, Δ)

Y₂' * Y₂ ≈ [1.;]

# output

true
```

Implementation

Internally this `geodesic` method calls `geodesic(::StiefelLieAlgHorMatrix)`.

source

2.8 Homogeneous Spaces

Homogeneous spaces are very important in `GeometricMachineLearning` as we can generalize existing neural network optimizers from vector spaces to such homogenous spaces. They are intricately linked to the notion of a *Lie Group* and its *Lie Algebra*[1].

Definition 2.8.1. A **homogeneous space** is a manifold $\mathcal{M}$ on which a Lie group G acts transitively, i.e.

$$\forall X, Y \in \mathcal{M} \quad \exists A \in G \text{ s.t. } AX = Y.$$

Now fix a distinct element $E \in \mathcal{M}$; we will refer to this as the *canonical element* or `StiefelProjection`. We can also establish an isomorphism between $\mathcal{M}$ and the quotient space $G/\sim$ with the equivalence relation:

$$A_1 \sim A_2 \iff A_1E = A_2E.$$

Note that this is independent of the chosen E.

The tangent spaces of $\mathcal{M}$ are of the form $T_Y\mathcal{M} = \mathfrak{g} \cdot Y$, i.e. can be fully described through its Lie algebra. Based on this we can perform a splitting of $\mathfrak{g}$ into two parts:

Definition 2.8.2. A **splitting of the Lie algebra** $\mathfrak{g}$ at an element of a homogeneous space Y is a decomposition into a **vertical** and a **horizontal** component, denoted by $\mathfrak{g} = \mathfrak{g}^{\text{ver},Y} \oplus \mathfrak{g}^{\text{hor},Y}$ such that

[1] Recall that a Lie group is a manifold that also has group structure. We say that a Lie group G *acts* on a manifold $\mathcal{M}$ if there is a map $G \times \mathcal{M} \to \mathcal{M}$ such that $(ab)x = a(bx)$ for $a, b \in G$ and $x \in \mathcal{M}$. For us the Lie algebra belonging to a Lie group, denoted by $\mathfrak{g}$, is the tangent space to the identity element $T_{\mathbb{I}}G$.

1. The **vertical component** $\mathfrak{g}^{\text{ver},Y}$ is the kernel of the map $\mathfrak{g} \to T_Y\mathcal{M}, V \mapsto VY$, i.e. $\mathfrak{g}^{\text{ver},Y} = \{V \in \mathfrak{g} : VY = 0\}$.
2. The **horizontal component** $\mathfrak{g}^{\text{hor},Y}$ is the orthogonal complement of $\mathfrak{g}^{\text{ver},Y}$ in $\mathfrak{g}$. It is isomorphic to $T_Y\mathcal{M}$.

Orthogonal complement means that $\forall V \in \mathfrak{g}^{\text{ver},Y}$ and $\forall B \in \mathfrak{g}^{\text{hor},Y}$ we have $\langle V, B\rangle = 0$ for some metric $\langle\cdot,\cdot\rangle$ defined on $\mathfrak{g}$.

We will refer to the isomorphism from $T_Y\mathcal{M}$ to $\mathfrak{g}^{\text{hor},Y}$ by Ω. We will give explicit examples of Ω below. The metric $\langle\cdot,\cdot\rangle$ on $\mathfrak{g}$ further induces a Riemannian metric on $\mathcal{M}$:

$$g_Y(\Delta_1, \Delta_2) = \langle \Omega(Y, \Delta_1), \Omega(Y, \Delta_2)\rangle \text{ for } \Delta_1, \Delta_2 \in T_Y\mathcal{M}.$$

Two examples of homogeneous spaces implemented in `GeometricMachineLearning` are the Stiefel manifold (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10). The Lie group $SO(N)$ acts transitively on both of these manifolds, i.e. turns them into homogeneous spaces. We give its Lie algebra as an example here:

Example 2.8.3. The Lie algebra of $SO(N)$ are the skew-symmetric matrices $\mathfrak{so}(N) := \{V \in \mathbb{R}^{N\times N} : V^T + V = 0\}$ and the canonical metric associated with it is simply $(V_1, V_2) \mapsto \frac{1}{2}\text{Tr}(V_1^T V_2)$.

2.9 The Stiefel Manifold

The Stiefel manifold $St(n, N)$ is the space of all orthonormal frames in $\mathbb{R}^{N\times n}$, i.e. matrices $Y \in \mathbb{R}^{N\times n}$ s.t. $Y^TY = \mathbb{I}_n$. It can also be seen as $SO(N)$ modulo an equivalence relation: $A \sim B \iff AE = BE$ for

$$E = \begin{bmatrix} \mathbb{I}_n \\ \mathbb{O} \end{bmatrix} \in St(n, N),$$

which is the canonical element of the Stiefel manifold that we call `StiefelProjection`. In words: the first n columns of A and B are the same. We also use this principle to draw random elements from the Stiefel manifold.

Remark 2.9.1. Drawing random elements from the Stiefel (and the Grassmann) manifold is done by first calling `rand(N, n)` (i.e. drawing from a normal distribution) and then performing a QR decomposition. We then take the first n columns of the Q matrix to be an element of the Stiefel manifold.

The tangent space to the element $Y \in St(n, N)$ can be determined by considering C^∞ curves on $SO(N)$ through $\mathbb{I}$. We write those curves as $t \mapsto A(t)$. Because $SO(N)$ acts transitively on $St(n, N)$ each C^∞ curve on $St(n, N)$ through Y can be written as $A(t)Y$ and we get:

$$T_Y St(n, N) = \{BY : B \in \mathfrak{g}\} = \{\Delta \in \mathbb{R}^{N\times n} : \Delta^TY + Y^T\Delta = \mathbb{O}\},$$

where the last equality[2] can be established through the isomorphism:

$$\Omega : T_Y St(n,N) \to \mathfrak{g}^{\text{hor},Y}, \Delta \mapsto (\mathbb{I} - \frac{1}{2}YY^T)\Delta Y^T - Y\Delta^T(\mathbb{I} - \frac{1}{2}YY^T).$$

That this is an isomorphism can be easily checked:

$$\Omega(\Delta)Y = (\mathbb{I} - \frac{1}{2}YY^T)\Delta - \frac{1}{2}Y\Delta^T Y = \Delta.$$

This isomorphism is implemented in `GeometricMachineLearning`:

```
using GeometricMachineLearning: Ω
Y = rand(StiefelManifold, 5, 3)
Δ = rgrad(Y, rand(5, 3))
Ω(Y, Δ) * Y.A ≈ Δ
```

```
true
```

The function `rgrad`, which maps $\mathbb{R}^{N\times n}$ to $T_Y St(n,N)$ is introduced below. We can now also introduce the Riemannian metric on $St(n,N)$:

$$g_Y(\Delta_1, \Delta_2) = \text{Tr}\left(\frac{1}{2}\Omega(\Delta_1)^T\Omega(\Delta_2)\right) = \text{Tr}(\Delta_1^T(\mathbb{I} - \frac{1}{2}YY^T)\Delta_2).$$

We can check that this is true:

```
using LinearAlgebra: tr
Δ₂ = rgrad(Y, rand(5, 3))
.5 * tr(Ω(Y, Δ)' * Ω(Y, Δ₂)) ≈ metric(Y, Δ, Δ₂)
```

```
true
```

The Riemannian Gradient for the Stiefel Manifold

We defined the Riemannian gradient (see Chapter 2.7) to be a vector field $\text{grad}^g L$ such that it is *compatible with the Riemannian metric* in some sense; the definition we gave relied on an explicit coordinate chart. We can also express the Riemannian gradient for matrix manifolds by not relying on an explicit coordinate representation (which would be computationally expensive) [40].

[2] Note that we can easily check $\{BY : B \in \mathfrak{g}\} \subset \{\Delta \in \mathbb{R}^{N\times n} : \Delta^T Y + Y^T\Delta = \mathbb{O}\}$. The isomorphism is used to proof $\{\Delta \in \mathbb{R}^{N\times n} : \Delta^T Y + Y^T\Delta = \mathbb{O}\} \subset \{BY : B \in \mathfrak{g}\}$ as $\Omega(\Delta) \in \mathfrak{g}^{\text{hor},Y} \subset \mathfrak{g}$ and $\Delta = \Omega(\Delta)Y$.

Definition 2.9.2. Given a Riemannian matrix manifold $\mathcal{M}$ we define the **Riemannian gradient** of $L : \mathcal{M} \to \mathbb{R}$ at Y, called $\text{grad}_Y L \in T_Y\mathcal{M}$, as the unique element of $T_Y\mathcal{M}$ such that for any other $\Delta \in T_Y\mathcal{M}$ we have

$$\text{Tr}((\nabla L)^T \Delta) = g_Y(\text{grad}_Y L, \Delta),$$

where Tr indicates the usual matrix trace.

For the Stiefel manifold the Riemannian gradient is given by:

$$\text{grad}_Y L = \nabla_Y L - Y(\nabla_Y L)^T Y =: \texttt{rgrad}(Y, \nabla_Y L),$$

where $\nabla_Y L$ refers to the Euclidean gradient, i.e.

$$[\nabla_Y L]_{ij} = \frac{\partial L}{\partial y_{ij}}.$$

The Euclidean gradient ∇L can in practice be obtained with an AD routine (see Chapter B.3). We then use the function `rgrad` to map $\nabla_Y L$ from $\mathbb{R}^{N\times n}$ to $T_Y St(n, N)$. We can check that this mapping indeed produces the Riemannian gradient[3]:

```
using LinearAlgebra: tr

Y = rand(StiefelManifold, 5, 3)
∇L = rand(5, 3)
gradL = rgrad(Y, ∇L)
Δ = rgrad(Y, rand(5, 3))

metric(Y, gradL, Δ) ≈ tr(∇L' * Δ)
```

```
true
```

2.10 The Grassmann Manifold

The Grassmann manifold is closely related to the Stiefel manifold, and an element of the Grassmann manifold can be represented through an element of the Stiefel manifold (but not vice-versa). An element of the Grassmann manifold $Gr(n, N)$ is a vector subspace $\subset \mathbb{R}^N$ of dimension n. Each such subspace (i.e. element of the Grassmann manifold) can be represented by a full-rank matrix $A \in \mathbb{R}^{N\times n}$ and we identify two elements with the following equivalence relation:

$$A_1 \sim A_2 \iff \exists C \in \mathbb{R}^{n\times n} \text{ s.t. } A_1 C = A_2.$$

[3]Here we are testing with a randomly drawn element $\Delta \in T_Y\mathcal{M}$.

The resulting manifold is of dimension $n(N-n)$. One can find a parametrization of the manifold the following way: Because the matrix Y has full rank, there have to be n independent rows in it: $i_1, \dots, i_n$. For simplicity assume that $i_1 = 1, i_2 = 2, \dots, i_n = n$ and call the matrix made up of these columns C. Then the mapping to the coordinate chart is: YC^{-1} and the last $N-n$ columns are the coordinates.

We can also define the Grassmann manifold based on the Stiefel manifold since elements of the Stiefel manifold are already full-rank matrices. In this case we have the following equivalence relation (for $Y_1, Y_2 \in St(n, N)$):

$$Y_1 \sim Y_2 \iff \exists C \in SO(n) \text{ s.t. } Y_1 C = Y_2.$$

In `GeometricMachineLearning` elements of the Grassmann manifold are drawn the same way as elements of the Stiefel manifold:

```
rand(GrassmannManifold{Float32}, 5, 3)
```

```
5×3 GrassmannManifold{Float32, Matrix{Float32}}:
 -0.0461715   0.0483366  -0.564308
  0.873444   -0.25248     0.244718
  0.45861     0.549674   -0.274063
  0.0829837   0.55703    -0.270157
  0.133247   -0.567005   -0.688167
```

The Riemannian Gradient of the Grassmann Manifold

Obtaining the Riemannian Gradient for the Grassmann manifold is slightly more difficult than it is in the case of the Stiefel manifold [40]. Since the Grassmann manifold can be obtained from the Stiefel manifold through an equivalence relation, we can however use this as a starting point.

Theorem 2.10.1. *The Riemannian gradient of a function L defined on the Grassmann manifold can be written as*

$$\text{grad}^{Gr}_Y L \simeq \nabla_Y L - Y Y^T \nabla_Y L,$$

where $\nabla_Y L$ is again the Euclidean gradient.

Proof. In a first step we identify charts on the Grassmann manifold to make dealing with it easier. For this consider the following open cover of the Grassmann manifold.

$$\{\mathcal{U}_W\}_{W \in St(n,N)} \quad \text{where} \quad \mathcal{U}_W = \{\text{span}(Y) : \det(W^T Y) \neq 0\}.$$

We can find a canonical bijective mapping from the set $\mathcal{U}_W$ to the set $\mathcal{S}_W := \{Y \in \mathbb{R}^{N \times n} : W^T Y = \mathbb{I}_n\}$:

$$\sigma_W : \mathcal{U}_W \to \mathcal{S}_W,\ \mathcal{Y} = \text{span}(Y) \mapsto Y(W^TY)^{-1} =: \hat{Y}.$$

That σ_W is well-defined is easy to see: Consider YC with $C \in \mathbb{R}^{n\times n}$ non-singular. Then $YC(W^TYC)^{-1} = Y(W^TY)^{-1} = \hat{Y}$. With this isomorphism we can also find a representation of elements of the tangent space:

$$T_{\mathcal{Y}}\sigma_W : T_{\mathcal{Y}}Gr(n,N) \to T_{\hat{Y}}\mathcal{S}_W.$$

We give an explicit representation of this isomorphism; because the map σ_W does not care about the representation of span(Y) we can perform the variations in $St(n,N)$. We write the variations as $Y(t) \in St(n,N)$ for $t \in (-\varepsilon, \varepsilon)$. We also set $Y(0) = Y$ and hence

$$\frac{d}{dt}Y(t)(W^TY(t))^{-1} = (\dot{Y}(0) - Y(W^TY)^{-1}W^T\dot{Y}(0))(W^TY)^{-1},$$

where $\dot{Y}(0) \in T_YSt(n,N)$. Also note note that we have $T_{\mathcal{Y}}\mathcal{U}_W = T_{\mathcal{Y}}Gr(n,N)$ because $\mathcal{U}_W$ is an open subset of $Gr(n,N)$. We thus can identify the tangent space $T_{\mathcal{Y}}Gr(n,N)$ with the following set:

$$T_{\hat{Y}}\mathcal{S}_W = \{(\Delta - YW^T\Delta)(W^TY)^{-1} : Y \in St(n,N) \text{ s.t. } \text{span}(Y) = \mathcal{Y} \text{ and } \Delta \in T_YSt(n,N)\}.$$

Further note that we can pick any element W to construct the charts for a neighborhood around the point $\mathcal{Y} \in Gr(n,N)$ as long as we have $\det(W^TY) \neq 0$ for $\text{span}(Y) = \mathcal{Y}$. We hence take $W = Y$ and get the identification:

$$T_{\mathcal{Y}}Gr(n,N) \equiv \{\Delta - YY^T\Delta : Y \in St(n,N) \text{ s.t. } \text{span}(Y) = \mathcal{Y} \text{ and } \Delta \in T_YSt(n,N)\},$$

which is very easy to handle computationally (we simply store and change the matrix Y that represents an element of the Grassmann manifold). In this representation the Riemannian gradient is then

$$\text{grad}^{Gr}_{\mathcal{Y}}L = \text{grad}^{St}_YL - YY^T\text{grad}^{St}_YL = \nabla_YL - YY^T\nabla_YL,$$

where grad^{St}_YL is the Riemannian gradient of the Stiefel manifold at Y. We proved our assertion. □

Library Functions

`GeometricMachineLearning.StiefelManifold` – Type.

```
StiefelManifold <: Manifold
```

An implementation of the Stiefel manifold [3]. The Stiefel manifold is the collection of all matrices $Y \in \mathbb{R}^{N\times n}$ whose columns are orthonormal, i.e.

$$St(n, N) = \{Y : Y^T Y = \mathbb{I}_n\}.$$

The Stiefel manifold can be shown to have manifold structure (as the name suggests) and this is heavily used in `GeometricMachineLearning`. It is further a compact space. More information can be found in the docstrings for `rgrad(::StiefelManifold, ::AbstractMatrix)` and `metric(::StiefelManifold, ::AbstractMatrix, ::AbstractMatrix)`.

source

`GeometricMachineLearning.StiefelProjection` – Type.

```
StiefelProjection(backend, T, N, n)
```

Make a matrix of the form $\begin{bmatrix}\mathbb{I} & \mathbb{O}\end{bmatrix}^T$ for a specific backend and data type.

An array that essentially does `vcat(I(n), zeros(N-n, n))` with GPU support.

Extended help

An instance of `StiefelProjection` should technically also belong to `StiefelManifold`.

source

`GeometricMachineLearning.GrassmannManifold` – Type.

```
GrassmannManifold <: Manifold
```

The `GrassmannManifold` is based on the `StiefelManifold`.

source

`GeometricMachineLearning.metric` – Method.

```
metric(Y::StiefelManifold, Δ₁::AbstractMatrix, Δ₂::AbstractMatrix)
```

Compute the dot product for `Δ₁` and `Δ₂` at `Y`.

This uses the canonical Riemannian metric for the Stiefel manifold:

$$g_Y : (\Delta_1, \Delta_2) \mapsto \mathrm{Tr}(\Delta_1^T(\mathbb{I} - \frac{1}{2}YY^T)\Delta_2).$$

source

`GeometricMachineLearning.rgrad` – Method.

```
rgrad(Y::StiefelManifold, ∇L::AbstractMatrix)
```

Compute the Riemannian gradient for the Stiefel manifold at `Y` based on `∇L`.

Here $Y \in St(N,n)$ and $\nabla L \in \mathbb{R}^{N\times n}$ is the Euclidean gradient.

The function computes the Riemannian gradient with respect to the canonical metric: `metric(::StiefelManifold, ::AbstractMatrix, ::AbstractMatrix)`.

The precise form of the mapping is:

$$\texttt{rgrad}(Y, \nabla L) \mapsto \nabla L - Y(\nabla L)^T Y$$

Note the property $Y^T\texttt{rgrad}(Y, \nabla L) \in \mathcal{S}_{\text{skew}}(n)$.

Examples

```
using GeometricMachineLearning

Y = StiefelManifold([1 0 ; 0 1 ; 0 0; 0 0])
Δ = [1 2; 3 4; 5 6; 7 8]
rgrad(Y, Δ)

# output

4×2 Matrix{Int64}:
 0  -1
 1   0
 5   6
 7   8
```

source

`GeometricMachineLearning.metric` – Method.

```
metric(Y::GrassmannManifold, Δ₁::AbstractMatrix, Δ₂::AbstractMatrix)
```

Compute the metric for vectors `Δ₁` and `Δ₂` at `Y`.

The representation of the Grassmann manifold is realized as a *quotient space of the Stiefel manifold*.

The metric for the Grassmann manifold is:

$$g_Y^{Gr}(\Delta_1, \Delta_2) = g_Y^{St}(\Delta_1, \Delta_2) = \text{Tr}(\Delta_1^T(\mathbb{I} - YY^T)\Delta_2) = \text{Tr}(\Delta_1^T\Delta_2),$$

where we used that $Y^T\Delta_i$ for $i = 1, 2$.

source

`GeometricMachineLearning.rgrad` – Method.

```
rgrad(Y::GrassmannManifold, ∇L::AbstractMatrix)
```

Compute the Riemannian gradient for the Grassmann manifold at `Y` based on `∇L`.

Here Y is a representation of $\text{span}(Y) \in Gr(n, N)$ and $\nabla L \in \mathbb{R}^{N \times n}$ is the Euclidean gradient.

This gradient has the property that it is orthogonal to the space spanned by Y.

The precise form of the mapping is:

$$\texttt{rgrad}(Y, \nabla L) \mapsto \nabla L - YY^T \nabla L.$$

Note the property $Y^T \text{rgrad}(Y, \nabla L) = \mathbb{O}$.

Also see `rgrad(::StiefelManifold, ::AbstractMatrix)`.

Examples

```
using GeometricMachineLearning

Y = GrassmannManifold([1 0 ; 0 1 ; 0 0; 0 0])
Δ = [1 2; 3 4; 5 6; 7 8]
rgrad(Y, Δ)

# output

4×2 Matrix{Int64}:
 0  0
 0  0
 5  6
 7  8
```

source

`GeometricMachineLearning.Ω` – Method.

```
Ω(Y::StiefelManifold{T}, Δ::AbstractMatrix{T}) where T
```

Perform *canonical horizontal lift* for the Stiefel manifold:

$$\Delta \mapsto (\mathbb{I} - \frac{1}{2} YY^T) \Delta Y^T - Y \Delta^T (\mathbb{I} - \frac{1}{2} YY^T).$$

Internally this performs

```
SkewSymMatrix(2 * (I(n) - .5 * Y * Y') * Δ * Y')
```

It uses `SkewSymMatrix` to save memory.

Examples

```
using GeometricMachineLearning
E = StiefelManifold(StiefelProjection(5, 2))
Δ = [0. -1.; 1. 0.; 2. 3.; 4. 5.; 6. 7.]
GeometricMachineLearning.Ω(E, Δ)

# output

5×5 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0  -1.0  -2.0  -4.0  -6.0
 1.0   0.0  -3.0  -5.0  -7.0
 2.0   3.0   0.0  -0.0  -0.0
 4.0   5.0   0.0   0.0  -0.0
 6.0   7.0   0.0   0.0   0.0
```

Note that the output of Ω is a skew-symmetric matrix, i.e. an element of $\mathfrak{g}$.

source

`GeometricMachineLearning.Ω` – Method.

```
Ω(Y::GrassmannManifold{T}, Δ::AbstractMatrix{T}) where T
```

Perform the *canonical horizontal lift* for the Grassmann manifold:

$$\Delta \mapsto \Omega^{St}(\Delta),$$

where Ω^{St} is the canonical horizontal lift for the Stiefel manifold.

```
using GeometricMachineLearning
E = GrassmannManifold(StiefelProjection(5, 2))
Δ = [0. 0.; 0. 0.; 2. 3.; 4. 5.; 6. 7.]
GeometricMachineLearning.Ω(E, Δ)

# output

5×5 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0  -0.0  -2.0  -4.0  -6.0
 0.0   0.0  -3.0  -5.0  -7.0
 2.0   3.0   0.0  -0.0  -0.0
 4.0   5.0   0.0   0.0  -0.0
 6.0   7.0   0.0   0.0   0.0
```

source

2.11 Global Tangent Spaces

In `GeometricMachineLearning` standard neural network optimizers are generalized to homogeneous spaces (see Chapter 2.8) by leveraging the special structure of the tangent spaces of this class of manifolds. When we introduced homogeneous spaces we already talked about that every tangent space to a homogeneous space $T_Y\mathcal{M}$ is of the form:

$$T_Y\mathcal{M} = \mathfrak{g} \cdot Y := \{AY : A \in \mathfrak{g}\}.$$

We then have a decomposition of $\mathfrak{g}$ into a vertical part $\mathfrak{g}^{\text{ver},Y}$ and a horizontal part $\mathfrak{g}^{\text{hor},Y}$ and the horizontal part is isomorphic to $T_Y\mathcal{M}$ via:

$$\mathfrak{g}^{\text{hor},Y} = \{\Omega(\Delta) : \Delta \in T_Y\mathcal{M}\}.$$

We now identify a special element $E \in \mathcal{M}$ and designate the horizontal component $\mathfrak{g}^{\text{hor},E}$ as our *global tangent space*. We will refer to this global tangent space by $\mathfrak{g}^{\text{hor}}$. We can now find a transformation from any $\mathfrak{g}^{\text{hor},Y}$ to $\mathfrak{g}^{\text{hor}}$ and vice-versa (these spaces are isomorphic).

Theorem 2.11.1. *Let $A \in G$ an element such that $AE = Y$. Then we have*

$$A^{-1} \cdot \mathfrak{g}^{\text{hor},Y} \cdot A = \mathfrak{g}^{\text{hor}},$$

i.e. for every element $B \in \mathfrak{g}^{\text{hor}}$ we can find a $B^Y \in \mathfrak{g}^{\text{hor},Y}$ s.t. $B = A^{-1}B^YA$ (and vice-versa).

Proof. We first show that for every $B^Y \in \mathfrak{g}^{\text{hor},Y}$ the element $A^{-1}B^YA$ is in $\mathfrak{g}^{\text{hor}}$. First note that $A^{-1}B^YA \in \mathfrak{g}$ by a fundamental theorem of Lie group theory (closedness of the Lie algebra under adjoint action). Now assume that $A^{-1}B^YA$ is not fully contained in $\mathfrak{g}^{\text{hor}}$, i.e. it also has a vertical component. So we would lose information when performing $A^{-1}B^YA \mapsto A^{-1}B^YAE = A^{-1}B^YY$, but this contradicts the fact that $B^Y \in \mathfrak{g}^{\text{hor},Y}$. We now have to proof that for every $B \in \mathfrak{g}^{\text{hor}}$ we can find an element in $\mathfrak{g}^{\text{hor},Y}$ such that this element is mapped to B. By a argument similar to the one above we can show that $ABA^{-1} \in \mathfrak{g}^{\text{hor},Y}$ and this element maps to B. Proofing that the map is injective is now trivial. □

We should note that we have written all Lie group and Lie algebra actions as simple matrix multiplications, like $AE = Y$. For some Lie groups and Lie algebras, as the Lie group of isomorphisms on some domain $\mathcal{D}$, this notation may not be appropriate [41]. These Lie groups are however not relevant for what we use in `GeometricMachineLearning` and we will stick to regular matrix notation.

Global Sections

Note that the theorem above requires us to find an element $A \in G$ such that $AE = Y$. We will call such a mapping $\lambda : \mathcal{M} \to G$ a *global section*[1].

Definition 2.11.2. We call a mapping λ from a homogeneous space $\mathcal{M}$ to its associated Lie group G a **global section** if $\forall Y \in \mathcal{M}$ it satisfies:

$$\lambda(Y)E = Y,$$

where E is the distinct element of the homogeneous space.

[1] Global sections are also crucial for parallel transport (see Chapter 5.3) in `GeometricMachineLearning`. A global section is first updated, i.e. $\Lambda^{(t)} \leftarrow \text{update}(\Lambda^{(t-1)})$; and on the basis of this we then update the element of the manifold $Y \in \mathcal{M}$ and the tangent vector $\Delta \in T\mathcal{M}$.

Note that in general global sections are not unique because the rank of G is in general greater than that of $\mathcal{M}$. We give an example of how to construct such a global section for the Stiefel and the Grassmann manifolds below.

The Global Tangent Space for the Stiefel Manifold

We now discuss the specific form of the global tangent space for the Stiefel manifold (see Chapter 2.9). We pick as distinct element E (which build by calling `StiefelProjection`):

$$E = \begin{bmatrix} \mathbb{I}_n \\ \mathbb{O} \end{bmatrix} \in St(n, N).$$

Based on this, elements of the vector space $\mathfrak{g}^{\text{hor},E} =: \mathfrak{g}^{\text{hor}}$ are:

$$\bar{B} = \begin{pmatrix} A & B^T \\ B & \mathbb{O} \end{pmatrix},$$

where A is a skew-symmetric matrix of size $n \times n$ and B is an arbitrary matrix of size $(N-n) \times n$. Arrays of type $\mathfrak{g}^{\text{hor},E} \equiv \mathfrak{g}^{\text{hor}}$ are implemented in `GeometricMachineLearning` under the name `StiefelLieAlgHorMatrix`.

We can call this with e.g. a skew-symmetric matrix A and an arbitrary matrix B:

```
N, n = 5, 2

A = rand(SkewSymMatrix, n)
```

```
2×2 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0        -0.0488926
 0.0488926   0.0
```

```
B = rand(N - n, n)
```

```
3×2 Matrix{Float64}:
 0.0454726  0.829864
 0.323484   0.0908533
 0.979466   0.500674
```

The constructor is then called as follows:

```
B̄ = StiefelLieAlgHorMatrix(A, B, N, n)
```

```
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0        -0.0488926  -0.0454726  -0.323484   -0.979466
 0.0488926   0.0        -0.829864   -0.0908533  -0.500674
 0.0454726   0.829864    0.0         0.0         0.0
 0.323484    0.0908533   0.0         0.0         0.0
 0.979466    0.500674    0.0         0.0         0.0
```

We can also call it with a matrix of shape $N \times N$:

```
B̃₂ = Matrix(B̃) # note that this does not have any special structure

StiefelLieAlgHorMatrix(B̃₂, n)
```

```
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, SubArray{Float64, 2,
↪  Matrix{Float64}, Tuple{UnitRange{Int64}, UnitRange{Int64}}, false}}:
 0.0        -0.0488926  -0.0454726  -0.323484   -0.979466
 0.0488926   0.0        -0.829864   -0.0908533  -0.500674
 0.0454726   0.829864    0.0         0.0         0.0
 0.323484    0.0908533   0.0         0.0         0.0
 0.979466    0.500674    0.0         0.0         0.0
```

Or we can call it on $T_E\mathcal{M} \subset \mathbb{R}^{N \times n}$, i.e. a matrix of shape $N \times n$:

```
E = StiefelProjection(N, n)
```

```
5×2 StiefelProjection{Float64, Matrix{Float64}}:
 1.0  0.0
 0.0  1.0
 0.0  0.0
 0.0  0.0
 0.0  0.0
```

```
B̃₃ = B̃ * E

StiefelLieAlgHorMatrix(B̃₃, n)
```

```
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, SubArray{Float64, 2,
↪  Matrix{Float64}, Tuple{UnitRange{Int64}, UnitRange{Int64}}, false}}:
 0.0        -0.0488926  -0.0454726  -0.323484   -0.979466
 0.0488926   0.0        -0.829864   -0.0908533  -0.500674
 0.0454726   0.829864    0.0         0.0         0.0
 0.323484    0.0908533   0.0         0.0         0.0
 0.979466    0.500674    0.0         0.0         0.0
```

We now demonstrate how to map from an element of $\mathfrak{g}^{\mathrm{hor},Y}$ to an element of $\mathfrak{g}^{\mathrm{hor}}$:

```
using GeometricMachineLearning: Ω

Y = rand(StiefelManifold, N, n)
Δ = rgrad(Y, rand(N, n))
ΩΔ = Ω(Y, Δ)
λY = GlobalSection(Y)

λY_mat = Matrix(λY)

round.(λY_mat' * ΩΔ * λY_mat; digits = 3)
```

```
5×5 Matrix{Float64}:
 -0.0     0.086   0.808   0.459  -0.332
 -0.086  -0.0     0.294   0.666  -0.737
 -0.808  -0.294   0.0    -0.0     0.0
 -0.459  -0.666   0.0    -0.0    -0.0
  0.332   0.737  -0.0     0.0    -0.0
```

Performing this computation directly is computationally very inefficient however and the user is strongly discouraged to call `Matrix` on an instance of `GlobalSection`. The better option is calling `global_rep`:

```
_round(global_rep(λY, Δ); digits = 3)
```

```
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
  0.0     0.086  0.808  0.459  -0.332
 -0.086   0.0    0.294  0.666  -0.737
 -0.808  -0.294  0.0    0.0     0.0
 -0.459  -0.666  0.0    0.0     0.0
  0.332   0.737  0.0    0.0     0.0
```

Internally `GlobalSection` calls the function `GeometricMachineLearning.global_section` which does the following for the Stiefel manifold:

```
A = randn(N, N - n) # or the gpu equivalent
A = A - Y * (Y' * A)
Y⊥ = qr(A).Q[1:N, 1:(N - n)]
```

So we draw $(N-n)$ new columns randomly, subtract the part that is spanned by the columns of Y and then perform a QR composition on the resulting matrix. The Q part of the decomposition is a matrix of $(N-n)$ columns that is orthogonal to Y and is typically referred to as $Y_\perp$ [21, 25, 40]. We can easily check that this $Y_\perp$ is indeed orthogonal to Y.

Theorem 2.11.3. *The matrix $Y_\perp$ constructed with the algorithm above satisfies*

$$Y^T Y_\perp = \mathbb{O}_{n\times n},$$

and

$$(Y_\perp)^T Y_\perp = \mathbb{I}_n,$$

i.e. all the columns in the big matrix $[Y, Y_\perp] \in \mathbb{R}^{N\times N}$ *are mutually orthonormal and it therefore is an element of* $SO(N)$.

Proof. The second property is trivially satisfied because the Q component of a QR decomposition is an orthogonal matrix. For the first property note that $Y^T QR = \mathbb{O}$ is zero because we have subtracted the Y component from the matrix QR. The matrix $R \in \mathbb{R}^{N\times(N-n)}$ further has the property $[R]_{ij} = 0$ for $i > j$ and we have that

$$(Y^T Q)R = [r_{11}(Y^T Q)_{1\bullet}, r_{12}(Y^T Q)_{1\bullet} + r_{22}(Y^T Q)_{2\bullet}, \ldots, \sum_{i=1}^{N-n} r_{i(N-n)}(Y^T Q)_{i\bullet}].$$

Now all the coefficients r_{ii} are non-zero because the matrix we performed the QR decomposition on has full rank and we can see that if $(Y^T Q)R$ is zero $Y^T Q$ also has to be zero. □

The function `global_rep` furthermore makes use of the following:

$$\texttt{global\_rep}(Y) = \lambda(Y)^T \Omega(Y,\Delta)\lambda(Y) = EY^T\Delta E^T + \begin{bmatrix}\mathbb{O}\\ \bar{\lambda}^T\Delta E^T\end{bmatrix} - \begin{bmatrix}\mathbb{O} & E\Delta^T\bar{\lambda}\end{bmatrix},$$

where $\lambda(Y) = [Y, \bar{\lambda}]$.

Proof. We derive the expression above:

$$\begin{aligned}
\lambda(Y)^T\Omega(Y,\Delta)\lambda(Y) &= \lambda(Y)^T[(\mathbb{I} - \frac{1}{2}YY^T)\Delta Y^T - Y\Delta^T(\mathbb{I} - \frac{1}{2}YY^T)]\lambda(Y)\\
&= \lambda(Y)^T[(\mathbb{I} - \frac{1}{2}YY^T)\Delta E^T - Y\Delta^T(\lambda(Y) - \frac{1}{2}YE^T)]\\
&= \lambda(Y)^T\Delta E^T - \frac{1}{2}EY^T\Delta E^T - E\Delta^T\lambda(Y) + \frac{1}{2}E\Delta^T YE^T\\
&= \begin{bmatrix}Y^T\Delta E^T\\ \bar{\lambda}\Delta E^T\end{bmatrix} - \frac{1}{2}EY^T\Delta E - \begin{bmatrix}E\Delta^T Y & E\Delta^T\bar{\lambda}\end{bmatrix} + \frac{1}{2}E\Delta^T YE^T\\
&= \begin{bmatrix}Y^T\Delta E^T\\ \bar{\lambda}\Delta E^T\end{bmatrix} + E\Delta^T YE^T - \begin{bmatrix}E\Delta^T Y & E\Delta^T\bar{\lambda}\end{bmatrix}\\
&= EY^T\Delta E^T + E\Delta^T YE^T - E\Delta^T YE^T + \begin{bmatrix}\mathbb{O}\\ \bar{\lambda}\Delta E^T\end{bmatrix} - \begin{bmatrix}\mathbb{O} & E\Delta^T\bar{\lambda}\end{bmatrix}\\
&= EY^T\Delta E^T + \begin{bmatrix}\mathbb{O}_{n\times N}\\ \bar{\lambda}\Delta E^T\end{bmatrix} - \begin{bmatrix}\mathbb{O}_{N\times n} & E\Delta^T\bar{\lambda}\end{bmatrix},
\end{aligned}$$

which proofs our assertion. □

This expression of `global_rep` means we only need $Y^T\Delta$ and $\bar{\lambda}^T\Delta$ and this is what is used internally.

We now discuss the global tangent space for the Grassmann manifold. This is similar to the Stiefel case.

Global Tangent Space for the Grassmann Manifold

In the case of the Grassmann manifold we construct the global tangent space with respect to the distinct element $\mathcal{E} = \text{span}(E) \in Gr(n, N)$, where E is again the same matrix.

The tangent tangent space $T_{\mathcal{E}}Gr(n, N)$ can be represented through matrices:

$$\begin{pmatrix} 0 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & 0 \\ b_{11} & \cdots & b_{1n} \\ \cdots & \cdots & \cdots \\ b_{(N-n)1} & \cdots & b_{(N-n)n} \end{pmatrix}.$$

This representation is based on the identification $T_{\mathcal{E}}Gr(n, N) \to T_E\mathcal{S}_E$ that was discussed in the section on the Grassmann manifold (see Chapter 2.10)[2]. We use the following notation:

$$\mathfrak{g}^{\text{hor}} = \mathfrak{g}^{\text{hor},\mathcal{E}} = \left\{ \begin{pmatrix} 0 & -B^T \\ B & 0 \end{pmatrix} : B \in \mathbb{R}^{(N-n)\times n} \text{ is arbitrary} \right\}.$$

This is equivalent to the horizontal component of $\mathfrak{g}$ for the Stiefel manifold for the case when A is zero. This is a reflection of the rotational invariance of the Grassmann manifold: the skew-symmetric matrices A are connected to the group of rotations $O(n)$ which is factored out in the Grassmann manifold $Gr(n, N) \simeq St(n, N)/O(n)$. In `GeometricMachineLearning` we thus treat the Grassmann manifold as being embedded in the Stiefel manifold. In [21] viewing the Grassmann manifold as a quotient space of the Stiefel manifold is important for "feasibility" in "practical computations".

Library Functions

`GeometricMachineLearning.AbstractLieAlgHorMatrix` – Type.

```
AbstractLieAlgHorMatrix <: AbstractMatrix
```

`AbstractLieAlgHorMatrix` is a supertype for various horizontal components of Lie algebras. We usually call this $\mathfrak{g}^{\text{hor}}$.

See `StiefelLieAlgHorMatrix` and `GrassmannLieAlgHorMatrix` for concrete examples.

source

`GeometricMachineLearning.StiefelLieAlgHorMatrix` – Type.

[2] We derived the following expression for the Riemannian gradient of the Grassmann manifold: $\text{grad}^{Gr}_Y L = \nabla_Y L - YY^T\nabla_Y L$. The tangent space to the element $\mathcal{E}$ can thus be written as $\bar{B} - EE^T\bar{B}$ where $B \in \mathbb{R}^{N\times n}$ and the matrices in this tangent space have the desired form.

```
StiefelLieAlgHorMatrix(A::SkewSymMatrix, B::AbstractMatrix, N::Integer, n::Integer)
```

Build an instance of `StiefelLieAlgHorMatrix` based on a skew-symmetric matrix `A` and an arbitrary matrix `B`.

An element of StiefelLieAlgMatrix takes the form:

$$\begin{pmatrix} A & B^T \\ B & \mathbb{O} \end{pmatrix},$$

where A is skew-symmetric (this is `SkewSymMatrix` in `GeometricMachineLearning`).

Also see `GrassmannLieAlgHorMatrix`.

Extended help

`StiefelLieAlgHorMatrix` is the *horizontal component of the Lie algebra of skew-symmetric matrices* (with respect to the canonical metric).

The projection here is: $\pi : S \to SE$ where

$$E = \begin{bmatrix} \mathbb{I}_n \\ \mathbb{O}_{(N-n)\times n} \end{bmatrix}.$$

The matrix E is implemented under `StiefelProjection` in `GeometricMachineLearning`.

source

`GeometricMachineLearning.StiefelLieAlgHorMatrix` – Method.

```
StiefelLieAlgHorMatrix(D::AbstractMatrix, n::Integer)
```

Take a big matrix as input and build an instance of `StiefelLieAlgHorMatrix`.

The integer N in $St(n, N)$ is the number of rows of `D`.

Extended help

If the constructor is called with a big $N \times N$ matrix, then the projection is performed the following way:

$$\begin{pmatrix} A & B_1 \\ B_2 & D \end{pmatrix} \mapsto \begin{pmatrix} \text{skew}(A) & -B_2^T \\ B_2 & \mathbb{O} \end{pmatrix}.$$

The operation skew : $\mathbb{R}^{n\times n} \to \mathcal{S}_{\text{skew}}(n)$ is the skew-symmetrization operation. This is equivalent to calling of `SkewSymMatrix` with an $n \times n$ matrix.

This can also be seen as the operation:

$$D \mapsto \Omega(E, DE) = \text{skew}\left(2\left(\mathbb{I} - \frac{1}{2}EE^T\right)DEE^T\right).$$

Also see `GeometricMachineLearning.Ω`.

source

`GeometricMachineLearning.GrassmannLieAlgHorMatrix` – Type.

```
GrassmannLieAlgHorMatrix(B::AbstractMatrix, N::Integer, n::Integer)
```

Build an instance of `GrassmannLieAlgHorMatrix` based on an arbitrary matrix `B` of size $(N-n)\times n$.

`GrassmannLieAlgHorMatrix` is the *horizontal component of the Lie algebra of skew-symmetric matrices* (with respect to the canonical metric).

Extended help

The projection here is: $\pi : S \to SE/\sim$ where

$$E = \begin{bmatrix} \mathbb{I}_n \\ \mathbb{O}_{(N-n)\times n} \end{bmatrix},$$

and the equivalence relation is

$$V_1 \sim V_2 \iff \exists A \in \mathcal{S}_{\text{skew}}(n) \text{ such that } V_2 = V_1 + \begin{bmatrix} A \\ \mathbb{O} \end{bmatrix}$$

An element of GrassmannLieAlgMatrix takes the form:

$$\begin{pmatrix} \bar{\mathbb{O}} & B^T \\ B & \mathbb{O} \end{pmatrix},$$

where $\bar{\mathbb{O}} \in \mathbb{R}^{n\times n}$ and $\mathbb{O} \in \mathbb{R}^{(N-n)\times(N-n)}$.

source

`GeometricMachineLearning.GrassmannLieAlgHorMatrix` – Method.

```
GrassmannLieAlgHorMatrix(D::AbstractMatrix, n::Integer)
```

Take a big matrix as input and build an instance of `GrassmannLieAlgHorMatrix`.

The integer N in $Gr(n, N)$ here is the number of rows of `D`.

Extended help

If the constructor is called with a big $N \times N$ matrix, then the projection is performed the following way:

$$\begin{pmatrix} A & B_1 \\ B_2 & D \end{pmatrix} \mapsto \begin{pmatrix} \bar{\mathbb{O}} & -B_2^T \\ B_2 & \mathbb{O} \end{pmatrix}.$$

This can also be seen as the operation:

$$D \mapsto \Omega(E, DE - EE^TDE),$$

where Ω is the horizontal lift `GeometricMachineLearning.Ω`.

source

`GeometricMachineLearning.GlobalSection` – Type.

```
GlobalSection(Y)
```

Construct a global section for `Y`.

A global section λ is a mapping from a homogeneous space $\mathcal{M}$ to the corresponding Lie group G such that

$$\lambda(Y)E = Y,$$

Also see `apply_section` and `global_rep`.

Implementation

For an implementation of `GlobalSection` for a custom array (especially manifolds), the function `global_section` has to be generalized.

source

`Base.Matrix` – Method.

```
Matrix(λY::GlobalSection)
```

Put `λY` into matrix form.

This is not recommended if speed is important!

Use `apply_section` and `global_rep` instead!

source

`GeometricMachineLearning.apply_section` – Function.

```
apply_section(λY::GlobalSection{T, AT}, Y₂::AT) where {T, AT <: StiefelManifold{T}}
```

Apply `λY` to `Y₂`.

Mathematically this is the group action of the element $\lambda Y \in G$ on the element Y_2 of the homogeneous space $\mathcal{M}$.

Internally it calls `apply_section!`.

source

`GeometricMachineLearning.apply_section!` – Function.

```
apply_section!(Y::AT, λY::GlobalSection{T, AT}, Y₂::AT) where {T, AT<:StiefelManifold{T}}
```

Apply `λY` to `Y₂` and store the result in `Y`.

This is the inplace version of `apply_section`.

source

`Base.:*` – Method.

```
λY * Y
```

Apply the element `λY` onto `Y`.

Here `λY` is an element of a Lie group and `Y` is an element of a homogeneous space.

source

`GeometricMachineLearning.global_section` – Method.

```
global_section(Y::StiefelManifold)
```

Compute a matrix of size $N \times (N - n)$ whose columns are orthogonal to the columns in `Y`.

This matrix is also called $Y_\perp$ [21, 25, 40].

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: global_section
import Random

Random.seed!(123)

Y = StiefelManifold([1. 0.; 0. 1.; 0. 0.; 0. 0.])

round.(Matrix(global_section(Y)); digits = 3)

# output

4×2 Matrix{Float64}:
 0.0    -0.0
 0.0     0.0
 0.936  -0.353
 0.353   0.936
```

Further note that we convert the `QRCompactWYQ` object to a `Matrix` before we display it.

Implementation

The implementation is done with a QR decomposition (`LinearAlgebra.qr!`). Internally we do:

```
A = randn(N, N - n) # or the gpu equivalent
A = A - Y.A * (Y.A' * A)
qr!(A).Q
```

source

```
global_section(Y::GrassmannManifold)
```

Compute a matrix of size $N \times (N - n)$ whose columns are orthogonal to the columns in `Y`.

The method `global_section` for the Grassmann manifold is equivalent to that for the `StiefelManifold` (we represent the Grassmann manifold as an embedding in the Stiefel manifold).

See the documentation for `global_section(Y::StiefelManifold{T}) where T`.

source

`GeometricMachineLearning.global_section` – Method.

```
global_section(Y::StiefelManifold)
```

Compute a matrix of size $N \times (N - n)$ whose columns are orthogonal to the columns in `Y`.

This matrix is also called $Y_\perp$ [21, 25, 40].

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: global_section
import Random

Random.seed!(123)

Y = StiefelManifold([1. 0.; 0. 1.; 0. 0.; 0. 0.])

round.(Matrix(global_section(Y)); digits = 3)

# output

4×2 Matrix{Float64}:
 0.0    -0.0
 0.0     0.0
 0.936  -0.353
 0.353   0.936
```

Further note that we convert the `QRCompactWYQ` object to a `Matrix` before we display it.

Implementation

The implementation is done with a QR decomposition (`LinearAlgebra.qr!`). Internally we do:

```
A = randn(N, N - n) # or the gpu equivalent
A = A - Y.A * (Y.A' * A)
qr!(A).Q
```

source

```
global_section(Y::GrassmannManifold)
```

Compute a matrix of size $N \times (N - n)$ whose columns are orthogonal to the columns in `Y`.

The method `global_section` for the Grassmann manifold is equivalent to that for the `StiefelManifold` (we represent the Grassmann manifold as an embedding in the Stiefel manifold).

See the documentation for `global_section(Y::StiefelManifold{T}) where T`.

source

`GeometricMachineLearning.global_rep` – Function.

```
global_rep(λY::GlobalSection{T, AT}, Δ::AbstractMatrix{T}) where {T, AT<:StiefelManifold{T}}
```

Express `Δ` (an the tangent space of `Y`) as an instance of `StiefelLieAlgHorMatrix`.

This maps an element from $T_Y\mathcal{M}$ to an element of $\mathfrak{g}^{\text{hor}}$.

These two spaces are isomorphic where the isomorphism where the isomorphism is established through $\lambda(Y) \in G$ via:

$$T_Y\mathcal{M} \to \mathfrak{g}^{\text{hor}}, \Delta \mapsto \lambda(Y)^{-1}\Omega(Y, \Delta)\lambda(Y).$$

Also see `GeometricMachineLearning.Ω`.

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: _round
import Random

Random.seed!(123)

Y = rand(StiefelManifold, 6, 3)
Δ = rgrad(Y, randn(6, 3))
λY = GlobalSection(Y)

_round(global_rep(λY, Δ); digits = 3)

# output

6×6 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
  0.0     0.679   1.925   0.981  -2.058   0.4
 -0.679   0.0     0.298  -0.424   0.733  -0.919
 -1.925  -0.298   0.0    -1.815   1.409   1.085
 -0.981   0.424   1.815   0.0     0.0     0.0
  2.058  -0.733  -1.409   0.0     0.0     0.0
 -0.4     0.919  -1.085   0.0     0.0     0.0
```

Implementation

The function `global_rep` does in fact not perform the entire map $\lambda(Y)^{-1}\Omega(Y,\Delta)\lambda(Y)$ but only

$$\Delta \mapsto \mathrm{skew}(Y^T\Delta),$$

to get the small skew-symmetric matrix $A \in \mathcal{S}_{\mathrm{skew}}(n)$ and

$$\Delta \mapsto (\lambda(Y)^T_{[1:N,n:N]}\Delta)_{[1:(N-n),1:n]},$$

to get the arbitrary matrix $B \in \mathbb{R}^{(N-n)\times n}$.

source

```
global_rep(λY::GlobalSection{T, AT}, Δ::AbstractMatrix{T}) where {T, AT<:GrassmannManifold{T}}
```

Express `Δ` (an element of the tangent space of `Y`) as an instance of `GrassmannLieAlgHorMatrix`.

The method `global_rep` for `GrassmannManifold` is similar to that for `StiefelManifold`.

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: _round
import Random

Random.seed!(123)

Y = rand(GrassmannManifold, 6, 3)
Δ = rgrad(Y, randn(6, 3))
λY = GlobalSection(Y)

_round(global_rep(λY, Δ); digits = 3)

# output

6×6 GrassmannLieAlgHorMatrix{Float64, Matrix{Float64}}:
  0.0     0.0     0.0     0.981  -2.058   0.4
  0.0     0.0     0.0    -0.424   0.733  -0.919
  0.0     0.0     0.0    -1.815   1.409   1.085
 -0.981   0.424   1.815   0.0     0.0     0.0
  2.058  -0.733  -1.409   0.0     0.0     0.0
 -0.4     0.919  -1.085   0.0     0.0     0.0
```

source

Chapter Summary

In this chapter we discussed mathematical core concepts in this dissertation: aspects from general topology and analysis. We furthermore discussed Riemannian geometry and parallel transport which are crucial concepts for manifold optimization. Homogeneous spaces were introduced as an important subcategory of manifolds and we showed how to find a global tangent space representation

for them. This was done by identifying a distinct element $E \in \mathcal{M}$ and then considering the action of the Lie algebra $\mathfrak{g}$ on this element. We have presented our basic application user interface (API) for the `Julia` package `GeometricMachineLearning`. This API will be extended in Part II with a general optimizer framework.

References

[34] S. Lipschutz. *General Topology* (McGraw-Hill Book Company, New York City, New York, 1965).

[37] S. Lang. *Real and functional analysis*. Vol. 142 (Springer Science & Business Media, 2012).

[35] S. Lang. *Fundamentals of differential geometry*. Vol. 191 (Springer Science & Business Media, 2012).

[36] S. I. Richard L. Bishop. *Tensor Analysis on Manifolds* (Dover Publications, Mineola, New York, 1980).

[39] M. P. Do Carmo and J. Flaherty Francis. *Riemannian geometry*. Vol. 2 (Springer, 1992).

[40] P.-A. Absil, R. Mahony and R. Sepulchre. *Riemannian geometry of Grassmann manifolds with a view on algorithmic computation*. Acta Applicandae Mathematica **80**, 199–220 (2004).

[21] T. Bendokat, R. Zimmermann and P.-A. Absil. *A Grassmann manifold handbook: Basic geometry and computational aspects*, arXiv preprint arXiv:2011.13699 (2020).

[94] T. Frankel. *The geometry of physics: an introduction* (Cambridge university press, Cambridge, UK, 2011).

[65] B. O'neill. *Semi-Riemannian geometry with applications to relativity* (Academic press, New York City, New York, 1983).

[22] T. Bendokat and R. Zimmermann. *The real symplectic Stiefel and Grassmann manifolds: metrics, geodesics and applications*, arXiv preprint arXiv:2108.12447 (2021).

Chapter 3

Geometric Structure

A central topic of this dissertation is *structure preservation*. In this chapter we first discuss the notions of *symplecticity* and *volume preservation*. After this we introduce the concept of *structure-preserving neural networks*.

3.1 Symplectic Systems

Symplectic systems are ODEs whose vector field has a specific structure that is very restrictive. Before we introduce symplectic vector fields on manifolds we first have to define what a *symplectic structure* is:

Definition 3.1.1. A **symplectic structure** or **symplectic 2-form** Ω assigns to each $x \in \mathcal{M}$ a mapping

$$\Omega_x : T_x\mathcal{M} \times T_x\mathcal{M} \to \mathbb{R},$$

such that Ω_x is skew-symmetric and nondegenerate, and Ω is closed.

We forego the precise definition of *closedness* because it would require us to introduce differential forms [36, 42]. This property is also closely related to the *Jacobi identity* [43, Chapter 4.4]. After having defined a symplectic structure, we can introduce *Hamiltonian vector fields*[1]:

Definition 3.1.2. A **Hamiltonian vector field** at $x \in \mathcal{M}$ corresponding to the function $H : \mathcal{M} \to \mathbb{R}$ (called **the Hamiltonian**) is a vector field that has the following property:

$$\Omega(X_H, \dot{\gamma}(0)) = \frac{d}{dt}\bigg|_{t=0} H(\gamma(t)), \Omega(X_H, \dot{\gamma}(0)) = \frac{d}{dt}\bigg|_{t=0} H(\gamma(t)),$$

where γ is a C^∞ curve through x.

Of particular importance for us here are *canonical Hamiltonian systems*:

Example 3.1.3. To obtain a canonical Hamiltonian system we take $\mathcal{M} = \mathbb{R}^{2d}$ and

$$\Omega_z = \mathbb{J}_{2d}^T = \begin{pmatrix} \mathbb{O} & -\mathbb{I} \\ \mathbb{I} & \mathbb{O} \end{pmatrix}$$

[1] Also compare this to the definition of the Riemannian gradient (see Chapter 2.7).

for all z. We call $\mathbb{J}_{2d}$ the **Poisson tensor**. In this case the vector field can be written as:

$$X_H(z) = \mathbb{J}_{2d}\nabla_z H,$$

where ∇H is the Euclidean gradient.

Typically we further split the z coordinates on $\mathbb{R}^{2d}$ into (q, p) coordinates, where q and p are the first d components of z and the second d components of z respectively, i.e.

$$z = \begin{pmatrix} q \\ p \end{pmatrix}.$$

We can then reformulate a Hamiltonian vector field as two separate vector fields:

$$\begin{aligned} \dot{q} &= \frac{\partial H}{\partial p} \quad \text{and} \\ \dot{p} &= -\frac{\partial H}{\partial q}. \end{aligned}$$

Solution of Symplectic Systems

The flow (see Chapter 2.6) of a Hamiltonian ODE has very restrictive properties, the most important one of these is called *symplecticity* [3]. This property dramatically restricts the dynamically accessible states of the flow map. For a canonical Hamiltonian system symplecticity is defined as follows:

Definition 3.1.4. A map $\phi : \mathbb{R}^{2d} \to \mathbb{R}^{2d}$ is called **symplectic** on $U \subset \mathbb{R}^{2d}$ if

$$(\nabla_z \phi)^T \mathbb{J}_{2d} \nabla_z \phi = \mathbb{J}_{2d}, (\nabla_z \phi)^T \mathbb{J}_{2d} \nabla_z \phi = \mathbb{J}_{2d},$$

for all $z \in U$.

A similar definition of symplecticity holds for the general case of symplectic manifolds $(\mathcal{M}, \Omega)$ [42]. The following holds:

Theorem 3.1.5. *The flow of a Hamiltonian system is symplectic*

Proof. We proof this statement only for canonical Hamiltonian systems here. Consider the flow of a Hamiltonian ODE $\varphi^t : \mathbb{R} \to \mathbb{R}$. For this we have:

$$\begin{aligned} \frac{d}{dt}\left((\nabla\varphi^t)^T \mathbb{J} \nabla\varphi^t\right) &= (\mathbb{J}\nabla^2 H)^T \mathbb{J}\nabla\varphi^t + (\nabla\varphi^t)^T \mathbb{J}\mathbb{J}\nabla^2 H \\ &= \nabla^2 H \nabla\varphi^t - (\nabla\varphi^t)^T \nabla^2 H, \end{aligned}$$

and this expression is zero at $t = 0$. This computation holds for all points on the domain on which H is defined and φ^t is symplectic. □

The discipline of finding numerical approximations of flows φ^t such that these numerical approximations also preserve certain properties of that flow (such as symplecticity) is referred to as *structure-preserving numerical integration* or *geometric numerical integration* [3]. The `Julia` library `GeometricIntegrators` [44] offers a wide array of such geometric numerical integrators for a broad class of systems (not just canonical Hamiltonian systems).

In addition to being symplectic, the flow of a Hamiltonian vector field is also energy-preserving:

Theorem 3.1.6. *The flow of a Hamiltonian vector field preserves the Hamiltonian H, i.e.*

$$H(\varphi^t(z_0)) = H(z_0),$$

for all $t \in [0, T]$.

Proof. We differentiate $H(\varphi^t(z_0))$ with respect to t:

$$\frac{d}{dt}H(\varphi^t(z_0)) = (\nabla_{\varphi^t(z_0)}H)^T \frac{d}{dt}\varphi^t(z_0) = (\nabla_{\varphi^t(z_0)}H)^T \mathbb{J} \nabla_{\varphi^t(z_0)}H = \mathbb{O},$$

because $\mathbb{J}$ is skew-symmetric. □

It is important to note that symplecticity is a very strong property[2] that may not be achievable in some practical applications. If preservation of symplecticity is not achievable, it may however still be advantageous to consider weaker properties such as volume preservation (see Chapter 3.2).

Library Functions

`GeometricMachineLearning.PoissonTensor` – Type.

```
PoissonTensor(n2)
```

Returns a (canonical) Poisson tensor of size $2n \times 2n$:

$$\mathbb{J}_{2n} = \begin{pmatrix} \mathbb{O} & \mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{O} \end{pmatrix}$$

Arguments

It can also be called with a `backend` and a `type`:

[2] Symplecticity imposes in general greater restrictions on the flow map than e.g. conservation of energy. The famous *Ge-Marsden theorem* [45] says that one cannot achieve preservation of energy and preservation of symplecticity at the same time, unless the numerical method is the exact integral of the Hamiltonian ODE. In practice we hence almost always choose a symplectic over an energy-preserving scheme.

```
using GeometricMachineLearning

backend = CPU()
T = Float16

PoissonTensor(backend, 4, T)

# output

4×4 PoissonTensor{Float16, Matrix{Float16}}:
  0.0   0.0  1.0  0.0
  0.0   0.0  0.0  1.0
 -1.0   0.0  0.0  0.0
  0.0  -1.0  0.0  0.0
```

source

GeometricMachineLearning.QPT – Type.

```
QPT
```

The type for data in (q, p) coordinates. It encompasses various array types.

Examples

```
using GeometricMachineLearning: QPT

# allocate two vectors
data1 = (q = rand(5), p = rand(5))

# allocate two matrices
data2 = (q = rand(5, 4), p = rand(5, 4))

# allocate two tensors
data3 = (q = rand(5, 4, 2), p = rand(5, 4, 2))

(typeof(data1) <: QPT, typeof(data2) <: QPT, typeof(data3) <: QPT)

# output

(true, true, true)
```

We can also do:

```
using GeometricMachineLearning: QPT, PoissonTensor

𝕁 = PoissonTensor(4)
qp = (q = [1, 2], p = [3, 4])

𝕁 * qp

# output
```

```
(q = [3, 4], p = [-1, -2])
```

source

GeometricMachineLearning.QPTOAT – Type.

```
QPTOAT
```

A union of two types:

```
const QPTOAT = Union{QPT, AbstractArray}
```

This could be data in $(q, p) \in \mathbb{R}^{2d}$ form or come from an arbitrary vector space.

source

3.2 Divergence-Free Vector Fields

The quality of being *divergence-free* greatly restricts the number of possible vector fields and also the dynamically accessible states of the flow map. It is however a weaker property then being Hamiltonian (see Chapter 3.1). We first define what it means to be divergence-free:

Definition 3.2.1. A vector field $X : \mathcal{M} \to T\mathcal{M}$ defined on a d-dimensional Riemannian manifold $(\mathcal{M}, g)$ is called **divergence-free** if $\forall z \in \mathcal{M}$ we have:

$$\mathrm{div}(X) = \sum_{i=1}^{d} \frac{1}{\rho} \frac{\partial \left(\frac{\rho}{\sqrt{g_{ii}}} \hat{X}^i \right)}{\partial z^i} = \sum_{i=1}^{d} \frac{1}{\sqrt{\det g}} \frac{\partial \left(\sqrt{\frac{\det g}{g_{ii}}} \hat{X}^i \right)}{\partial z^i} = 0,$$

for some parametrization of a neighborhood around z.

Remark 3.2.2. If we do not deal with a general Riemannian manifold but simply with a vector space, the divergence for $X : \mathbb{R}^d \to \mathbb{R}^d$ is usually written as

$$\mathrm{div}(X) = \nabla \cdot X = \sum_{i=1}^{d} \frac{\partial X_i}{\partial z_i},$$

where z are global coordinates.

We further define what it means to be volume-preserving:

Definition 3.2.3. We call a map $\phi : \mathcal{M} \to \mathcal{M}$ **volume-preserving** if for all volume elements $V \subset \mathcal{M}$ we have that $\mathrm{vol}(V) = \mathrm{vol}(\phi(V))$, where vol is a measure of a volume in a Riemannian manifold.

Remark 3.2.4. If we deal with vector spaces instead of more general manifolds the property of being **volume-preserving** in some domain $\mathcal{D} \subset \mathbb{R}^d$ can be expressed as

$$\det \nabla_z \varphi^t = 1 \quad \forall z \in \mathcal{D}, t \in [0, T].$$

So the columns of the Jacobian of φ^t span a volume element of size 1 for each z and each t.

We can proof the theorem:

Theorem 3.2.5. *The flow of a divergence-free vector field is volume-preserving.*

Here we only proof this statement for the case of a vector space. A proof of the more general statement can be found in standard textbooks on differential geometry[1], e.g. [35, 36].

Proof. We refer to the flow of X by $\varphi^t : \mathbb{R}^d \to \mathbb{R}^d$ and have the following property:

$$\frac{d}{dt} \nabla \varphi^t(z) = \nabla X(\varphi^t(z)) \nabla \varphi^t(z).$$

Note that we used the convention $[\nabla X(z)]_{ij} = \partial / \partial z_j X_i$ here. This expression for $d/dt \nabla \varphi^t(x)$ further implies:

$$\mathrm{Tr}\left((\nabla \varphi^t(z))^{-1} \frac{d}{dt} \nabla \varphi^t(z) \right) = \mathrm{Tr}\left((\nabla \varphi^t(z))^{-1} \nabla X(\varphi^t(z)) \nabla \varphi^t(z) \right) = \mathrm{Tr}(\nabla X(\varphi^t(z))) = 0,$$

where we have used $\mathrm{Tr}(ABC) = \mathrm{Tr}(BCA)$ in the second equality, and $\mathrm{Tr}(\nabla X) = \sum_{i=1}^d \partial X_i / \partial z_i = \mathrm{div}(X)$ and the divergence-freeness of X in the third equality. We further have

$$\mathrm{Tr}(A^{-1} \dot{A}) = \frac{\frac{d}{dt} \det(A)}{\det(A)},$$

which can be derived from the classical result $\det(\exp(A)) = \exp(\mathrm{Tr}(A))$. Hence we have

$$\frac{d}{dt} \det(\nabla \varphi^t(z)) = 0,$$

and the result is proved. □

It is a classical result that *all Hamiltonian vector fields are divergence-free*, so volume-preservation is weaker than preservation of symplecticity [42].

[1] Together with a precise definition of Riemannian integration and the volume form introduced above.

3.3 Structure-Preserving Neural Networks

What we means by a *structure-preserving neural network* or a *geometric neural network* is a modification of a standard neural networks such that it satisfies certain properties like symplecticity (see Chapter 3.1) or volume preservation (see Chapter 3.2). We first define standard neural networks:

Definition 3.3.1. A **neural network architecture** is a parameter-dependent realization of a function:

$$\text{architecture} : \mathbb{P} \to \mathcal{C}(\mathcal{D}, \mathcal{M}), \Theta \mapsto \mathcal{NN}_\Theta,$$

where Θ are the *parameters of the neural network* (we call $\mathbb{P}$ the parameter space). $\mathbb{P}$, the domain space $\mathcal{D}$ and the target space $\mathcal{M}$ of the neural network may be spaces with arbitrary structure in general (i.e. need not be vector spaces).

In this text the spaces $\mathcal{D}$ and $\mathcal{M}$ are vector spaces in most cases[1]. The parameter space $\mathbb{P}$ is however build from manifolds in many cases (see Chapter 5.1). Weights have to be put on manifolds to realize certain architectures that would otherwise not be possible (see Chapter 8.2) and can make training more efficient in other cases (see Chapter 11.1).

It is a classical result [8] that one-layer feedforward neural networks[2] are *universal approximators*:

Theorem 3.3.2. *Neural networks are dense in the space of continuous functions $\mathcal{C}(U, \mathbb{R}^m)$ in the* compact-open topology*, i.e. for every compact subset $K \subset U$, real number $\varepsilon > 0$ and function $f \in \mathcal{C}(U, \mathbb{R}^m)$ we can find an integer N as well as weights*

$$\Theta = (A, b, C) \in \mathbb{R}^{N \times n} \times \mathbb{R}^N \times \mathbb{R}^{m \times N} =: \mathbb{P}$$

such that

$$\sup_{x \in K} ||f(x) - C\sigma(Ax + b)|| < \varepsilon,$$

i.e. neural networks can approximate f arbitrarily well on any compact set K.

The universal approximation theorem has also been generalized to other neural network architectures [46–48].

A *structure-preserving* or *geometric* neural networks is a neural network that has additional properties:

Definition 3.3.3. A **structure-preserving neural network architecture** is a parameter-dependent realization of a function:

[1] One exception is Grassmann learning (see Chapter 12.1) where we learn a vector space.

[2] We obtain one-layer feedforward neural networks by identifying $\mathcal{P} = \mathbb{R}^{N \times n} \times \mathbb{R}^N \times \mathbb{R}^{m \times N} \ni (A, b, C) =: \Theta$ and $\mathcal{NN}_\Theta(x) = C\sigma(Ax + b)$ for some scalar function $\sigma : \mathbb{R} \to \mathbb{R}$ that is non-polynomial.

$$\text{sp} \cdot \text{architecture} : \mathbb{P} \to \mathcal{C}(\mathcal{D}, \mathcal{M}), \Theta \mapsto \mathcal{NN}_\Theta,$$

such that $\mathcal{NN}_\Theta$ preserves some structure.

Example 3.3.4. We say that a neural network is **symplectic** if $\mathcal{NN}_\Theta : \mathbb{R}^n \to \mathbb{R}^m$ (with $m \geq n$) preserves $\mathbb{J}$, i.e.

$$(\nabla_z \mathcal{NN}_\Theta)^T \mathbb{J}_{2m} (\nabla_z \mathcal{NN}_\Theta) = \mathbb{J}_{2n},$$

where z are coordinates on $\mathbb{R}^n$.

If we have $m = n$ then we can use SympNets (see Chapter 8.5) to realize such architectures; SympNets furthermore are universal approximators for the set of canonical symplectic maps[3] [10]. If $m \neq n$ we can use symplectic autoencoders (see Chapter 8.2) to realize such an architecture. A different class of neural networks are *volume-preserving neural networks*:

Example 3.3.5. We say that a neural network is **volume-preserving** if $\mathcal{NN}_\Theta : \mathbb{R}^n \to \mathbb{R}^n$ is such that:

$$\det(\nabla_z \mathcal{NN}_\Theta) = 1,$$

where z are coordinates on $\mathbb{R}^n$.

Note that here we keep the dimension constant. Volume-preserving neural networks can be built on the basis of feedforward neural networks (see Chapter 8.6) or transformers (see Chapter 8.8).

Chapter Summary

In this chapter we introduced *symplecticity* and *volume presentation* (or divergence-freeness for the corresponding vector field) as examples of geometric structure. *Symplecticity* is a property of the flow of Hamiltonian vector fields that dramatically restricts the accessible states of freedom. In many applications, neural networks, similar to classical numerical methods, aim at modeling the flow of a differential equation. Because symplecticity is a very restrictive property that we know the flow of the differential equation has, it is advantageous to also imbue the neural network with this property. We hence defined *structure-preserving neural networks* as the ones that preserve symplecticity (or other structure) in this chapter.

References

[42] V. I. Arnold. *Mathematical methods of classical mechanics*. Vol. 60 of *Graduate Texts in Mathematics* (Springer Verlag, Berlin, 1978).

[3] Other neural network architectures that were developed with the same aim are *Hamiltonian neural networks* [49], *Hénon nets* [50] and *generalized Hamiltonian neural networks* [51]. [52] gives an overview over *structure preserving neural networks*.

[3] E. Hairer, C. Lubich and G. Wanner. *Geometric Numerical integration: structure-preserving algorithms for ordinary differential equations* (Springer, Heidelberg, 2006).

[84] B. Leimkuhler and S. Reich. *Simulating hamiltonian dynamics*. No. 14 (Cambridge university press, 2004).

[49] S. Greydanus, M. Dzamba and J. Yosinski. *Hamiltonian neural networks*. Advances in neural information processing systems **32** (2019).

[50] J. W. Burby, Q. Tang and R. Maulik. *Fast neural Poincaré maps for toroidal magnetic fields*. Plasma Physics and Controlled Fusion **63**, 024001 (2020).

[51] P. Horn, V. S. Ulibarrena, B. Koren and S. P. Zwart. *A generalized framework of neural networks for Hamiltonian systems*. Journal of Computational Physics **521**, 113536 (2025).

[52] E. Celledoni, M. J. Ehrhardt, C. Etmann, R. I. McLachlan, B. Owren, C.-B. Schonlieb and F. Sherry. *Structure-preserving deep learning*. European journal of applied mathematics **32**, 888–936 (2021).

Chapter 4

Reduced Order Modeling

In this chapter we discuss *data-driven reduced order modeling*. This serves as a motivation for designing structure-preserving neural networks. We here first discuss the general workflow of reduced order modeling and explain the split into an *offline phase* and an *online phase*. Next we discuss *proper orthogonal decomposition* and *autoencoders* as examples of machine learning techniques that can be used for the offline phase of reduced order modeling. The chapter concludes with an explanation of how to adjust a reduced order model for the Hamiltonian setting in order to make it structure-preserving.

4.1 Basic Concepts of Reduced Order Modeling

Reduced order modeling is a data-driven technique that exploits the structure of parametric partial differential equations (PPDEs) to make repeated simulations of this PPDE much cheaper.

For this consider a PPDE written in the form: $F(z(\mu);\mu) = 0$ where $z(\mu)$ evolves on an infinite-dimensional Hilbert space V.

In modeling any PDE we have to choose a discretization (particle discretization, finite element method, ...) of V which will be denoted by $V_h \simeq \mathbb{R}^N$. The space V_h is not infinite-dimensional but its dimension N is still very large. Solving a discretized PDE in this space is typically very expensive. In reduced order modeling we utilize the fact that slightly different choices of parameters μ will give qualitatively similar solutions. We can therefore perform a few simulations in the full space V_h and then make successive simulations cheaper by *learning* from the past simulations:

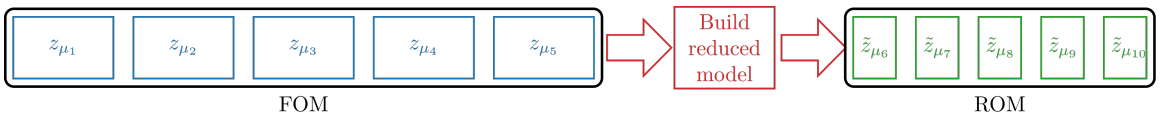


Figure 4.1: Schematic representation of a reduced order modeling framework. The width of the individual blocks represent how long it takes to perform a simulation.

In the figure above we refer to the discretized PDE as the *full order model* (FOM) and to the cheaper representation (that we construct in a data-driven manner) as the *reduced order model* (ROM). We now introduce the *solution manifold*, which is a crucial concept in reduced order modeling.

The Solution Manifold

To any PPDE and a certain parameter set $\mathbb{P}$ we associate a *solution manifold*:

$$\mathcal{M} = \{z(\mu) : F(z(\mu); \mu) = 0, \mu \in \mathbb{P}\}.$$

A motivation for reduced order modeling is that even though the space V_h is of very high-dimension, the solution manifold will typically be a very small space. The image here shows a two-dimensional solution manifold[1] embedded in $V_h \equiv \mathbb{R}^3$.

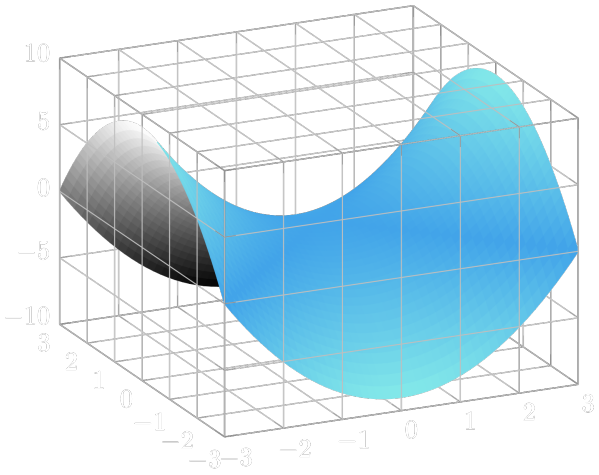


Figure 4.2: A representation of a two-dimensional solution manifold embedded in three-dimensional Euclidean space.

As an actual example of a solution manifold consider the one-dimensional wave equation [53]:

$$\partial_{tt}^2 q(t, \omega; \mu) = \mu^2 \partial_{\omega\omega}^2 q(t, \omega; \mu) \text{ on } I \times \Omega,$$

where $I = (0, 1)$ and $\Omega = (-1/2, 1/2)$. As initial condition for the first derivative we have $\partial_t q(0, \omega; \mu) = -\mu \partial_\omega q_0(\xi; \mu)$ and furthermore $q(t, \omega; \mu) = 0$ on the boundary (i.e. $\omega \in \{-1/2, 1/2\}$).

The solution manifold is a two-dimensional submanifold of an infinite-dimensional function space:

$$\mathcal{M} = \{(t, \omega) \mapsto q(t, \omega; \mu) = q_0(\omega - \mu t; \mu) : \mu \in \mathbb{P} \subset \mathbb{R}\}.$$

We can plot some of the *points* on $\mathcal{M}$ (each curve at a specific time corresponds to one point on the solution manifold):

[1] The systems we deal with usually have much greater dimension of course. The dimension of V_h will be in the thousands and the dimension of the solution manifold will be a few orders of magnitudes smaller. Because this cannot be easily visualized, we resort to showing a two-dimensional manifold in a three-dimensional space here.

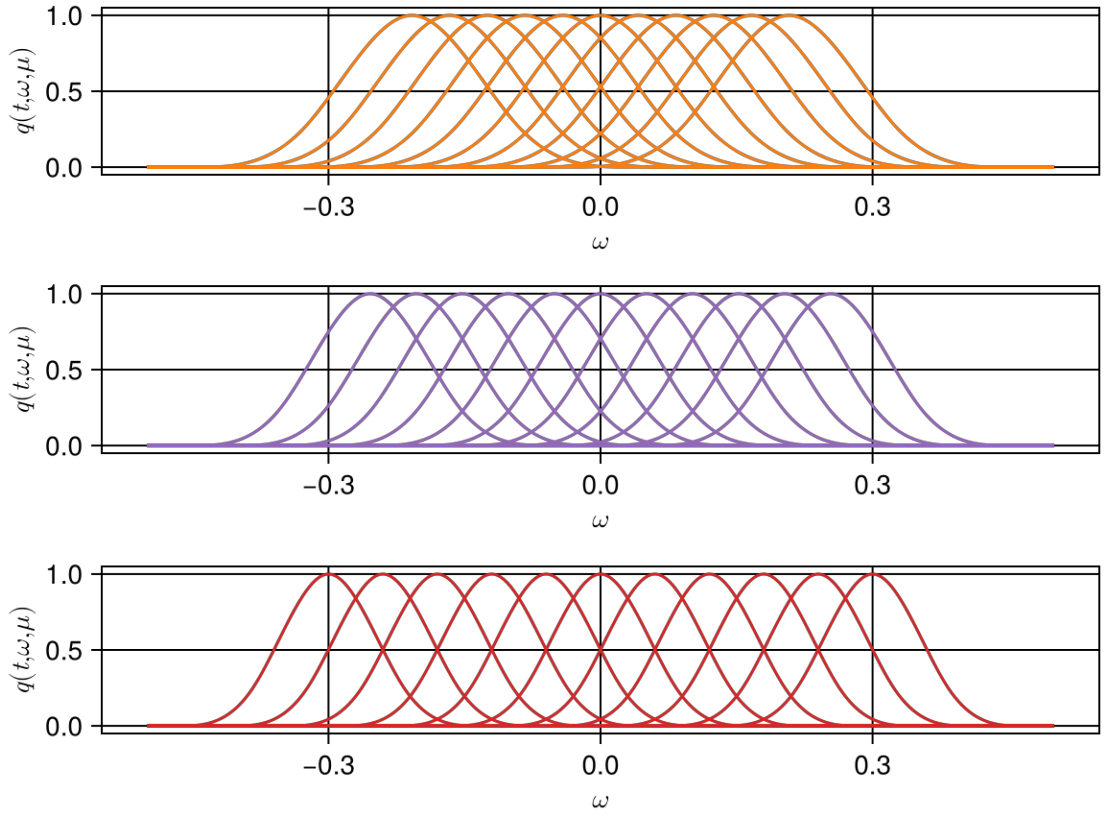


Figure 4.3: Solution of the one-dimensional wave equation for three different parameters. These solutions evolve on a two-dimensional solution manifold.

Here we plotted the curves for the time steps (0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0) and the parameter values (0.416, 0.508, 0.6). We see that, depending on the parameter value μ, the wave travels at different speeds. In reduced order modeling we try to find an approximation to the solution manifolds, i.e. model the evolution of the curve in a cheap way for different parameter values μ. Neural networks offer a way of doing so efficiently!

General Workflow

In reduced order modeling we aim to construct an approximation to the solution manifold and that is ideally of a dimension not much greater than that of the solution manifold and then (approximately) solve the so-called *reduced equations* in the small space. Constructing this approximation to the solution manifold can be divided into three steps[2]:

1. Discretize the PDE, i.e. find $V \to V_h$.
2. Solve the discretized PDE on V_h for a set of parameter instances $\mu \in \mathbb{P}$.
3. Build a reduced basis with the data obtained from having solved the discretized PDE. This step consists of finding two mappings: the *reduction* $\mathcal{P}$ and the *reconstruction* $\mathcal{R}$.

[2]Approximating the solution manifold is referred to as the *offline phase* of reduced order modeling.

The third step can be done with various machine learning (ML) techniques. Traditionally the most popular of these has been *proper orthogonal decomposition* (POD), but in recent years *autoencoders* have become a widely-used alternative [54, 55].

After having obtained $\mathcal{P}$ and $\mathcal{R}$ we still need to solve the *reduced system*. Solving the reduced system is typically referred to as the *online phase* in reduced order modeling. This is sketched below:

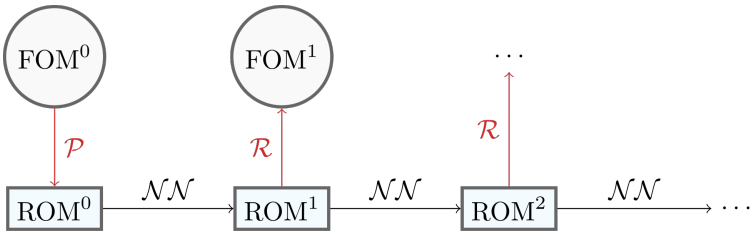


Figure 4.4: The offline phase in reduced order modeling consists of finding the reduction and the reconstruction. In the online phase we solve the reduced model.

In this figure the online phase consists of applying the mapping $\mathcal{NN}$ in the low-dimensional space in order to predict the next time step; this can either be done with a standard integrator [44] or, as is indicated here, with a neural network (see Chapter 8.3). Crucially this step can be made very cheap when compared to the full-order model[3]. In the following we discuss how an equation for the reduced model can be found classically, without relying on a neural network for the online phase.

Obtaining the Reduced System via Galerkin Projection

Galerkin projection [56] offers a way of constructing an ODE on the reduced space once the reconstruction $\mathcal{R}$ has been found.

Definition 4.1.1. Given a full-order model described by a differential equation $\hat{F}(\cdot;\mu) : V \to V$, where V may be an infinite-dimensional Hilbert space (PDE case) or a finite-dimensional vector space $\mathbb{R}^N$ (ODE case), and a reconstruction $\mathcal{R} : \mathbb{R}^n \to V$, we can find an equation on the reduced space $\mathbb{R}^n$. For this we first take as possible solutions for the equation

$$\hat{F}(\hat{u}(t);\mu) - \hat{u}'(t) =: F(\hat{u}(t);\mu) = 0$$

the ones that are the *result of a reconstruction*:

$$\hat{F}(\mathcal{R}(u(t));\mu) - d\mathcal{R}u'(t) = 0,$$

where $u : [0,T] \to \mathbb{R}^n$ is an orbit on the reduced space and $d\mathcal{R}$ is the differential of the reconstruction; this is $\nabla\mathcal{R}$ if V is finite-dimensional. Typically we test this expression with a set of basis functions or vectors $\{\tilde{\psi}_1, \ldots, \tilde{\psi}_n\}$ and hence obtain n scalar equations:

$$\langle \hat{F}(\mathcal{R}(u(t));\mu) - d\mathcal{R}u'(t), \psi_i \rangle_V \text{ for } 1 \leq i \leq n.$$

[3]Solving the reduced system is typically faster by a factor of 10^2 or more (see Chapter 9.1).

Such a procedure to obtain a reduced equation is known as **Galerkin projection**.

We give specific examples of reduced systems obtained with a Galerkin projection when introducing proper orthogonal decomposition (see Chapter 4.2), autoencoders (see Chapter 4.3), proper symplectic decomposition and symplectic auteoncoders (see Chapter 4.5).

Kolmogorov n-width

The Kolmogorov n-width [57] measures how well some set $\mathcal{M}$ (typically the solution manifold) can be approximated with a linear subspace:

$$d_n(\mathcal{M}) := \inf_{V_n \subset V; \dim V_n = n} \sup(u \in \mathcal{M}) \inf_{v_n \in V_n} ||u - v_n||_V,$$

with $\mathcal{M} \subset V$ and V is a (typically infinite-dimensional) Banach space. For advection-dominated problems (among others) the *decay of the Kolmogorov n-width is very slow*, i.e. one has to pick n very high in order to obtain useful approximations (see [58] and [53]). As proper orthogonal decomposition (see Chapter 4.2) is a linear approximation to the solution manifold, this does not work very well if the decay of the Kolmogorov n-width is slow.

In order to overcome this, techniques based on neural networks [55] and optimal transport [53] have been used.

4.2 Proper Orthogonal Decomposition

Proper orthogonal decomposition (POD, [59]) is perhaps the most widely-used technique for data-driven reduced order modeling (see Chapter 4.1). POD approximates the reduction and the reconstruction through linear maps. Assume that the big discretized space has dimension N and we try to model the solution manifold with an n-dimensional subspace. POD then models the reduction $\mathcal{P} : \mathbb{R}^N \to \mathbb{R}^n$ through a matrix $\in \mathbb{R}^{n \times N}$ and the reconstruction $\mathcal{R} : \mathbb{R}^n \to \mathbb{R}^N$ through a matrix $\in \mathbb{R}^{N \times n}$. If we are given a snapshot matrix (see Chapter A.1) finding $\mathcal{P}$ and $\mathcal{R}$ amounts to a simple application of *singular value decomposition* (SVD).

Theorem 4.2.1. *Given a snapshot matrix $M = [u_1, \dots, u_{\texttt{nts}}] \in \mathbb{R}^{N \times \texttt{nts}}$, where* `nts` *is the* number of time steps*, the ideal linear subspace that can best approximate the data stored in M are the first n columns of the V matrix in an SVD: $M = VDU^T$. The problem of finding this subspace can either be phrased as a maximization problem:*

$$\max_{\psi_1, \dots, \psi_n \in \mathbb{R}^N} \sum_{i=1}^{n} \sum_{j=1}^{\texttt{nts}} |\langle u_j, \psi_i \rangle_{\mathbb{R}^N}|^2 \ s.t. \ \langle \psi_i, \psi_j \rangle = \delta_{ij} \ for \ 1 \leq i, \ j \leq n,$$

or as a minimization problem:

$$\min_{\psi_1, \dots, \psi_n \in \mathbb{R}^N} \sum_{j=1}^{\texttt{nts}} |u_j - \sum_{i=1}^{n} \psi_i \langle u_j, u_i \rangle|^2 \ s.t. \ \langle \psi_i, \psi_j \rangle = \delta_{ij} \ for \ 1 \leq i, \ j \leq n.$$

In both these cases we have

$$[\psi_1, \psi_2, \dots, \psi_n] = V[1:\mathtt{N}, 1:\mathtt{n}],$$

where V is obtained via an SVD of M.

A proof of the statement above can be found in e.g. [60]. We can obtain the reduced equations via Galerkin projection (see Chapter 4.1):

Theorem 4.2.2. *Consider a full-order model on $\mathbb{R}^N$ described by the vector field $\hat{u}'(t) = X(\hat{u}(t))$. For a POD basis the reduced vector field, obtained via Galerkin projection, is:*

$$u'(t) = V^T X(Vu(t)),$$

where we used $\{\tilde{\psi}_i = Ve_i\}_{i=1,\dots,n}$ as test functions. $e_i \in \mathbb{R}^n$ is the vector that is zero everywhere except for the i-th entry, where it is one.

Proof. If we take as test function $\tilde{\psi}_i = Ve_i$, then we get:

$$e_i^T V^T X(Vu(t)) \overset{!}{=} e_i^T V^T V u'(t) = e_i^T u'(t),$$

and since this must be true for every $i = 1, \dots, n$ we obtain the desired expression for the reduced vector field. □

In recent years another approach to model $\mathcal{P}$ and $\mathcal{R}$ has become popular, namely to use neural networks to do so.

4.3 Autoencoders

Autoencoders are a popular tool in machine learning to perform *data compression* [1]. The idea is always to find a low-dimensional representation of high-dimensional data. This is also referred to as *learning a feature space*. This idea straightforwardly lends itself towards an application in reduced order modeling. In this setting we *learn* two mappings that are modeled with neural networks:

Definition 4.3.1. An **autoencoder** is a tuple of two mappings $(\mathcal{P}, \mathcal{R})$ called the **reduction** and the **reconstruction**:

1. The reduction $\mathcal{P} : \mathbb{R}^N \to \mathbb{R}^n$ is modeled with a neural network that maps high-dimensional data to a low-dimensional feature space. This network is also referred to as the **encoder** and we routinely denote it by $\Psi^{\text{enc}}_{\theta_1}$ to stress the parameter-dependence on θ_1.
2. The reconstruction $\mathcal{R} : \mathbb{R}^n \to \mathbb{R}^N$ is modeled with a neural network that maps inputs from the low-dimensional feature space to the high-dimensional space in which the original data were collected. This network is also referred to as the **decoder** and we routinely denote it by $\Psi^{\text{dec}}_{\theta_2}$ to stress the parameter-dependence on θ_2.

During training we optimize the autoencoder for minimizing the *projection error*.

Unlike in the POD case we have to resort to using neural network optimizers (see Chapter 5.1) in order to adapt the neural network to the data at hand as opposed to simply using SVD. The use of autoencoders instead of POD is extremely advantageous in the case when we deal with problems that exhibit a slowly-decaying Kolmogorov n-width. During training we minimize the projection error (see Chapter 4.4).

Remark 4.3.2. Note that POD can be seen as a special case of an autoencoder where the encoder and the decoder both consist of only one matrix. If we restrict this matrix to be orthonormal, i.e. optimize on the Stiefel manifold, then the best solution we can obtain is equivalent to applying SVD and finding the POD basis.

The Reduced Equations for the Autoencoder

Equivalently to the POD case, we get a reduced vector field when we reduce with the autoencoder:

Theorem 4.3.3. *Consider a full-order model on $\mathbb{R}^N$ described by the vector field $\hat{u}'(t) = X(\hat{u}(t))$. If we reduce with an autoencoder $(\Psi^{\text{enc}}, \Psi^{\text{dec}})$ we obtain a reduced vector field via Galerkin projection:*

$$u'(t) = (\nabla\Psi^{\text{dec}})^T X(\Psi^{\text{dec}}(u(t))),$$

where we used $\{\tilde{\psi}_i = \Psi^{\text{dec}}(e_i)\}_{i=1,\dots,n}$ as test functions. $e_i \in \mathbb{R}^n$ is the vector that is zero everywhere except for the i-th entry, where it is one.

Proof. If we take as test function $\tilde{\psi}_i = (\nabla\Psi^{\text{dec}})e_i$ we get:

$$e_i^T(\Psi^{\text{dec}})^+ X(\Psi^{\text{dec}}(u(t))) \overset{!}{=} e_i^T(\Psi^{\text{dec}})^+(\nabla\Psi^{\text{dec}})u'(t) = e_i^T u'(t),$$

and since this must be true for every $i = 1, \dots, n$ we obtain the desired expression for the reduced vector field. □

Both POD and standard autoencoders suffer from the problem that they completely neglect the structure of the differential equation and the data they are applied to. This can have grave consequences [11, 61, 62]. Hamiltonian model order reduction (see Chapter 4.5) can improve the approximation significantly in these situations.

Library Functions

`GeometricMachineLearning.AutoEncoder` – Type.

```
AutoEncoder <: Architecture
```

The abstract `AutoEncoder` type.

An autoencoder [1] is a neural network consisting of an encoder Ψ^e and a decoder Ψ^d. In the simplest case they are trained on some data set $\mathcal{D}$ to reduce the following error:

$$||\Psi^d \circ \Psi^e(\mathcal{D}) - \mathcal{D}||,$$

which we call the *reconstruction error*, *projection error* or *autoencoder error* (see the docs for `AutoEncoderLoss`) and $||\cdot||$ is some norm.

Implementation

`AutoEncoder` is an abstract type. If a custom `<:AutoEncoder` architecture is implemented it should have the fields `full_dim`, `reduced_dim`, `n_encoder_blocks` and `n_decoder_blocks`.

`n_encoder_blocks` and `n_decoder_blocks` indicate how often the dimension is changed in the encoder (respectively the decoder).

Further the routines `encoder` and `decoder` should be extended.

source

`GeometricMachineLearning.Encoder` – Type.

```
Encoder <: Architecture
```

This is the abstract `Encoder` type.

Most often this should not be called directly, but rather through the `encoder` function.

Implementation

If a custom `<:Encoder` architecture is implemented it should have the fields `full_dim`, `reduced_dim` and `n_encoder_blocks`.

source

`GeometricMachineLearning.Decoder` – Type.

```
Decoder <: Architecture
```

This is the abstract `Decoder` type.

Most often this should not be called directly, but rather through the `decoder` function.

Implementation

If a custom `<:Decoder` architecture is implemented it should have the fields `full_dim`, `reduced_dim` and `n_decoder_blocks`.

source

`GeometricMachineLearning.UnknownEncoder` – Type.

```
UnknownEncoder(full_dim, reduced_dim, n_encoder_blocks)
```

Make an instance of `UnknownEncoder`.

This should be used if one wants to use an `Encoder` that does not have any specific structure.

Examples

We show how to make an encoder from a custom architecture:

```
using GeometricMachineLearning
using GeometricMachineLearning: UnknownEncoder, params

model = Chain(Dense(5, 3, tanh; use_bias = false), Dense(3, 2, identity; use_bias = false))
nn = NeuralNetwork(UnknownEncoder(5, 2, 2), model, params(NeuralNetwork(model)), CPU())

typeof(nn) <: NeuralNetwork{<:GeometricMachineLearning.Encoder}

# output

true
```

source

`GeometricMachineLearning.UnknownDecoder` – Type.

```
UnknownDecoder(full_dim, reduced_dim, n_encoder_blocks)
```

Make an instance of `UnknownDecoder`.

This should be used if one wants to use an `Decoder` that does not have any specific structure.

An example of using this can be constructed analogously to `UnknownDecoder`.

source

`GeometricMachineLearning.encoder` – Function.

```
encoder(nn::NeuralNetwork{<:AutoEncoder})
```

Obtain the *encoder* from an `AutoEncoder` neural network.

source

```
encoder(nn)
```

Make a neural network of type `Encoder` out of an arbitrary neural network.

Implementation

Internally this allocates a new nerual network of type `UnknownEncoder` and takes the parameters and the backend from `nn`.

source

`GeometricMachineLearning.decoder` – Function.

```
decoder(nn::NeuralNetwork{<:AutoEncoder})
```

Obtain the *decoder* from an `AutoEncoder` neural network.

source

```
decoder(nn)
```

Make a neural network of type `Decoder` out of an arbitrary neural network.

Implementation

Internally this allocates a new nerual network of type `UnknownDecoder` and takes the parameters and the backend from `nn`.

source

4.4 Losses and Errors

In general we distinguish between *losses* that are used during training of a neural network and *errors* that arise in the context of reduced order modeling.

Different Neural Network Losses

`GeometricMachineLearning` has a number of loss functions implemented that can be called *standard losses*. Those are the `FeedForwardLoss` (defined in `AbstractNeuralNetworks`), the `TransformerLoss`, the `AutoEncoderLoss` and the `ReducedLoss`. How to implement custom losses is shown in a tutorial (see Chapter C.1).

A Note on Physics-Informed Neural Networks

A popular trend in recent years has been considering known physical properties of the differential equation, or the entire differential equation, through the loss function [7]. This is one way of considering physical properties, and `GeometricMachineLearning` allows for a flexible implementation of custom losses (see Chapter C.1), but this is nonetheless discouraged. In general a neural networks consists of *three ingredients*:

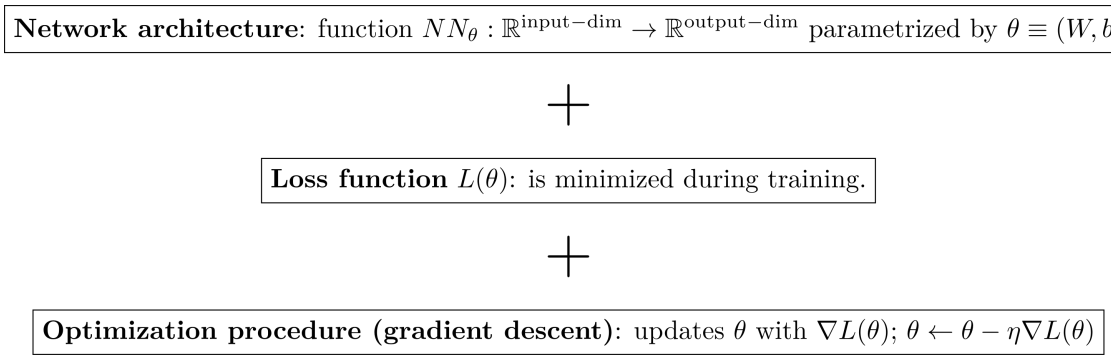


Figure 4.5: The three ingredients that go into neural network-based machine learning.

Instead of considering certain properties through the loss function, we instead do so by enforcing them strongly through the network architecture and the optimizer; the latter pertains to manifold optimization (see Chapter 5.1). The advantages of this approach are the strong enforcement of properties that we know our network should have and much easier training because we do not have to tune hyperparameters.

Projection and Reduction Errors of Reduced Models

Two errors that are of very big importance in reduced order modeling are the *projection* and the *reduction error*. During training one typically aims at minimizing the projection error, but for the actual application of the model the reduction error is often more important.

Projection Error

The projection error computes how well a reduced basis, represented by the reduction $\mathcal{P}$ and the reconstruction $\mathcal{R}$, can represent the data with which it is build. In mathematical terms:

$$e_{\text{proj}}(\mu) := \frac{||\mathcal{R} \circ \mathcal{P}(M) - M||}{||M||},$$

where $||\cdot||$ is the Frobenius norm (one could also optimize for different norms). The corresponding function in `GeometricMachineLearning` is `projection_error`. The projection error is equivalent to `AutoEncoderLoss` and is used for training under that name.

Reduction Error

The reduction error measures how far the reduced system diverges from the full-order system during integration (online stage). In mathematical terms (and for a single initial condition):

$$e_{\text{red}}(\mu) := \sqrt{\frac{\sum_{t=0}^{K} ||\mathbf{x}^{(t)}(\mu) - \mathcal{R}(\mathbf{x}_r^{(t)}(\mu))||^2}{\sum_{t=0}^{K} ||\mathbf{x}^{(t)}(\mu)||^2}},$$

where $\mathbf{x}^{(t)}$ is the solution of the FOM at point t and $\mathbf{x}_r^{(t)}$ is the solution of the ROM (in the reduced basis) at point t. The reduction error, as opposed to the projection error, not only measures how well the solution manifold is represented by the reduced basis, but also measures how well the FOM dynamics are approximated by the ROM dynamics (via the induced vector field on the reduced basis). The corresponding function in `GeometricMachineLearning` is `reduction_error`. The reduction error is, in contract to the projection error, typically not used during training (even though some authors are using a similar error to do so [55]).

Library Functions

`GeometricMachineLearning.TransformerLoss` – Type.

```
TransformerLoss(seq_length, prediction_window)
```

Make an instance of the transformer loss.

This should be used together with a neural network of type `TransformerIntegrator`.

Example

`TransformerLoss` applies a neural network to an input and compares it to the `output` via an L_2 norm:

```
using GeometricMachineLearning
using LinearAlgebra: norm
import Random

const d = 2
const seq_length = 3
const prediction_window = 2

Random.seed!(123)
arch = StandardTransformerIntegrator(d)
nn = NeuralNetwork(arch)

input_mat =  [1. 2. 3.; 4. 5. 6.]
output_mat = [1. 2.; 3. 4.]
loss = TransformerLoss(seq_length, prediction_window)

# start of prediction
const sop = seq_length - prediction_window + 1
loss(nn, input_mat, output_mat) ≈ norm(output_mat - nn(input_mat)[:, sop:end]) /
↪  norm(output_mat)

# output

true
```

So `TransformerLoss` simply does:

$$\texttt{loss}(\mathcal{NN}, \texttt{input}, \texttt{output}) = ||\mathcal{NN}(\texttt{input})[(\texttt{sl} - \texttt{pw} + 1) : \texttt{end}] - \texttt{output}||/||\texttt{output}||,$$

where $||\cdot||$ is the L_2 norm.

Parameters

The `prediction_window` specifies how many time steps are predicted into the future. It defaults to the value specified for `seq_length`.

source

`GeometricMachineLearning.AutoEncoderLoss` – Type.

```
AutoEncoderLoss()
```

Make an instance of `AutoEncoderLoss`.

This loss should always be used together with a neural network of type `AutoEncoder` (and it is also the default for training such a network).

Example

`AutoEncoderLoss` applies a neural network to an input and compares it to the `output` via an L_2 norm:

```
using GeometricMachineLearning
using LinearAlgebra: norm
import Random
Random.seed!(123)

const N = 4
const n = 1
arch = SymplecticAutoencoder(2*N, 2*n)
nn = NeuralNetwork(arch)

input_vec =  [1., 2., 3., 4., 5., 6., 7., 8.]
loss = AutoEncoderLoss()

loss(nn, input_vec) ≈ norm(input_vec - nn(input_vec)) / norm(input_vec)

# output

true
```

So `AutoEncoderLoss` simply does:

$$\texttt{loss}(\mathcal{NN}, \texttt{input}) = ||\mathcal{NN}(\texttt{input}) - \texttt{input}||/||\texttt{input}||,$$

where $||\cdot||$ is the L_2 norm.

Parameters

This loss does not have any parameters.

source

`GeometricMachineLearning.ReducedLoss` – Type.

```
ReducedLoss(encoder, decoder)
```

Make an instance of `ReducedLoss` based on an `Encoder` and a `Decoder`.

This loss should be used together with a `NeuralNetworkIntegrator` or `TransformerIntegrator`.

Example

`ReducedLoss` applies the *encoder*, *integrator* and *decoder* neural networks in this order to an input and compares it to the `output` via an L_2 norm:

```
using GeometricMachineLearning
using LinearAlgebra: norm
import Random
Random.seed!(123)

const N = 4
const n = 1

Ψᵉ = NeuralNetwork(Chain(Dense(N, n), Dense(n, n))) |> encoder
```

```
Ψᵈ = NeuralNetwork(Chain(Dense(n, n), Dense(n, N))) |> decoder
transformer = NeuralNetwork(StandardTransformerIntegrator(n))

input_mat =  [1.  2.;  3.  4.;  5.  6.;  7.  8.]
output_mat = [9. 10.; 11. 12.; 13. 14.; 15. 16.]
loss = ReducedLoss(Ψᵉ, Ψᵈ)

output_prediction = Ψᵈ(transformer(Ψᵉ(input_mat)))
loss(transformer, input_mat, output_mat) ≈ norm(output_mat - output_prediction) /
↪  norm(output_mat)

# output

true
```

So the loss computes:

$$\text{loss}_{\mathcal{E},\mathcal{D}}(\mathcal{NN}, \text{input}, \text{output}) = ||\mathcal{D}(\mathcal{NN}(\mathcal{E}(\text{input}))) - \text{output}||,$$

where $\mathcal{E}$ is the `Encoder`, $\mathcal{D}$ is the `Decoder`. $\mathcal{NN}$ is the neural network we compute the loss of.

source

`GeometricMachineLearning.projection_error` – Function.

```
projection_error(rs)
```

Compute the projection error for a `HRedSys`.

Arguments

If the full system has already been integrated, then the projection error can be computed quicker:

```
projection_error(rs, sol_full)
```

This saves the cost of again integrating the systems.

source

`GeometricMachineLearning.reduction_error` – Function.

```
reduction_error(rs)
```

Compute the reduction error for a `HRedSys`.

Arguments

If the full system and the reduced system have already been integrated, then the reduction error can be computed quicker:

```
reduction_error(rs, sol_full, sol_reduced)
```

This saves the cost of again integrating the respective systems.

source

4.5 Hamiltonian Model Order Reduction

Hamiltonian PDEs are partial differential equations that, like its ODE counterpart, have a Hamiltonian associated with it. The linear wave equation can be written as such a Hamiltonian PDE with

$$\mathcal{H}(q, p; \mu) := \frac{1}{2} \int_\Omega \mu^2 (\partial_\xi q(t, \xi; \mu))^2 + p(t, \xi; \mu)^2 d\xi.$$

Note that in contrast to the ODE case where the Hamiltonian is a function $H(\cdot; \mu) : \mathbb{R}^{2d} \to \mathbb{R}$, we now have a functional $\mathcal{H}(\cdot, \cdot; \mu) : \mathcal{C}^\infty(\mathcal{D}) \times \mathcal{C}^\infty(\mathcal{D}) \to \mathbb{R}$. The PDE for this Hamiltonian can be obtained similarly as in the ODE case:

$$\partial_t q(t, \xi; \mu) = \frac{\delta \mathcal{H}}{\delta p} = p(t, \xi; \mu), \quad \partial_t p(t, \xi; \mu) = -\frac{\delta \mathcal{H}}{\delta q} = \mu^2 \partial_{\xi\xi} q(t, \xi; \mu)$$

Neglecting the Hamiltonian structure of a system can have grave consequences on the performance of the reduced order model [11, 61, 62] which is why all algorithms in `GeometricMachineLearning` designed for producing reduced order models respect the structure of the system.

The Symplectic Solution Manifold

As with regular parametric PDEs, we also associate a solution manifold with Hamiltonian PDEs. This is a finite-dimensional manifold, on which the dynamics can be described through a Hamiltonian ODE. The reduced system, with which we approximate this symplectic solution manifold, is a low dimensional symplectic vector space $\mathbb{R}^{2n}$ together with a reduction $\mathcal{P}$ and a reconstruction $\mathcal{R}$. If we now take an initial condition on the solution manifold $\hat{u}_0 \in \mathcal{M} \approx \mathcal{R}(\mathbb{R}^{2n})$ and project it to the reduced space with $\mathcal{P}$, we get $u = \mathcal{P}(\hat{u}_0)$. We can now integrate it on the reduced space via the induced differential equation, which is of canonical Hamiltonian form, and obtain an orbit $u(t)$ which can then be mapped back to an orbit on the solution manifold[1] via $\mathcal{R}$. The resulting orbit $\mathcal{R}(u(t))$ is ideally the unique orbit on the full order model $\hat{u}(t) \in \mathcal{M}$.

For Hamiltonian model order reduction we additionally require that the reduction $\mathcal{P}$ satisfies

$$\nabla_z \mathcal{P} \mathbb{J}_{2N} (\nabla_z \mathcal{P})^T = \mathbb{J}_{2n} \text{ for } z \in \mathbb{R}^{2N}$$

and the reconstruction $\mathcal{R}$ satisfies[2]

[1] To be precise, an *approximation of the solution manifold* $\mathcal{R}(\mathbb{R}^{2n})$, as we are not able to find the solution manifold exactly in practice.

[2] We should note that satisfying this *symplecticity condition* is much more important for the reconstruction than for the reduction. There is a lack of research on whether the symplecticity condition for the projection is really needed; in [62] it is entirely ignored for example.

$$(\nabla_z \mathcal{R})^T \mathbb{J}_{2N} \nabla_z \mathcal{R} = \mathbb{J}_{2n}.$$

With this we have

Theorem 4.5.1. *A Hamiltonian system on the reduced space* $(\mathbb{R}^{2n}, \mathbb{J}_{2n}^T)$ *is equivalent to a* non-canonical symplectic system $(\mathcal{M}, \mathbb{J}_{2N}^T|_{\mathcal{M}})$ *where*

$$\mathcal{M} = \mathcal{R}(\mathbb{R}^{2n})$$

is an approximation to the solution manifold. We further have

$$\mathbb{J}_{2N}|_{\mathcal{M}}(z) = ((\nabla_z \mathcal{R})^+)^T \mathbb{J}_{2n} (\nabla_z \mathcal{R})^+,$$

so the dynamics on $\mathcal{M}$ *can be described through a Hamiltonian ODE on* $\mathbb{R}^{2n}$.

For the proof we use the fact that $\mathcal{M} = \mathcal{R}(\mathbb{R}^{2n})$ is a manifold whose coordinate chart is the local inverse (see Chapter 2.5) of $\mathcal{R}$ which we will call ψ, i.e. around a point $y \in \mathcal{M}$ we have $\psi \circ \mathcal{R}(y) = y$.[3] We further define the *symplectic inverse* of a matrix $A \in \mathbb{R}^{2N \times 2n}$ as

$$A^+ = \mathbb{J}_{2n}^T A^T \mathbb{J}_{2N},$$

which gives:

$$A^+ A = \mathbb{J}_{2n}^T A^T \mathbb{J}_{2N} A = \mathbb{I}_{2n},$$

iff A is symplectic, i.e. $A^T \mathbb{J}_{2N} A = \mathbb{J}_{2n}$.

Proof. Note that the tangent space at $y = \mathcal{R}(z)$ to $\mathcal{M}$ is:

$$T_y \mathcal{M} = \{(\nabla_z \mathcal{R}) v : v \in \mathbb{R}^{2n}\}.$$

The mapping

$$\mathcal{M} \to T\mathcal{M}, y \mapsto (\nabla_z \mathcal{R}) \mathbb{J}_{2n} \nabla_z H$$

[3] A similar proof can be found in [63]. Further note that, if we enforced the condition $\mathcal{P} \circ \mathcal{R} = \mathrm{id}$ exactly, the projection $\mathcal{P}$ would be equal to the local inverse ψ. For the proof here we however only require the existence of ψ, not its explicit construction as $\mathcal{P}$.

is clearly a vector field. We now prove that it is symplectic and equal to $\mathbb{J}_{2N}\nabla_y(H \circ \psi)$. For this first note that we have $\mathbb{I} = (\nabla_z \mathcal{R})^+ \nabla_z \mathcal{R} = (\nabla_{\mathcal{R}(z)}\psi)\nabla_z \mathcal{R}$ and that the pseudoinverse is unique. We then have:

$$\mathbb{J}_{2N}\nabla_y H \circ \psi = \mathbb{J}_{2N}(\nabla_y \psi)^T \nabla_{\psi(y)} H = \mathbb{J}_{2N}\left((\nabla_{\psi(y)}\mathcal{R})^+\right)^T \nabla_{\psi(y)} H = (\nabla_{\psi(y)}\mathcal{R})\mathbb{J}_{2n}\nabla_{\psi(y)} H,$$

which proves that every Hamiltonian system on $\mathbb{R}^{2n}$ induces a Hamiltonian system on $\mathcal{M}$. Conversely assume we are given a Hamiltonian vector field whose flow map evolves on $\mathcal{M}$, which we denote by

$$\hat{X}(z) = \mathbb{J}_{2N}\nabla_z \hat{H} = (\nabla_{\psi(z)}\mathcal{R})\bar{X}(\psi(z)),$$

where $\bar{X}$ is a vector field on the reduced space. In the last equality we used that the flow map evolves on $\mathcal{M}$, so the corresponding vector field needs to map to $T\mathcal{M}$. We further have:

$$(\nabla_{\psi(z)}\mathcal{R})^+ \hat{X}(z) = \mathbb{J}_{2n}(\nabla_{\psi(z)}\mathcal{R})^T \nabla_z \hat{H} = \mathbb{J}_{2n}\nabla_{\psi(z)}(\hat{H} \circ \mathcal{R}) = \bar{X}(\psi(z)),$$

and we see that the vector field on the reduced space also has to be Hamiltonian. We can thus express a high-dimensional Hamiltonian system on $\mathcal{M}$ with a low-dimensional Hamiltonian system on $\mathbb{R}^{2n}$. □

In the proof we used that *the pseudoinverse is unique*. This is not true in general [11], but holds for the architectures discussed here (proper symplectic decomposition and symplectic autoencoders). We will postpone the proof of this until after we introduced symplectic autoencoders in detail (see Chapter 8.2).

This theorem serves as the basis for Hamiltonian model order reduction via proper symplectic decomposition and symplectic autoencoders. We will now briefly introduce these two approaches[4].

Proper Symplectic Decomposition

For proper symplectic decomposition (PSD) the reduction $\mathcal{P}$ and the reconstruction $\mathcal{R}$ are constrained to be linear, orthonormal and symplectic. Note that these first two properties are shared with POD (see Chapter 4.2). The easiest way[5] to enforce this is through the so-called "cotangent lift" [11]:

$$\mathcal{R} \equiv \Psi_{\text{CL}} = \begin{bmatrix} \Phi & \mathbb{O} \\ \mathbb{O} & \Phi \end{bmatrix} \text{ where } \Phi \in St(n, N) \subset \mathbb{R}^{N \times n},$$

i.e. both Φ and Ψ_{CL} are elements of the Stiefel manifold (see Chapter 2.9) and we furthermore have $\Psi_{\text{CL}}^T \mathbb{J}_{2N} \Psi_{\text{CL}} = \mathbb{J}_{2n}$, i.e. Ψ_{CL} is symplectic. If the snapshot matrix (see Chapter A.1) is of the form:

[4] We will discuss symplectic autoencoders later in a dedicated section (see Chapter 8.2).

[5] The original PSD paper [11] proposes another approach to build linear reductions and reconstructions with the so-called "complex SVD." In practice this only brings minor advantages over the cotangent lift however [61].

$$M = \left[\begin{array}{c|c|c|c} \hat{q}_1(t_0) & \hat{q}_1(t_1) & \dots & \hat{q}_1(t_f) \\ \hat{q}_2(t_0) & \hat{q}_2(t_1) & \dots & \hat{q}_2(t_f) \\ \dots & \dots & \dots & \dots \\ \hat{q}_N(t_0) & \hat{q}_N(t_1) & \dots & \hat{q}_N(t_f) \\ \hat{p}_1(t_0) & \hat{p}_1(t_1) & \dots & \hat{p}_1(t_f) \\ \hat{p}_2(t_0) & \hat{p}_2(t_1) & \dots & \hat{p}_2(t_f) \\ \dots & \dots & \dots & \dots \\ \hat{p}_N(t_0) & \hat{p}_N(t_1) & \dots & \hat{p}_N(t_f) \end{array}\right],$$

with $\texttt{nts} := f - 1$ is *the number of time steps*. Then Φ_{CL} can be computed in a very straight-forward manner:

Theorem 4.5.2. *The ideal cotangent lift Ψ_{CL} for the snapshot matrix of form*

$$M = \begin{bmatrix} M_q \\ M_p \end{bmatrix},$$

i.e. the cotangent lift that minimizes the projection error, can be obtained the following way:

1. *Rearrange the rows of the matrix M such that we end up with a $N \times (2\texttt{nts})$ matrix: $\hat{M} := [M_q, M_p]$.*
2. *Perform SVD: $\hat{M} = V\Sigma U^T$.*
3. *Set $\Phi \leftarrow U[:, 1:\texttt{n}]$.*

Ψ_{CL} is then built based on this Φ.

For details on the cotangent lift (and other methods for linear symplectic model reduction) consult [11]. In `GeometricMachineLearning` we use the function `solve!` for this task.

Symplectic Autoencoders

Symplectic Autoencoders are a type of neural network suitable for treating Hamiltonian parametrized PDEs with slowly decaying Kolmogorov n-width. It is based on PSD and symplectic neural networks (see Chapter 8.5)[6].

Symplectic autoencoders are motivated similarly to standard autoencoders for model order reduction: PSD suffers from similar shortcomings as regular POD. PSD is a linear map and the approximation space $\tilde{\mathcal{M}} = \{\Psi^{\text{dec}}(z_r) \in \mathbb{R}^{2N} : z_r \in \mathbb{R}^{2n}\}$ is therefore also linear. For problems with slowly-decaying Kolmogorov n-width this leads to very poor approximations.

In order to overcome this difficulty we use neural networks, more specifically SympNets (see Chapter 8.5), together with cotangent lift-like matrices. The resulting architecture, symplectic autoencoders, are discussed in the dedicated section on neural network architectures (see Chapter 8.2).

[6] We call these SympNets most of the time. This term was coined in [10].

Workflow for Symplectic ROM

As with any other data-driven reduced order modeling technique (see Chapter 4.1) we first discretize the PDE. This should be done with a structure-preserving scheme, thus yielding a (high-dimensional) Hamiltonian ODE as a result. Going back to the example of the linear wave equation, we can discretize this equation with finite differences to obtain a Hamiltonian ODE:

$$\mathcal{H}_{\text{discr}}(z(t;\mu);\mu) := \frac{1}{2}x(t;\mu)^T \begin{bmatrix} -\mu^2 D_{\xi\xi} & \mathbb{O} \\ \mathbb{O} & \mathbb{I} \end{bmatrix} x(t;\mu).$$

In Hamiltonian reduced order modeling we try to find a symplectic submanifold in the solution space[7] that captures the dynamics of the full system as well as possible.

Similar to the regular PDE case we again build an encoder $\mathcal{P} \equiv \Psi^{\text{enc}}$ and a decoder $\mathcal{R} \equiv \Psi^{\text{dec}}$; but now both these mappings are required to be symplectic.

Concretely this means:

1. The encoder is a mapping from a high-dimensional symplectic space to a low-dimensional symplectic space, i.e. $\Psi^{\text{enc}} : \mathbb{R}^{2N} \to \mathbb{R}^{2n}$ such that $\nabla\Psi^{\text{enc}}\mathbb{J}_{2N}(\nabla\Psi^{\text{enc}})^T = \mathbb{J}_{2n}$.
2. The decoder is a mapping from a low-dimensional symplectic space to a high-dimensional symplectic space, i.e. $\Psi^{\text{dec}} : \mathbb{R}^{2n} \to \mathbb{R}^{2N}$ such that $(\nabla\Psi^{\text{dec}})^T\mathbb{J}_{2N}\nabla\Psi^{\text{dec}} = \mathbb{J}_{2n}$.

If these two maps are constrained to linear maps this amounts to PSD.

After we obtained Ψ^{enc} and Ψ^{dec} we can construct the reduced model. `GeometricMachineLearning` has a *symplectic Galerkin projection* implemented. This symplectic Galerkin projection does:

$$\begin{pmatrix} v_r(q,p) \\ f_r(q,p) \end{pmatrix} = \frac{d}{dt}\begin{pmatrix} q \\ p \end{pmatrix} = \mathbb{J}_{2n}(\nabla_z\mathcal{R})^T\mathbb{J}_{2N}^T \begin{pmatrix} v(\mathcal{R}(q,p)) \\ f(\mathcal{R}(q,p)) \end{pmatrix},$$

where v are the first n components of the vector field and f are the second n components of the vector field. The superscript r indicated a *reduced* vector field. These reduced vector fields are built with `GeometricMachineLearning.build_v_reduced` and `GeometricMachineLearning.build_f_reduced`.

Library Functions

`GeometricMachineLearning.SymplecticEncoder` – Type.

```
SymplecticEncoder <: Encoder
```

This is the abstract `SymplecticEncoder` type.

See `Encoder` for the super type and `NonLinearSymplecticEncoder` for a derived `struct`.

source

[7] The submanifold, that approximates the solution manifold, is $\tilde{\mathcal{M}} = \{\Psi^{\text{dec}}(z_r) \in \mathbb{R}^{2N} : u_r \in \mathrm{R}^{2n}\}$ where z_r is the reduced state of the system. By a slight abuse of notation we also denote $\tilde{\mathcal{M}}$ by $\mathcal{M}$ as we have done previously when showing equivalence between Hamiltonian vector fields on $\mathbb{R}^{2n}$ and $\mathcal{M}$.

`GeometricMachineLearning.SymplecticDecoder` – Type.

```
SymplecticDecoder <: Decoder
```

This is the abstract `SymplecticDecoder` type.

See `Decoder` for the super type and `NonLinearSymplecticDecoder` for a derived `struct`.

source

`GeometricMachineLearning.NonLinearSymplecticEncoder` – Type.

```
NonLinearSymplecticEncoder
```

This should not be called direclty. Instad use `SymplecticAutoencoder`.

An instance of `NonLinearSymplecticEncoder` can be generated by calling `encoder` on an instance of `SymplecticAutoencoder`.

source

`GeometricMachineLearning.NonLinearSymplecticDecoder` – Type.

```
NonLinearSymplecticDecoder
```

This should not be called direclty. Instad use `SymplecticAutoencoder`.

An instance of `NonLinearSymplecticDecoder` can be generated by calling `decoder` on an instance of `SymplecticAutoencoder`.

source

`GeometricMachineLearning.HRedSys` – Type.

```
HRedSys
```

`HRedSys` computes the reconstructed dynamics in the full system based on the reduced one. Optionally it can be compared to the FOM solution.

It can be called using two different constructors.

Constructors

The standard constructor is:

```
HRedSys(N, n; encoder, decoder, v_full, f_full, v_reduced, f_reduced, parameters, timespan,
↪  timestep, ics, projection_error)
```

where

- `encoder`: a function $\mathbb{R}^{2N} \mapsto \mathbb{R}^{2n}$

- `decoder`: a (differentiable) function $\mathbb{R}^{2n} \mapsto \mathbb{R}^{2N}$
- `v_full`: a (differentiable) mapping defined the same way as in GeometricIntegrators.
- `f_full`: a (differentiable) mapping defined the same way as in GeometricIntegrators.
- `v_reduced`: a (differentiable) mapping defined the same way as in GeometricIntegrators.
- `f_reduced`: a (differentiable) mapping defined the same way as in GeometricIntegrators.
- `integrator`: is used for integrating the reduced system.
- `parameters`: a NamedTuple that parametrizes the vector fields (the same for full*vector*field and reduced*vector*field)
- `timespan`: a tuple `(t₀, tₗ)` that specifies start and end point of the time interval over which integration is performed.
- `timestep`: the time step
- `ics`: the initial condition for the big system.

The other, and more convenient, constructor is:

```
HRedSys(odeensemble, encoder, decoder)
```

where `odeensemble` can be a `HODEEnsemble` or a `HODEProblem`. `encoder` and `decoder` have to be neural networks. With this constructor one can also pass an integrator via the keyword argument `integrator`. The default is `ImplicitMidpoint()`. Internally this calls the functions `build_v_reduced` and `build_f_reduced` in order to build the reduced vector fields.

source

`GeometricMachineLearning.build_v_reduced` – Function.

```
build_v_reduced(v_full, f_full, decoder)
```

Builds the reduced vector field (q part) based on the full vector field for a Hamiltonian system.

We derive the reduced vector field via the reduced Hamiltonian: $\tilde{H} := H \circ \Psi^{\text{dec}}$.

We then get

$$\begin{aligned}\mathbb{J}_{2n}\nabla_\xi\tilde{H} = \mathbb{J}_{2n}(\nabla\Psi^{\text{dec}})^T\mathbb{J}_{2N}^T\mathbb{J}_{2N}\nabla_z H = \mathbb{J}_{2n}(\nabla\Psi^{\text{dec}})^T\mathbb{J}_{2N}^T\begin{pmatrix} v(z) \\ f(z)\end{pmatrix} \\ = \begin{pmatrix} -(\nabla_p\Psi_q)^T f(z) + (\nabla_p\Psi_p)^T v(z) \\ (\nabla_q\Psi_q)^T f(z) - (\nabla_q\Psi_p)^T v(z)\end{pmatrix}.\end{aligned}$$

`build_v_reduced` outputs the first half of the entries of this vector field.

source

`GeometricMachineLearning.build_f_reduced` – Function.

```
build_f_reduced(v_full, f_full, decoder)
```

Builds the reduced vector field (p part) based on the full vector field for a Hamiltonian system.

`build_f_reduced` outputs the second half of the entries of this vector field.

See `build_v_reduced` for more information.

source

`GeometricMachineLearning.PSDLayer` – Type.

```
PSDLayer(input_dim, output_dim)
```

Make an instance of `PSDLayer`.

This is a PSD-like layer used for symplectic autoencoders. One layer has the following shape:

$$A = \begin{bmatrix} \Phi & \mathbb{O} \\ \mathbb{O} & \Phi \end{bmatrix},$$

where Φ is an element of the Stiefel manifold $St(n, N)$.

source

`GeometricMachineLearning.PSDArch` – Type.

```
PSDArch <: SymplecticCompression
```

`PSDArch` is the architecture that corresponds to `PSDLayer`.

The architecture

Proper symplectic decomposition (PSD) can be seen as a `SymplecticAutoencoder` for which the decoder and the encoder are both PSD-like matrices (see the docs for `PSDLayer`.

Training

For optimizing the parameters in this architecture no neural network training is necessary (see the docs for `solve!`).

The constructor

The constructor only takes two arguments as input:

- `full_dim::Integer`
- `reduced_dim::Integer`

source

`GeometricMachineLearning.solve!` – Function.

```
solve!(nn::NeuralNetwork{<:PSDArch}, input)
```

Solve the cotangent lift problem for an input.

`PSDArch` does not require neural network training since it is a strictly linear operation that can be solved with singular value decomposition (SVD).

source

Chapter Summary

In this chapter we introduced the concept of *data-driven reduced order modeling*. It was shown how data-driven reduced order modeling can be motivated by the *solution manifold*, the (nonlinear) space containing all the solutions to the (parameter-dependent) differential equation. *Proper orthogonal decomposition* (POD) and *autoencoders* were discussed as specific examples of performing the offline phase in reduced order modeling. POD is an application of *singular value decomposition* and as such belongs to the realm of linear algebra. Autoencoders are more general approximators that are build with neural networks and have to be optimized during a *training stage*. Autoencoders are especially well-suited for problems with a *slowly-decaying Kolmogorov n-width*. The Kolmogorov n-width provides a quantitative measure for how well something can be approximated with a linear subspace.

We furthermore discussed how a reduced order model can be made structure-preserving and showed *proper symplectic decomposition* (PSD) to be the structure-preserving alternative to POD. PSD is a more restrictive form of POD where additional conditions on the reduction $\mathcal{P}$ and the reconstruction $\mathcal{R}$ are imposed. With this construction a Hamiltonian full order model is reduced to a Hamiltonian reduced order model. At the very end of the chapter we teased *symplectic autoencoders* which will be discussed in detail in Part III. These offer, similar to standard autoencoders, a way of constructing more general symplectic reductions and are thus well-suited to problems with a slowly-decaying Kolmogorov n-width.

References

[55] K. Lee and K. T. Carlberg. *Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders.* Journal of Computational Physics **404**, 108973 (2020).

[54] S. Fresca, L. Dede' and A. Manzoni. *A comprehensive deep learning-based approach to reduced order modeling of nonlinear time-dependent parametrized PDEs.* Journal of Scientific Computing **87**, 1–36 (2021).

[53] T. Blickhan. *A registration method for reduced basis problems using linear optimal transport*, arXiv preprint arXiv:2304.14884 (2023).

[59] A. Chatterjee. *An introduction to the proper orthogonal decomposition.* Current science, 808–817 (2000).

[11] L. Peng and K. Mohseni. *Symplectic model reduction of Hamiltonian systems.* SIAM Journal on Scientific Computing **38**, A1–A27 (2016).

[62] P. Buchfink, S. Glas and B. Haasdonk. *Symplectic model reduction of Hamiltonian systems on nonlinear manifolds and approximation with weakly symplectic autoencoder.* SIAM Journal on Scientific Computing **45**, A289–A311 (2023).

Part II

Optimizers

Chapter 5

General Framework for Manifold Optimization

In this chapter we introduce a *general framework for manifold optimization* that is needed to efficiently train symplectic autoencoders. We start this chapter by discussing optimization for neural network in general and explain how we can generalize this to homogeneous spaces. We will see that an important ingredient for doing so are *retractions* which we then elaborate on. After discussing how to make the computation of retractions efficient for homogeneous spaces we conclude the chapter by introducing the notion of *parallel transport* which we need to extend the notion of *momentum* in neural network optimization.

5.1 Neural Network Optimizers

In this section we present the general Optimizer framework used in `GeometricMachineLearning`. For more information on the particular steps involved in this consult the documentation on the various optimizer methods such as the gradient optimizer, the momentum optimizer and the Adam optimizer (see Chapter 6.1), and the documentation on retractions (see Chapter 5.2).

During *optimization* we aim at changing the neural network parameters in such a way to minimize the loss function. A loss function assigns a scalar value to the weights that parametrize the neural network (see Chapter 3.3):

$$L : \mathbb{P} \to \mathbb{R}_{\geq 0}, \quad \Theta \mapsto L(\Theta),$$

where $\mathbb{P}$ is the parameter space. We can then phrase the optimization task as:

Definition 5.1.1. Given a neural network $\mathcal{NN}$ parametrized by Θ and a loss function $L : \mathbb{P} \to \mathbb{R}$ we call an algorithm an **iterative optimizer** (or simply **optimizer**) if it performs the following task:

$$\Theta \leftarrow \texttt{Optimizer}(\Theta, \text{past history}, t),$$

with the aim of decreasing the value $L(\Theta)$ in each optimization step.

The past history of the optimization is stored in a cache (`AdamCache`, `MomentumCache`, `GradientCache`, ...) in `GeometricMachineLearning`.

Optimization for neural networks is (almost always) some variation on gradient descent. The most basic form of gradient descent is a discretization of the *gradient flow equation*:

$$\dot{\Theta} = -\nabla_\Theta L,$$

by means of an Euler time-stepping scheme:

$$\Theta^{t+1} = \Theta^t - h\nabla_{\Theta^t} L,$$

where η (the time step of the Euler scheme) is referred to as the *learning rate*.

This equation can easily be generalized to manifolds (see Chapter 2.5) with the following two steps:

1. modify $-\nabla_{\Theta^t} L \implies -h\text{grad}_{\Theta^t} L$, i.e. replace the Euclidean gradient by a Riemannian gradient (see Chapter 2.7) and
2. replace addition with the geodesic map (see Chapter 2.7).

To sum up, we then have:

$$\Theta^{t+1} = \text{geodesic}(\Theta^t, -h\text{grad}_{\Theta^t} L).$$

In practice we very often do not use the geodesic map but approximations thereof. These approximations are called retractions (see Chapter 5.2).

Generalization to Homogeneous Spaces

In order to generalize neural network optimizers to homogeneous spaces (see Chapter 2.8) we utilize their corresponding global tangent space representation (see Chapter 2.11) $\mathfrak{g}^{\text{hor}}$.

When introducing the notion of a global tangent space (see Chapter 2.11) we discussed how an element of the tangent space $T_Y\mathcal{M}$ can be represented in $\mathfrak{g}^{\text{hor}}$ by performing two mappings:

1. the first one is the horizontal lift Ω (see the docstring for `GeometricMachineLearning.Ω`) and
2. the second one is performing the adjoint operation[1] of $\lambda(Y)$, the section of Y, on $\Omega(\Delta)$.

The two steps together are performed as `global_rep` in `GeometricMachineLearning`. So we lift to $\mathfrak{g}^{\text{hor}}$:

$$\texttt{global\_rep} : T_Y\mathcal{M} \to \mathfrak{g}^{\text{hor}},$$

and then perform all the steps of the optimizer in $\mathfrak{g}^{\text{hor}}$. We can visualize all the steps required in the generalization of the optimizers:

[1] By the *adjoint operation* $\text{ad}_A : \mathfrak{g} \to \mathfrak{g}$ for an element $A \in G$ we mean $B \mapsto A^{-1}BA$.

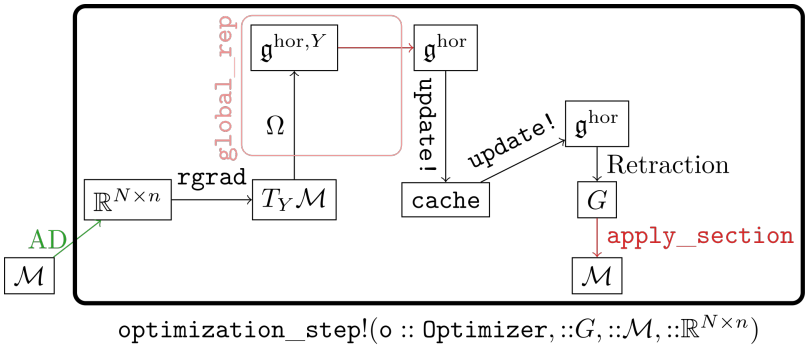


Figure 5.1: Schematic visualization of neural network optimizers to homogeneous spaces.

This picture summarizes all steps involved in an optimization step:

1. map the Euclidean gradient $\nabla L \in \mathbb{R}^{N\times n}$ that was obtained via automatic differentiation (see Chapter B.3) to the Riemannian gradient $\mathrm{grad}L \in T_Y\mathcal{M}$ with the function `rgrad`,
2. obtain the global tangent space representation of $\mathrm{grad}L$ in $\mathfrak{g}^{\mathrm{hor}}$ with the function `global_rep`,
3. perform an `update!`; this consists of two steps: (i) update the cache and (ii) output a *final velocity*,
4. use this final velocity to update the global section (see Chapter 2.11) $\Lambda \in G$,
5. use the updated global section to update the neural network weight $\in \mathcal{M}$. This is done with `apply_section`.

The `cache` stores information about previous optimization steps and is dependent on the optimizer. Typically the cache is represented as one or more elements in $\mathfrak{g}^{\mathrm{hor}}$. Based on this the optimizer method (represented by `update!` in the figure) computes a *final velocity*. This final velocity is again an element of $\mathfrak{g}^{\mathrm{hor}}$. The particular form of the cache and the updating rule depends on which optimizer method we use (see Chapter 6.1).

The final velocity is then fed into a retraction (see Chapter 5.2)[2]. For computational reasons we split the retraction into two steps, referred to as "Retraction" and `apply_section` above. These two mappings together are equivalent to:

$$\mathrm{retraction}(\Delta) = \mathrm{retraction}(\lambda(Y)B^\Delta E) = \lambda(Y)\mathrm{Retraction}(B^\Delta),$$

where $\Delta \in T_{\mathcal{M}}$ and B^Δ is its representation in $\mathfrak{g}^{\mathrm{hor}}$ as $B^\Delta = \lambda(Y)^{-1}\Omega(\Delta)\lambda(Y)$.

Library Functions

`GeometricMachineLearning.Optimizer` – Type.

[2] A retraction is an approximation of the geodesic map (see Chapter 2.7)

```
Optimizer(method, cache, step, retraction)
```

Store the `method` (e.g. `AdamOptimizer` with corresponding hyperparameters), the `cache` (e.g. `AdamCache`), the optimization step and the retraction.

It takes as input an optimization method and the parameters of a network.

Before one can call `Optimizer` a `OptimizerMethod` that stores all the hyperparameters of the optimizer needs to be specified.

Functor

For an instance `o` of `Optimizer`, we can call the corresponding functor as:

```
o(nn, dl, batch, n_epochs, loss)
```

The arguments are:

1. `nn::NeuralNetwork`
2. `dl::DataLoader`
3. `batch::Batch`
4. `n_epochs::Integer`
5. `loss::NetworkLoss`

The last argument is optional for many neural network architectures. We have the following defaults:

- A `TransformerIntegrator` uses `TransformerLoss`.
- A `NeuralNetworkIntegrator` uses `FeedForwardLoss` (this loss is defined in `AbstractNeuralNetworks`).
- An `AutoEncoder` uses `AutoEncoderLoss`.

In addition there is an optional keyword argument that can be supplied to the functor:

- `show_progress=true`: This specifies whether a progress bar should be shown during training.

Implementation

Internally the functor for `Optimizer` calls `GlobalSection` once at the start and then `optimize_for_one_epoch!` for each epoch.

source

`GeometricMachineLearning.optimize_for_one_epoch!` – Function.

```
optimize_for_one_epoch!(opt, model, ps, dl, batch, loss, λY)
```

Sample the data contained in `dl` according to `batch` and optimize for these batches.

This step also performs automatic differentiation on `loss`.

The output of `optimize_for_one_epoch!` is the average loss over all batches of the epoch:

$$output = \frac{1}{\texttt{steps\_per\_epoch}} \sum_{t=1}^{\texttt{steps\_per\_epoch}} \texttt{loss}(\theta^{(t-1)}).$$

This is done because any *reverse differentiation* routine always has two outputs; for `Zygote`:

```
loss_value, pullback = Zygote.pullback(ps -> loss(model, ps, input, output), ps)
```

So we get the value for the loss for free whenever we compute the pullback with AD.

Arguments

All the arguments are mandatory (there are no defaults):

1. an instance of `Optimizer`.
2. the neural network model.
3. the neural network parameters `ps`.
4. the data (i.e. an instance of `DataLoader`).
5. `batch`::`Batch`: stores `batch_size` (and optionally `seq_length` and `prediction_window`).
6. `loss`::`NetworkLoss`.
7. the *section* `λY` of the parameters `ps`.

Implementation

Internally this calls `optimization_step!` for each minibatch.

The number of minibatches can be determined with `number_of_batches`:

```
using GeometricMachineLearning
using GeometricMachineLearning: number_of_batches

data = [1, 2, 3, 4, 5]
batch = Batch(2)
dl = DataLoader(data; suppress_info = true)

number_of_batches(dl, batch)

# output

3
```

source

`GeometricMachineLearning.optimization_step!` – Function.

```
optimization_step!(o, λY, ps, dx)
```

Update the weights `ps` based on an `Optimizer`, a `cache` and first-order derivatives `dx`.

`optimization_step!` is calling `update!` internally. `update!` has to be implemented for every `OptimizerMethod`.

Arguments

All arguments into `optimization_step!` are mandatory:

1. `o::Optimizer`,
2. `λY::NamedTuple`: this named tuple has the same keys as `ps`, but contains `GlobalSection`s,
3. `ps::NamedTuple`: the neural network parameters,
4. `dx::Union{NamedTuple, NeuralNetworkParameters}`: the gradients stores as a NamedTuple.

All the arguments are given as `NamedTuple`s as the neural network weights are stores in that format.

```
using GeometricMachineLearning
using GeometricMachineLearning: params

l = StiefelLayer(3, 5)
ps = params(NeuralNetwork(Chain(l), Float32)).L1
cache = apply_toNT(MomentumCache, ps)
o = Optimizer(MomentumOptimizer(), cache, 0, geodesic)
λY = GlobalSection(ps)
dx = (weight = rand(Float32, 5, 3), )

# call the optimizer
optimization_step!(o, λY, ps, dx)

_test_nt(x) = typeof(x) <: NamedTuple

_test_nt(λY) & _test_nt(ps) & _test_nt(cache) & _test_nt(dx)

# output

true
```

Extended help

The derivatives `dx` here are usually obtained via an AD routine by differentiating a loss function, i.e. `dx` is $\nabla_x L$.

source

5.2 Retractions

In practice we usually do not solve the geodesic equation exactly in each optimization step (even though this is possible and computationally feasible), but prefer approximations that are called "retractions" [25] for numerical stability. The definition of a retraction in `GeometricMachineLearning` is slightly different from how it is usually defined in textbooks [3, 25]. We discuss these differences here.

Classical Retractions

By "classical retraction" we here mean the textbook definition.

Definition 5.2.1. A **classical retraction** is a smooth map

$$R : T\mathcal{M} \to \mathcal{M} : (x, v) \mapsto R_x(v),$$

such that each curve $c(t) := R_x(tv)$ is a local approximation of a geodesic, i.e. the following two conditions hold:

1. $c(0) = x$ and
2. $c'(0) = v$.

Perhaps the most common example for matrix manifolds is the *Cayley retraction*. It is a retraction for many matrix Lie groups [3, 22, 64].

Example 5.2.2. The **Cayley retraction** for $V \in T_{\mathbb{I}}G \equiv \mathfrak{g}$ is defined as

$$\text{Cayley}(V) = \left(\mathbb{I} - \frac{1}{2}V\right)^{-1}\left(\mathbb{I} + \frac{1}{2}V\right).$$

We show that the Cayley transform is a retraction for $G = SO(N)$ at $\mathbb{I} \in SO(N)$:

Proof. The Cayley transform trivially satisfies $\text{Cayley}(\mathbb{O}) = \mathbb{I}$. So what we have to show is the second condition for a retraction and that $\text{Cayley}(V) \in SO(N)$. For this take $V \in \mathfrak{so}(N)$. We then have

$$\left.\frac{d}{dt}\right|_{t=0} \text{Cayley}(tV) = \left.\frac{d}{dt}\right|_{t=0} \left(\mathbb{I} - \frac{1}{2}tV\right)^{-1}\left(\mathbb{I} + \frac{1}{2}tV\right) = \frac{1}{2}V - \frac{1}{2}V^T = V,$$

which satisfies the second condition. We further have

$$\left.\frac{d}{dt}\right|_{t=0} (\text{Cayley}(tV))^T \text{Cayley}(tV) = (\frac{1}{2}V - \frac{1}{2}V^T)^T + \frac{1}{2}V - \frac{1}{2}V^T = 0.$$

This proofs that the Cayley transform maps to $SO(N)$. □

We should mention that the factor $\frac{1}{2}$ is sometimes left out in the definition of the Cayley transform when used in different contexts. But it is necessary for defining a retraction as without it the second condition is not satisfied.

Remark 5.2.3. We can also use the Cayley retraction at a different point than the identity $\mathbb{I}$. For this consider $\bar{A} \in SO(N)$ and $\bar{B} \in T_{\bar{A}}SO(N) = \{\bar{B} \in \mathbb{R}^{N\times N} : \bar{A}^T\bar{B} + \bar{B}^T\bar{A} = \mathbb{O}\}$. We then have $\bar{A}^T\bar{B} \in \mathfrak{so}(N)$ and

$$\overline{\text{Cayley}} : T_{\bar{A}}SO(N) \to SO(N), \bar{B} \mapsto \bar{A}\text{Cayley}(\bar{A}^T\bar{B}),$$

is a retraction $\forall \bar{A} \in SO(N)$.

As a retraction is always an approximation of the geodesic map, we now compare the `cayley` retraction for the example we introduced along Riemannian manifolds (see Chapter 2.7):

```
η_increments = 0.2 : 0.2 : 5.4
Δ_increments = [Δ * η for η in η_increments]

Y_increments_geodesic = [geodesic(Y, Δ_increment) for Δ_increment in Δ_increments]
Y_increments_cayley = [cayley(Y, Δ_increment) for Δ_increment in Δ_increments]
```

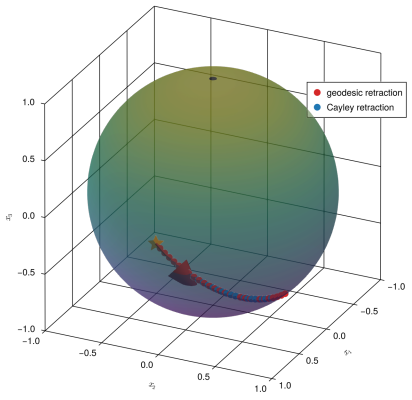


Figure 5.2: Comparison between the geodesic and the Cayley retraction.

We see that for small Δ increments the Cayley retraction seems to match the geodesic retraction very well, but for larger values there is a notable discrepancy. We can plot this discrepancy directly:

```
zip_ob = zip(Y_increments_geodesic, Y_increments_cayley, axes(Y_increments_geodesic, 1))
discrepancies = [norm(Y_geo_inc - Y_cay_inc) for (Y_geo_inc, Y_cay_inc, _) in zip_ob]
nothing
```

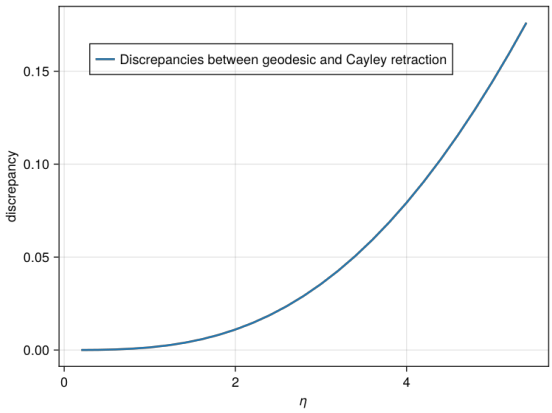


Figure 5.3: Discrepancy between the geodesic and the Cayley retraction.

In `GeometricMachineLearning`

The way we use *retractions*[1] in `GeometricMachineLearning` is slightly different from their classical definition:

Definition 5.2.4. Given a section $\lambda : \mathcal{M} \to G$, where $\mathcal{M}$ is a homogeneous space, a **retraction** is a map Retraction : $\mathfrak{g}^{\text{hor}} \to G$ such that

$$\Delta \mapsto \lambda(Y)\text{Retraction}(\lambda(Y)^{-1}\Omega(\Delta)\lambda(Y))E,$$

is a classical retraction.

This map Retraction is also what was visualized in the figure on the general optimization framework (see Chapter 5.1). We now discuss how the geodesic retraction (exponential map) and the Cayley retraction are implemented in `GeometricMachineLearning`.

Retractions for Homogeneous Spaces

Here we harness special properties of homogeneous spaces to obtain computationally efficient retractions for the Stiefel manifold (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10). This is also discussed in e.g. [21, 22].

The *geodesic retraction* is a retraction whose associated curve is also the unique geodesic. For many matrix Lie groups (including $SO(N)$) geodesics are obtained by simply evaluating the exponential map [25, 65]:

Theorem 5.2.5. *The geodesic on a compact matrix Lie group G with bi-invariant metric for $\bar{B} \in T_{\bar{A}}G$ is simply*

[1] Classical retractions are also defined in `GeometricMachineLearning` under the same name, i.e. there is e.g. a method `cayley(::StiefelLieAlgHorMatrix)` and a method `cayley(::StiefelManifold, ::AbstractMatrix)` (the latter being the classical retraction); but the user is *strongly discouraged* from using classical retractions as these are computationally inefficient.

$$\gamma(t) = \exp(t \cdot \bar{B}\bar{A}^{-1})\bar{A} = A\exp(t \cdot \bar{A}^{-1}\bar{B}^n),$$

where $\exp : \mathfrak{g} \to G$ *is the matrix exponential map.*

The last equality in the equation above is a result of:

$$\begin{aligned}\exp(\bar{A}^{-1}\hat{B}\bar{A}) = \sum_{k=1}^{\infty} \frac{1}{k!}(\bar{A}^{-1}\hat{B}\bar{A})^k &= \sum_{k=1}^{\infty} \frac{1}{k!} \underbrace{(\bar{A}^{-1}\hat{B}\bar{A}) \cdots (A^{-1}\hat{B}\bar{A})}_{k \text{ times}} \\ &= \sum_{k=1}^{\infty} \frac{1}{k!} \bar{A}^{-1}\hat{B}^k\bar{A} = \bar{A}^{-1}\exp(\hat{B})\bar{A}.\end{aligned}$$

Because $SO(N)$ is compact and we furnish it with the canonical metric, i.e.

$$g : T_{\bar{A}}G \times T_{\bar{A}}G \to \mathbb{R}, (B_1, B_2) \mapsto \mathrm{Tr}(B_1^T B_2) = \mathrm{Tr}((B_1\bar{A}^{-1})^T(B_2\bar{A}^{-1})),$$

its geodesics are thus equivalent to the exponential maps. We now use this observation to obtain an expression for the geodesics on the Stiefel manifold (see Chapter 2.9). We use the following theorem from [65, Proposition 25.7]:

Theorem 5.2.6. *The geodesics for a naturally reductive homogeneous space* $\mathcal{M}$ *starting at* Y *are given by:*

$$\gamma_\Delta(t) = \exp(t \cdot \Omega(\Delta))Y,$$

where the exp *is the exponential map for the Lie group* G *corresponding to* $\mathcal{M}$*.*

The theorem requires the homogeneous space to be naturally reductive:

Definition 5.2.7. A homogeneous space is called **naturally reductive** if the following two conditions hold:

1. $\bar{A}^{-1}\bar{B}\bar{A} \in \mathfrak{g}^{\mathrm{hor}}$ for every $\bar{B} \in \mathfrak{g}^{\mathrm{hor}}$ and $\bar{A} \in \exp(\mathfrak{g}^{\mathrm{ver}})$,
2. $g([X, Y]^{\mathrm{hor}}, Z) = g(X, [Y, Z]^{\mathrm{hor}})$ for all $X, Y, Z \in \mathfrak{g}^{\mathrm{hor}}$,

where $[X, Y]^{\mathrm{hor}} = \Omega(XYE - YXE)$. If only the first condition holds the homogeneous space is called **reductive** (but not **naturally reductive**).

We state here without proof that the Stiefel manifold (see Chapter 2.9) and the Grassmann manifold (see Chapter 2.10) are naturally reductive. We can however provide empirical evidence here:

```
B̄ = rand(SkewSymMatrix, 6) # ∈ 𝔤
Ā = exp(B̄ - StiefelLieAlgHorMatrix(B̄, 3)) # ∈ exp(𝔤ᵛᵉʳ)

X = rand(StiefelLieAlgHorMatrix, 6, 3) # ∈ 𝔤ʰᵒʳ
Y = rand(StiefelLieAlgHorMatrix, 6, 3) # ∈ 𝔤ʰᵒʳ
Z = rand(StiefelLieAlgHorMatrix, 6, 3) # ∈ 𝔤ʰᵒʳ

Ā' * X * Ā # this has to be in 𝔤ʰᵒʳ for St(3, 6) to be reductive
```

```
6×6 Matrix{Float64}:
 0.0       -0.914247  -0.387374  -0.650933  -0.741877  -0.7678
 0.914247   0.0       -0.771916  -0.454759  -0.869684  -0.80972
 0.387374   0.771916   0.0       -0.924238  -0.292897  -0.749455
 0.650933   0.454759   0.924238   0.0        0.0        0.0
 0.741877   0.869684   0.292897   0.0        0.0        0.0
 0.7678     0.80972    0.749455   0.0        0.0        0.0
```

verifies the first property and

```
adʰᵒʳ(X, Y) = StiefelLieAlgHorMatrix(X * Y - Y * X, 3)

tr(adʰᵒʳ(X, Y)' * Z) ≈ tr(X' * adʰᵒʳ(Y, Z))
```

```
true
```

verifies the second.

In `GeometricMachineLearning` we always work with elements in $\mathfrak{g}^{\mathrm{hor}}$ and the Lie group G is always $SO(N)$. We hence use:

$$\gamma_\Delta(t) = \exp(\lambda(Y)\lambda(Y)^{-1}\Omega(\Delta)\lambda(Y)\lambda(Y)^{-1})Y = \lambda(Y)\exp(\lambda(Y)^{-1}\Omega(\Delta)\lambda(Y))E.$$

Based on this we define the maps:

$$\texttt{geodesic} : \mathfrak{g}^{\mathrm{hor}} \to G, \bar{B} \mapsto \exp(\bar{B}),$$

and

$$\texttt{cayley} : \mathfrak{g}^{\mathrm{hor}} \to G, \bar{B} \mapsto \mathrm{Cayley}(\bar{B}),$$

where $\bar{B} = \lambda(Y)^{-1}\Omega(\Delta)\lambda(Y)$. These expressions for `geodesic` and `cayley` are the ones that we typically use in `GeometricMachineLearning` for computational reasons. We show how we can utilize the sparse structure of $\mathfrak{g}^{\mathrm{hor}}$ for computing the geodesic retraction and the Cayley retraction (i.e. the expressions $\exp(\bar{B})$ and $\mathrm{Cayley}(\bar{B})$ for $\bar{B} \in \mathfrak{g}^{\mathrm{hor}}$). Similar derivations can be found in [22, 66, 67].

Remark 5.2.8. Further note that, even though the global section $\lambda : \mathcal{M} \to G$ is not unique, the final geodesic $\gamma_\Delta(t) = \lambda(Y)\exp(\lambda(Y)^{-1}\Omega(\Delta)\lambda(Y))E$ does not depend on the particular section we choose.

The Geodesic Retraction

An element $\bar{B}$ of $\mathfrak{g}^{\text{hor}}$ can be written as:

$$\bar{B} = \begin{bmatrix} A & -B^T \\ B & \mathbb{O} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}A & \mathbb{I} \\ B & \mathbb{O} \end{bmatrix} \begin{bmatrix} \mathbb{I} & \mathbb{O} \\ \frac{1}{2}A & -B^T \end{bmatrix} =: B'(B'')^T,$$

where we exploit the sparse structure of the array, i.e. it is a multiplication of a $N \times 2n$ with a $2n \times N$ matrix.

We further use the following:

$$\begin{aligned} \exp(B'(B'')^T) &= \sum_{n=0}^{\infty} \frac{1}{n!}(B'(B'')^T)^n = \mathbb{I} + \sum_{n=1}^{\infty} \frac{1}{n!} B'((B'')^T B')^{n-1}(B'')^T \\ &= \mathbb{I} + B' \left(\sum_{n=1}^{\infty} \frac{1}{n!}((B'')^T B')^{n-1} \right) B'' =: \mathbb{I} + B'\mathfrak{A}(B', B'')B'', \end{aligned}$$

where we defined $\mathfrak{A}(B', B'') := \sum_{n=1}^{\infty} \frac{1}{n!}((B'')^T B')^{n-1}$. Note that evaluating $\mathfrak{A}$ relies on computing products of *small* matrices of size $2n \times 2n$. We do this by relying on a simple Taylor expansion (see the docstring for `GeometricMachineLearning.𝔄`).

The final expression we obtain is:

$$\exp(\bar{B}) = \mathbb{I} + B'\mathfrak{A}(B', B'')(B'')^T$$

The Cayley Retraction

For the Cayley retraction we leverage the decomposition of $\bar{B} = B'(B'')^T \in \mathfrak{g}^{\text{hor}}$ through the *Sherman-Morrison-Woodbury formula*:

$$(\mathbb{I} - \frac{1}{2}B'(B'')^T)^{-1} = \mathbb{I} + \frac{1}{2}B'(\mathbb{I} - \frac{1}{2}B'(B'')^T)^{-1}(B'')^T$$

So what we have to compute the inverse of:

$$\mathbb{I} - \frac{1}{2}\begin{bmatrix} \mathbb{I} & \mathbb{O} \\ \frac{1}{2}A & -B^T \end{bmatrix} \begin{bmatrix} \frac{1}{2}A & \mathbb{I} \\ B & \mathbb{O} \end{bmatrix} = \begin{bmatrix} \mathbb{I} - \frac{1}{4}A & -\frac{1}{2}\mathbb{I} \\ \frac{1}{2}B^T B - \frac{1}{8}A^2 & \mathbb{I} - \frac{1}{4}A \end{bmatrix}.$$

By leveraging the sparse structure of the matrices in $\mathfrak{g}^{\text{hor}}$ we arrive at the following expression for the Cayley retraction (similar to the case of the geodesic retraction):

$$\text{Cayley}(\bar{B}) = \mathbb{I} + \frac{1}{2}B'\left(\mathbb{I}_{2n} - \frac{1}{2}(B'')^T B'\right)^{-1}(B'')^T\left(\mathbb{I} + \frac{1}{2}\bar{B}\right),$$

where we have abbreviated $\mathbb{I} := \mathbb{I}_N$. We conclude with a remark:

Remark 5.2.9. As mentioned previously the Lie group $SO(N)$, i.e. the one corresponding to the Stiefel manifold and the Grassmann manifold, has a bi-invariant Riemannian metric associated with it: $(B_1, B_2) \mapsto \text{Tr}(B_1^T B_2)$. For other Lie groups (e.g. the symplectic group) the situation is slightly more difficult.

One of such Lie groups is the *group of symplectic matrices* [22]; for this group the expressions presented here are more complicated.

Library Functions

`GeometricMachineLearning.geodesic` – Method.

```
geodesic(B̄::StiefelLieAlgHorMatrix)
```

Compute the geodesic of an element in `StiefelLieAlgHorMatrix`.

Implementation

Internally this is using:

$$\mathbb{I} + B'\mathfrak{A}(B', B'')B'',$$

with

$$\bar{B} = \begin{bmatrix} A & -B^T \\ B & \mathbb{O} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}A & \mathbb{I} \\ B & \mathbb{O} \end{bmatrix}\begin{bmatrix} \mathbb{I} & \mathbb{O} \\ \frac{1}{2}A & -B^T \end{bmatrix} =: B'(B'')^T.$$

This is using a computationally efficient version of the matrix exponential $\mathfrak{A}$.

See `GeometricMachineLearning.𝔄`.

source

`GeometricMachineLearning.geodesic` – Method.

```
geodesic(B̄::GrassmannLieAlgHorMatrix)
```

Compute the geodesic of an element in `GrassmannLieAlgHorMatrix`.

This is equivalent to the method of `geodesic` for StiefelLieAlgHorMatrix (see Chapter 2.11).

See `geodesic(::StiefelLieAlgHorMatrix)`.

source

`GeometricMachineLearning.cayley` – Method.

```
cayley(B̄::StiefelLieAlgHorMatrix)
```

Compute the Cayley retraction of `B`.

Implementation

Internally this is using

$$\text{Cayley}(\bar{B}) = \mathbb{I} + \frac{1}{2}B'(\mathbb{I}_{2n} - \frac{1}{2}(B'')^T B')^{-1}(B'')^T(\mathbb{I} + \frac{1}{2}B),$$

with

$$\bar{B} = \begin{bmatrix} A & -B^T \\ B & \mathbb{O} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}A & \mathbb{I} \\ B & \mathbb{O} \end{bmatrix} \begin{bmatrix} \mathbb{I} & \mathbb{O} \\ \frac{1}{2}A & -B^T \end{bmatrix} =: B'(B'')^T,$$

i.e. $\bar{B}$ is expressed as a product of two $N \times 2n$ matrices.

source

`GeometricMachineLearning.cayley` – Method.

```
cayley(B̄::GrassmannLieAlgHorMatrix)
```

Compute the Cayley retraction of `B`.

This is equivalent to the method of `cayley` for StiefelLieAlgHorMatrix (see Chapter 2.11).

See `cayley(::StiefelLieAlgHorMatrix)`.

source

`GeometricMachineLearning.cayley` – Method.

```
cayley(Y::Manifold, Δ)
```

Take as input an element of a manifold `Y` and a tangent vector in `Δ` in the corresponding tangent space and compute the Cayley retraction.

In different notation: take as input an element x of $\mathcal{M}$ and an element of $T_x\mathcal{M}$ and return $\text{Cayley}(v_x)$.

Examples

```
using GeometricMachineLearning

Y = StiefelManifold([1. 0. 0.;]' |> Matrix)
Δ = [0. .5 0.;]' |> Matrix
Y₂ = cayley(Y, Δ)
```

```
Y₂' * Y₂ ≈ [1.;]

# output

true
```

See the example in [`geodesic(::Manifold{T}, ::AbstractMatrix{T}) where T`].

source

`GeometricMachineLearning.𝔄` – Method.

```
𝔄(A)
```

Compute $\mathfrak{A}(A) := \sum_{n=1}^{\infty} \frac{1}{n!}(A)^{n-1}$.

Implementation

This uses a Taylor expansion that iteratively adds terms with

```
while norm(Aⁿ) > ε
mul!(A_temp, Aⁿ, A)
Aⁿ .= A_temp
rmul!(Aⁿ, T(inv(n)))

𝔄A += Aⁿ
n += 1
end
```

until the norm of `Aⁿ` becomes smaller than machine precision. The counter `n` in the above algorithm is initialized as `2` The matrices `Aⁿ` and $\mathfrak{A}$ are initialized as the identity matrix.

source

`GeometricMachineLearning.𝔄` – Method.

```
𝔄(B̂, B̄)
```

Compute $\mathfrak{A}(B', B'') := \sum_{n=1}^{\infty} \frac{1}{n!}((B'')^T B')^{n-1}$.

This expression has the property $\mathbb{I} + B'\mathfrak{A}(B', B'')(B'')^T = \exp(B'(B'')^T)$.

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: 𝔄
import Random
Random.seed!(123)

B = rand(StiefelLieAlgHorMatrix, 10, 2)
B̂ = hcat(vcat(.5 * B.A, B.B), vcat(one(B.A), zero(B.B)))
```

```
B̃ = hcat(vcat(one(B.A), zero(B.B)), vcat(-.5 * B.A, -B.B))

one(B̂ * B̃') + B̂ * 𝔄(B̂, B̃) * B̃' ≈ exp(Matrix(B))

# output

true
```

source

5.3 Parallel Transport

The concept of *parallel transport along a geodesic* $\gamma : [0, T] \to \mathcal{M}$ describes moving a tangent vector from $T_x\mathcal{M}$ to $T_{\gamma(t)}\mathcal{M}$ such that its orientation with respect to the geodesic is preserved.

A precise definition of parallel transport needs a notion of a *connection* [21, 35, 36] and we forego it here. We simply state how to parallel transport vectors on the Lie group $SO(N)$ and the homogeneous spaces $St(n, N)$ and $Gr(n, N)$.

Theorem 5.3.1. *Given two elements $B_1^A, B_2^A \in T_A G$ the parallel transport of B_2^A along the geodesic of B_1^A is given by*

$$\Pi_{A \to \gamma_{B_1^A}(t)} B_2^A = A \exp(t \cdot A^{-1} B_1^A) A^{-1} B_2^A = A \exp(t \cdot B_1) B_2,$$

where $B_i := A^{-1} B_i^A$.

For the Stiefel manifold this is not much more complicated[1]:

Theorem 5.3.2. *Given two elements $\Delta_1, \Delta_2 \in T_Y\mathcal{M}$, the parallel transport of Δ_2 along the geodesic of Δ_1 is given by*

$$\Pi_{Y \to \gamma_{\Delta_1}(t)} \Delta_2 = \exp(t \cdot \Omega(Y, \Delta_1)) \Delta_2 = \lambda(Y) \exp(\bar{B}_1) \lambda(Y)^{-1} \Delta_2,$$

where $\bar{B}_1 = \lambda(Y)^{-1} \Omega(Y, \Delta_1) \lambda(Y)$.

We can further modify the expression of parallel transport for the Stiefel manifold:

$$\Pi_{Y \to \gamma_{\Delta_1}(t)} \Delta_2 = \lambda(Y) \exp(B_1) \lambda(Y) \Omega(Y, \Delta_2) Y = \lambda(Y) \exp(B_1) B_2 E,$$

where $B_2 = \lambda(Y)^{-1} \Omega(Y, \Delta_2) \lambda(Y)$. We can now define explicit updating rules for the global section (see Chapter 2.11) $\Lambda^{(\cdot)}$, the element of the homogeneous space $Y^{(\cdot)}$, the tangent vector $\Delta^{(\cdot)}$ and $D^{(\cdot)} = (\Lambda^{(\cdot)})^{-1} \Omega(\Delta^{(\cdot)}) \Lambda^{(\cdot)}$, its representation in $\mathfrak{g}^{\text{hor}}$.

We thus have:

[1] Here we do not provide a detailed proof that this constitutes a sound expression from the perspective of Riemannian geometry. A proof can be found in [68].

1. $\Lambda^{(t)} \leftarrow \Lambda^{(t-1)} \exp(B^{(t-1)})$,
2. $Y^{(t)} \leftarrow \Lambda^{(t)} E$,
3. $\Delta^{(t)} \leftarrow \Lambda^{(t-1)} \exp(B^{(t-1)})(\Lambda^{(t-1)})^{-1}\Delta^{(t-1)} = \Lambda^{(t)} D^{(t-1)} E$,
4. $D^{(t)} \leftarrow D^{(t-1)}$.

So we conveniently take parallel transport of vectors into account by representing them in $\mathfrak{g}^{\text{hor}}$: D does not change.

To demonstrate parallel transport we again use the example from when we introduced the concept of geodesics (see Chapter 2.7). We first set up the problem:

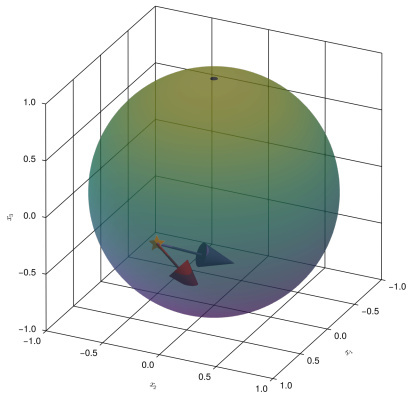


Figure 5.4: The purple vector should be transported along the geodesic of the red one.

Note that we have chosen the arrow here to have the same direction as before but only about half the magnitude. We further drew another arrow that we want to parallel transport (the purple arrow).

```
λY = GlobalSection(Y)
B = global_rep(λY, Δ)
B₂ = global_rep(λY, Δ₂)

E = StiefelProjection(3, 1)
Y_increments = []
Δ_transported = []
Δ₂_transported = []

const n_steps = 6
const timestep = 2

for _ in 1:n_steps
    update_section!(λY, timestep * B, geodesic)
    push!(Y_increments, copy(λY.Y))
    push!(Δ_transported, Matrix(λY) * B * E)
    push!(Δ₂_transported, Matrix(λY) * B₂ * E)
end
```

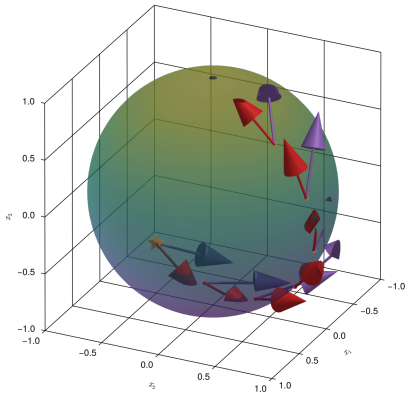


Figure 5.5: Parallel transport of the purple vector along the geodesic of the red one.

Note that the angle between the two vector is preserved as we go along the geodesic.

Chapter Summary

In this chapter we introduced our *optimizer framework* which will be used to efficiently train symplectic autoencoders and transformers with orthogonality constraints in Part IV. We proposed extending standard neural network optimizers to homogeneous spaces by introducing the extra operations `rgrad`, `global_rep` and "Retraction." The definition of a retraction we used here was slightly different from the usual one. We defined retractions as maps from the *global tangent space representation* $\mathfrak{g}^{\text{hor}}$ to the associated Lie group (and in addition satisfy two more conditions), i.e.

$$\text{Retraction} : \mathfrak{g}^{\text{hor}} \to G.$$

We further discussed what the operations `rgrad`, `global_rep` and "Retraction" look like in practice and concluded by introducing the concept of *parallel transport*. The presentation was accompanied by code snippets that demonstrate the application interface of `GeometricMachineLearning` throughout the chapter.

References

[26] B. Brantner. *Generalizing Adam To Manifolds For Efficiently Training Transformers*, arXiv preprint arXiv:2305.16901 (2023).

[68] M. Schlarb. *Covariant Derivatives on Homogeneous Spaces: Horizontal Lifts and Parallel Transport*. The Journal of Geometric Analysis **34**, 1–43 (2024).

[25] P.-A. Absil, R. Mahony and R. Sepulchre. *Optimization algorithms on matrix manifolds* (Princeton University Press, Princeton, New Jersey, 2008).

[64] B. Gao, N. T. Son, P.-A. Absil and T. Stykel. *Riemannian optimization on the symplectic Stiefel manifold*. SIAM Journal on Optimization **31**, 1546–1575 (2021).

[112] B. Gao, N. T. Son and T. Stykel. *Optimization on the symplectic Stiefel manifold: SR decomposition-based retraction and applications*. Linear Algebra and its Applications **682**, 50–85 (2024).

[21] T. Bendokat, R. Zimmermann and P.-A. Absil. *A Grassmann manifold handbook: Basic geometry and computational aspects*, arXiv preprint arXiv:2011.13699 (2020).

[22] T. Bendokat and R. Zimmermann. *The real symplectic Stiefel and Grassmann manifolds: metrics, geodesics and applications*, arXiv preprint arXiv:2108.12447 (2021).

Chapter 6

Optimizer Methods

In the previous chapter we introduced a general optimizer framework without giving explicit examples of neural network optimizers; this is done here. This chapter is divided into two sections: we first discuss *standard neural network optimizers* (including Adam) and then the more complicated BFGS optimizer. In the implementation of all these optimizers the *optimizer cache* will play an important role.

6.1 Standard Neural Network Optimizers

In this section we discuss optimization methods that are often used in training neural networks. The BFGS optimizer (see Chapter 6.2) may also be viewed as a *standard neural network optimizer* but is treated in a separate section because of its complexity. From a perspective of manifolds the *optimizer methods* outlined here operate on $\mathfrak{g}^{\mathrm{hor}}$ only. Each of them has a cache associated with it[1] and this cache is updated with the function `update!`. The precise role of this function is described below.

The Gradient Optimizer

The gradient optimizer is the simplest optimization algorithm used to train neural networks. It was already briefly discussed when we introduced Riemannian manifolds (see Chapter 2.7).

It simply does:

$$\text{weight} \leftarrow \text{weight} + (-\eta \cdot \text{gradient}),$$

where addition has to be replaced with appropriate operations in the manifold case[2].

When calling `GradientOptimizer` we can specify a learning rate η (or use the default).

```
const η = 0.01
method = GradientOptimizer(η)
```

[1] In the case of the gradient optimizer (see Chapter 6.1) this cache is trivial.

[2] In the manifold case the expression $-\eta \cdot$gradient is an element of the global tangent space (see Chapter 2.11) $\mathfrak{g}^{\mathrm{hor}}$ and a retraction maps from $\mathfrak{g}^{\mathrm{hor}}$. We then still have to compose it with the updated global section (see Chapter 5.3) $\Lambda^{(t)}$.

```
GradientOptimizer{Float64}(0.01)
```

In order to use the optimizer we need an instance of `Optimizer` that is called with the method and the weights of the neural network:

```
weight = (A = zeros(4, 4), )
o = Optimizer(method, weight)
```

```
Optimizer{GradientOptimizer{Float64}, @NamedTuple{A::GradientCache{Float64}},
↪  typeof(cayley)}(GradientOptimizer{Float64}(0.01), (A = GradientCache{Float64}(),), 0,
↪  GeometricMachineLearning.cayley)
```

If we operate on a derivative with `update!` this will compute a *final velocity* that is then used to compute a retraction (or simply perform addition if we do not deal with a manifold):

```
dx = (A = one(weight.A), )
update!(o, o.cache, dx)

dx.A
```

```
4×4 Matrix{Float64}:
 -0.01  -0.0   -0.0   -0.0
 -0.0   -0.01  -0.0   -0.0
 -0.0   -0.0   -0.01  -0.0
 -0.0   -0.0   -0.0   -0.01
```

So what has happened here is that the gradient `dx` was simply multiplied with $-\eta$ as the cache of the gradient optimizer is trivial.

The Momentum Optimizer

The momentum optimizer is similar to the gradient optimizer but further stores past information as *first moments*. We let these first moments *decay* with a *decay parameter* α:

$$\text{weights} \leftarrow \text{weights} + (\alpha \cdot \text{moment} - \eta \cdot \text{gradient}),$$

where addition has to be replaced with appropriate operations in the manifold case.

In the case of the momentum optimizer the cache is non-trivial:

```
const α = 0.5
method = MomentumOptimizer(η, α)
o = Optimizer(method, weight)

o.cache.A # the cache is stored for each array in `weight` (which is a `NamedTuple`)
```

```
`MomentumCache` that currently stores `B`as  ...
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

But as the cache is initialized with zeros it will lead to the same result as the gradient optimizer in the first iteration:

```
dx = (A = one(weight.A), )

update!(o, o.cache, dx)

dx.A
```

```
4×4 Matrix{Float64}:
 -0.01  -0.0   -0.0   -0.0
 -0.0   -0.01  -0.0   -0.0
 -0.0   -0.0   -0.01  -0.0
 -0.0   -0.0   -0.0   -0.01
```

The cache has changed however:

```
o.cache.A
```

```
`MomentumCache` that currently stores `B`as  ...
4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```

If we have weights on manifolds calling `Optimizer` will automatically allocate the correct cache on $\mathfrak{g}^{\mathrm{hor}}$:

```
weight = (Y = rand(StiefelManifold, 5, 3), )

Optimizer(method, weight).cache.Y
```

```
`MomentumCache` that currently stores `B`as  ...
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
```

```
0.0   0.0   0.0  -0.0  -0.0
0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0
```

So if the weight is $Y \in St(n, N)$ the corresponding cache is initialized as the zero element on $\mathfrak{g}^{\text{hor}} \subset \mathbb{R}^{N\times N}$ as this is the global tangent space representation corresponding to the StiefelManifold.

The Adam Optimizer

The Adam Optimizer is one of the most widely neural network optimizers. The cache of the Adam optimizer consists of *first and second moments*. The *first moments* B_1, similar to the momentum optimizer, store linear information about the current and previous gradients, and the *second moments* B_2 store quadratic information about current and previous gradients. These second moments can be interpreted as approximating the curvature of the optimization landscape.

If all the weights are on a vector space, then we directly compute updates for B_1 and B_2:

1. $B_1 \leftarrow ((\rho_1 - \rho_1^t)/(1-\rho_1^t)) \cdot B_1 + (1-\rho_1)/(1-\rho_1^t) \cdot \nabla L$,
2. $B_2 \leftarrow ((\rho_2 - \rho_1^t)/(1-\rho_2^t)) \cdot B_2 + (1-\rho_2)/(1-\rho_2^t) \cdot \nabla L \odot \nabla L$,

where $\odot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ is the *Hadamard product*: $[a \odot b]_i = a_i b_i$. ρ_1 and ρ_2 are hyperparameters. Their defaults, $\rho_1 = 0.9$ and $\rho_2 = 0.99$, are taken from [1, page 301]. After having updated the `cache` (i.e. B_1 and B_2) we compute a *velocity* with which the parameters of the network are then updated:

- $W_t \leftarrow -\eta B_1/\sqrt{B_2 + \delta}$,
- $Y^{(t+1)} \leftarrow Y^{(t)} + W^{(t)}$,

where the last addition has to be replaced with appropriate operations when dealing with manifolds. Further η is the *learning rate* and δ is a small constant that is added for stability. The division, square root and addition in the computation of W_t are performed element-wise.

In the following we show a schematic update that Adam performs for the case when no elements are on manifolds (also compare this figure with the general optimization framework (see Chapter 5.1)):

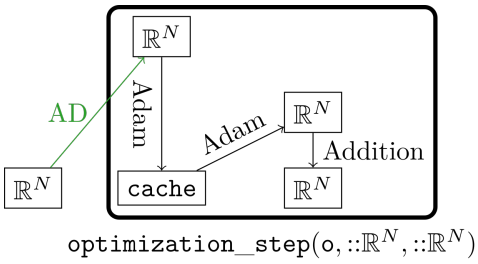


Figure 6.1: Schematic representation of the Adam optimizer. The first Adam step updates the first and second moments, and the second Adam step outputs the final velocity.

We demonstrate the Adam cache on the same example from before:

```
const ρ₁ = 0.9
const ρ₂ = 0.99
const δ = 1e-8

method = AdamOptimizer(η, ρ₁, ρ₂, δ)
o = Optimizer(method, weight)

o.cache.Y
```

```
`AdamCache` that currently stores `B₁` as ...
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
 0.0   0.0   0.0  -0.0  -0.0
 0.0   0.0   0.0   0.0   0.0
 0.0   0.0   0.0   0.0   0.0
and `B₂` as ...
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
 0.0   0.0   0.0  -0.0  -0.0
 0.0   0.0   0.0   0.0   0.0
 0.0   0.0   0.0   0.0   0.0
```

Weights on Manifolds

The problem with generalizing Adam to manifolds is that the Hadamard product $\odot$ as well as the other element-wise operations ($/$, $\sqrt{}$ and $+$) lack a clear geometric interpretation. In `GeometricMachineLearning` we get around this issue by utilizing the global tangent space representation (see Chapter 2.11). A similar approach is shown in [69].

The Adam Optimizer with Decay

The Adam optimizer with decay is similar to the standard Adam optimizer with the difference that the learning rate η decays exponentially. We start with a relatively high learning rate η_1 (e.g. 10^{-2}) and end with a low learning rate η_2 (e.g. 10^{-8}). If we want to use this optimizer we have to tell it beforehand how many epochs we train for such that it can adjust the learning rate decay accordingly:

```
const η₁ = 1e-2
const η₂ = 1e-6
const n_epochs = 1000

method = AdamOptimizerWithDecay(n_epochs, η₁, η₂, ρ₁, ρ₂, δ)
o = Optimizer(method, weight)
```

The cache is however exactly the same as for the Adam optimizer:

```
o.cache.Y
```

```
`AdamCache` that currently stores `B₁` as ...
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
 0.0   0.0   0.0  -0.0  -0.0
 0.0   0.0   0.0   0.0   0.0
 0.0   0.0   0.0   0.0   0.0
and `B₂` as ...
5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
 0.0   0.0   0.0  -0.0  -0.0
 0.0   0.0   0.0   0.0   0.0
 0.0   0.0   0.0   0.0   0.0
```

Library Functions

`GeometricMachineLearning.OptimizerMethod` – Type.

```
OptimizerMethod
```

Each `Optimizer` has to be called with an `OptimizerMethod`. This specifies how the neural network weights are updated in each optimization step.

source

`GeometricMachineLearning.GradientOptimizer` – Type.

```
GradientOptimizer(η)
```

Make an instance of a gradient optimizer.

This is the simplest neural network optimizer. It has no cache and computes the final velocity as:

$$\text{velocity} \leftarrow -\eta \nabla_{\text{weight}} L.$$

Implementation

The operations are done as memory efficiently as possible. This means the provided $\nabla_W L$ is mutated via:

```
rmul!(∇L, -method.η)
```

source

`GeometricMachineLearning.MomentumOptimizer` – Type.

```
MomentumOptimizer(η, α)
```

Make an instance of the momentum optimizer.

The momentum optimizer is similar to the `GradientOptimizer`. It however has a nontrivial cache that stores past history (see `MomentumCache`). The cache is updated via:

$$B^{\text{cache}} \leftarrow \alpha B^{\text{cache}} + \nabla_{\text{weights}} L$$

and then the final velocity is computed as

$$\text{velocity} \leftarrow -\eta B^{\text{cache}}.$$

Implementation

To save memory the *velocity* is stored in the input $\nabla_W L$. This is similar to the case of the `GradientOptimizer`.

source

`GeometricMachineLearning.AdamOptimizer` – Type.

```
AdamOptimizer(η, ρ₁, ρ₂, δ)
```

Make an instance of the Adam Optimizer.

Here the cache consists of first and second moments that are updated as

$$B_1 \leftarrow ((\rho_1 - \rho_1^t)/(1 - \rho_1^t)) \cdot B_1 + (1 - \rho_1)/(1 - \rho_1^t) \cdot \nabla L,$$

and

$$B_2 \leftarrow ((\rho_2 - \rho_1^t)/(1 - \rho_2^t)) \cdot B_2 + (1 - \rho_2)/(1 - \rho_2^t) \cdot \nabla L \odot \nabla L.$$

The final velocity is computed as:

$$\text{velocity} \leftarrow -\eta B_1 / \sqrt{B_2 + \delta}.$$

Implementation

The *velocity* is stored in the input to save memory:

```
mul!(B, -o.method.η, /ᵉˡᵉ(C.B₁, scalar_add(racᵉˡᵉ(C.B₂), o.method.δ)))
```

where `B` is the input to the [`update!`] function.

The algorithm and suggested defaults are taken from [1, page 301].

source

`GeometricMachineLearning.AdamOptimizerWithDecay` – Type.

```
AdamOptimizerWithDecay(n_epochs, η₁=1f-2, η₂=1f-6, ρ₁=9f-1, ρ₂=9.9f-1, δ=1f-8)
```

Make an instance of the Adam Optimizer with weight decay.

All except the first argument (the number of epochs) have defaults.

The difference to the standard `AdamOptimizer` is that we change the learning reate η in each step. Apart from the *time dependency* of η the two algorithms are however equivalent. $\eta(0)$ starts with a high value η_1 and then exponentially decrease until it reaches η_2 with

$$\eta(t) = \gamma^t \eta_1,$$

where $\gamma = \exp(\log(\eta_1/\eta_2)/\texttt{n\_epochs})$.

source

`GeometricMachineLearning.AbstractCache` – Type.

```
AbstractCache
```

`AbstractCache` has subtypes: `AdamCache`, `MomentumCache`, `GradientCache` and `BFGSCache`.

All of them can be initialized with providing an array (also supporting manifold types).

source

`GeometricMachineLearning.GradientCache` – Type.

```
GradientCache(Y)
```

Do not store anything.

The cache for the `GradientOptimizer` does not consider past information.

source

`GeometricMachineLearning.MomentumCache` – Type.

```
MomentumCache(Y)
```

Store the moment for `Y` (initialized as zeros).

The moment is called `B`.

If the cache is called with an instance of a `Manifold` it initializes the moments as elements of $\mathfrak{g}^{\mathrm{hor}}$ (`AbstractLieAlgHorMatrix`).

See `AdamCache`.

source

`GeometricMachineLearning.AdamCache` – Type.

```
AdamCache(Y)
```

Store the first and second moment for `Y` (initialized as zeros).

First and second moments are called `B₁` and `B₂`.

If the cache is called with an instance of a homogeneous space, e.g. the `StiefelManifold` $St(n, N)$ it initializes the moments as elements of $\mathfrak{g}^{\mathrm{hor}}$ (`StiefelLieAlgHorMatrix`).

Examples

```
using GeometricMachineLearning

Y = rand(StiefelManifold, 5, 3)
AdamCache(Y).B₁

# output

5×5 StiefelLieAlgHorMatrix{Float64, SkewSymMatrix{Float64, Vector{Float64}}, Matrix{Float64}}:
 0.0  -0.0  -0.0  -0.0  -0.0
 0.0   0.0  -0.0  -0.0  -0.0
 0.0   0.0   0.0  -0.0  -0.0
 0.0   0.0   0.0   0.0   0.0
 0.0   0.0   0.0   0.0   0.0
```

source

`AbstractNeuralNetworks.update!` – Method.

```
update!(o, cache, B)
```

Update the `cache` and output a final velocity that is stored in `B`.

Note that $B \in \mathfrak{g}^{\mathrm{hor}}$ in general.

In the manifold case the final velocity is the input to a retraction.

source

6.2 The BFGS Optimizer

The presentation shown here is largely taken from [24, chapters 3 and 6] with a derivation based on an online comment [70]. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm is a second order optimizer that can be also be used to train a neural network.

It is a version of a *quasi-Newton* method and is therefore especially suited for convex problems. As is the case with any other (quasi-)Newton method[1] the BFGS algorithm approximates the objective with a quadratic function in each optimization step:

$$m^{(k)}(x) = L(x^{(k)}) + (\nabla_{x^{(k)}} L)^T (x - x^{(k)}) + \frac{1}{2}(x - x^{(k)})^T R^{(k)} (x - x^{(k)}),$$

where $R^{(k)}$ is referred to as the *approximate Hessian*. We further require $R^{(k)}$ to be symmetric and positive definite. Differentiating the above expression and setting the derivative to zero gives us:

$$\nabla_x m^{(k)} = \nabla_{x^{(k)}} L + R^{(k)}(x - x^{(k)}) = 0,$$

or written differently:

$$x - x^{(k)} = -(R^{(k)})^{-1} \nabla_{x^{(k)}} L.$$

This value we will from now on call $p^{(k)} := -H^{(k)} \nabla_{x_k} L$ with $H^{(k)} := (R^{(k)})^{-1}$ and refer to as the *search direction*. The new iterate then is:

$$x^{(k+1)} = x^{(k)} + \eta^{(k)} p^{(k)},$$

where $\eta^{(k)}$ is the *step length*. Techniques that describe how to pick an appropriate $\eta^{(k)}$ are called *line-search methods* and are discussed below. First we discuss what requirements we impose on $R^{(k)}$. A first reasonable condition would be to require the gradient of $m^{(k)}$ to be equal to that of L at the points $x^{(k-1)}$ and $x^{(k)}$:

$$\begin{aligned} \nabla_{x^{(k)}} m^{(k)} &= \nabla_{x^{(k)}} L + R^{(k)}(x^{(k)} - x^{(k)}) \quad \stackrel{!}{=} \nabla_{x^{(k)}} L \qquad \text{and} \\ \nabla_{x^{(k-1)}} m^{(k)} &= \nabla_{x^{(k)}} L + R^{(k)}(x^{(k-1)} - x^{(k)}) \quad \stackrel{!}{=} \nabla_{x^{(k-1)}} L. \end{aligned}$$

The first one of these conditions is automatically satisfied. The second one can be rewritten as:

$$x^{(k)} - x^{(k-1)} \stackrel{!}{=} H^{(k)}(\nabla_{x^{(k)}} L - \nabla_{x^{(k-1)}} L).$$

The following notations are often used:

$$s^{(k-1)} := \eta^{(k-1)} p^{(k-1)} := x^{(k)} - x^{(k-1)} \quad \text{and} \quad y^{(k-1)} := \nabla_{x^{(k)}} L - \nabla_{x^{(k-1)}} L.$$

[1] Various Newton methods and quasi-Newton methods differ in how they model the *approximate Hessian*.

The condition mentioned above then becomes:

$$s^{(k-1)} \stackrel{!}{=} H^{(k)} y^{(k-1)},$$

and we call it the *secant equation*.

In order to pick the ideal $H^{(k)}$ we solve the following problem:

$$\begin{aligned} \min_{H} \quad & ||H - H^{(k-1)}||_W \\ \text{s.t.} \quad & H = H^T \quad \text{and} \\ \text{and} \quad & s^{(k-1)} = H y^{(k-1)}, \end{aligned}$$

where the first condition is symmetry and the second one is the secant equation. For the norm $||\cdot||_W$ we pick the weighted Frobenius norm:

$$||A||_W := ||W^{1/2} A W^{1/2}||_F,$$

where $||\cdot||_F$ is the usual Frobenius norm[2] and the matrix $W = \tilde{R}^{(k-1)}$ is the *average Hessian*:

$$\tilde{R}^{(k-1)} = \int_0^1 \nabla^2 f(x^{(k-1)} + \tau \eta^{(k-1)} p^{(k-1)}) d\tau.$$

We now state the solution to this minimization problem:

Theorem 6.2.1. *The solution of the minimization problem is:*

$$H^{(k)} = (\mathbb{I} - \frac{1}{(s^{(k-1)})^T y^{(k-1)}} s^{(k-1)} (y^{(k-1)})^T) H^{(k-1)} (\mathbb{I} - \frac{1}{(s^{k-1})^T y^{(k-1)}} y^{(k-1)} (s^{(k-1)})^T) + \frac{1}{(s^{(k-1)})^T y^{(k-1)}} s^{(k-1)} (s^{(k-1)})^T,$$

with $y^{(k-1)} = \nabla_{x^{(k)}} L - \nabla_{x^{(k-1)}} L$ *and* $s^{(k-1)} = x^{(k)} - x^{(k-1)}$ *as above.*

Proof. In order to find the ideal $H^{(k)}$ under the conditions described above, we introduce some notation:

- $\tilde{H}^{(k-1)} := W^{1/2} H^{(k-1)} W^{1/2}$,
- $\tilde{H} := W^{1/2} H W^{1/2}$,
- $\tilde{y}^{(k-1)} := W^{-1/2} y^{(k-1)}$,

[2]The Frobenius norm is $||A||_F^2 = \sum_{i,j} a_{ij}^2$.

- $\tilde{s}^{(k-1)} := W^{1/2} s^{(k-1)}$.

With this notation we can rewrite the problem of finding $H^{(k)}$ as:

$$\begin{aligned} \min_{\tilde{H}} \quad & ||\tilde{H} - \tilde{H}^{(k-1)}||_F \\ \text{s.t.} \quad & \tilde{H} = \tilde{H}^T \\ \text{and} \quad & \tilde{s}^{(k-1)} = \tilde{H}\tilde{y}^{(k-1)}. \end{aligned}$$

We further have $y^{(k-1)} = W s^{(k-1)}$ and hence $\tilde{y}^{(k-1)} = \tilde{s}^{(k-1)}$ by a corollary of the mean value theorem: $\int_0^1 g'(\xi_1 + \tau(\xi_2 - \xi_1))d\tau(\xi_2 - \xi_1) = g(\xi_2) - g(\xi_1)$ for a vector-valued function g.

Now we rewrite H and $H^{(k-1)}$ in a new basis $U = [u|u_\perp]$, where $u := \tilde{y}^{(k-1)}/||\tilde{y}^{(k-1)}||$ and $u_\perp$ is an orthogonal complement of u (i.e. we have $u^T u_\perp = 0$ and $u_\perp^T u_\perp = \mathbb{I}$):

$$\begin{aligned} U^T \tilde{H}^{(k-1)} U - U^T \tilde{H} U = \begin{bmatrix} u^T \\ u_\perp^T \end{bmatrix} (\tilde{H}^{(k-1)} - \tilde{H}) \begin{bmatrix} u & u_\perp \end{bmatrix} = \\ \begin{bmatrix} u^T \tilde{H}^{(k-1)} u - 1 & u^T \tilde{H}^{(k-1)} u_\perp \\ u_\perp^T \tilde{H}^{(k-1)} u & u_\perp^T (\tilde{H}^{(k-1)} - \tilde{H}^{(k)}) u_\perp \end{bmatrix}. \end{aligned}$$

By a property of the Frobenius norm we can consider the blocks independently:

$$\begin{aligned} ||\tilde{H}^{(k-1)} - \tilde{H}||_F^2 &= ||U^T(\tilde{H}^{(k-1)} - \tilde{H})U||_F^2 \\ &= (u^T \tilde{H}^{(k-1)} u - 1)^2 + ||u^T \tilde{H}^{(k-1)} u_\perp||_F^2 + ||u_\perp^T \tilde{H}^{(k-1)} u||_F^2 + ||u_\perp^T (\tilde{H}^{(k-1)} - \tilde{H}) u_\perp||_F^2. \end{aligned}$$

We see that $\tilde{H}$ only appears in the last term, which should therefore be made zero, i.e. the projections of $\tilde{H}_{k-1}$ and $\tilde{H}$ onto the space spanned by $u_\perp$ should coincide. With the condition $\tilde{H}u \stackrel{!}{=} u$ we hence get:

$$\tilde{H} = U \begin{bmatrix} 1 & 0 \\ 0 & u_\perp^T \tilde{H}^{(k-1)} u_\perp \end{bmatrix} U^T = uu^T + (\mathbb{I} - uu^T)\tilde{H}^{(k-1)}(\mathbb{I} - uu^T).$$

If we now map back to the original coordinate system, the ideal solution for $H^{(k)}$ is:

$$\begin{aligned} H^{(k)} = (\mathbb{I} - \frac{1}{(s^{(k-1)})^T y^{(k-1)}} s^{(k-1)} (y^{(k-1)})^T) H^{(k-1)} (\mathbb{I} - \frac{1}{(s^{k-1})^T y^{(k-1)}} y^{(k-1)} (s^{(k-1)})^T) + \\ \frac{1}{(s^{(k-1)})^T y^{(k-1)}} s^{(k-1)} (s^{(k-1)})^T, \end{aligned}$$

and the assertion is proved. □

The cache and the parameters are updated with:

1. Compute the gradient $\nabla_{x^{(k)}} L$,
2. obtain a negative search direction $p^{(k)} \leftarrow -H^{(k)} \nabla_{x^{(k)}} L$,
3. compute $s^{(k)} = \eta^{(k)} p^{(k)}$,
4. compute $y^{(k)} \leftarrow \nabla_{x^{(k)}} L - \nabla_{x^{(k-1)}} L$,
5. update $H^{(k+1)} \leftarrow (\mathbb{I} - \frac{1}{(s^{(k)})^T y^{(k)}} s^{(k)} (y^{(k)})^T) H^{(k)} (\mathbb{I} - \frac{1}{s^{(k)T} y^{(k)}} y^{(k)} (s^{(k)})^T) + \frac{1}{(s^{(k)})^T y^{(k)}} s^{(k)} (s^{(k)})^T$,
6. update $x^{(k+1)} \leftarrow x^{(k)} + s^{(k)}$.

The cache of the BFGS algorithm thus consists of the matrix $H^{(\cdot)}$ for each weight $x^{(\cdot)}$ in the neural network and the gradient for the previous time step $\nabla_{x^{(k-1)}} L$. $s^{(k)}$ here is again the *velocity* that we use to update the neural network weights.

The Riemannian Version of the BFGS Algorithm

Generalizing the BFGS algorithm to the setting of a Riemannian manifold is straightforward. All we have to do is replace Euclidean gradient by Riemannian ones (composed with a lift via `global_rep`):

$$\nabla_{x^{(k)}} L \implies (\Lambda^{(k)})^{-1} (\Omega(x^{(k)}, \text{grad}_{x^{(k)}} L)) \Lambda^{(k)} = \texttt{global\_rep}(\text{grad}_{x^{(k)}}),$$

and addition by a retraction:

$$x^{(k+1)} \leftarrow x^{(k)} + s^{(k)} \implies x^{(k+1)} \leftarrow \text{Retraction}(s^{(k)}) x^{(k)}.$$

The Hessian for the manifold BFGS algorithm is of size $\tilde{N} \times \tilde{N}$ where $\tilde{N} = \dim(\mathfrak{g}^{\text{hor}})$. For the global tangent space belonging to the Stiefel manifold (see Chapter 2.11) we have $\tilde{N} = (N-n)n + n(n-1) \div 2$.

In order to multiply the stored weights with the Hessian H we use the vectorization operation `vec` for all weights:

```
A = SkewSymMatrix([1, 2, 3], 3)
B = [4 5 6; ]
B̃ = StiefelLieAlgHorMatrix(A, B, 4, 3)
B̃ |> vec
```

```
vcat(3-element Vector{Int64}, 3-element Vector{Int64}):
 1
 2
 3
 4
 5
 6
```

The H matrix in the cache is correspondingly initialized as:

```
BFGSCache(B̄)
```

```
"`BFGSCache` that currently stores `B`as  ..."4×4 StiefelLieAlgHorMatrix{Int64,
↪  SkewSymMatrix{Int64, Vector{Int64}}, Matrix{Int64}}:
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
... and `H` as
6×6 Matrix{Int64}:
 1  0  0  0  0  0
 0  1  0  0  0  0
 0  0  1  0  0  0
 0  0  0  1  0  0
 0  0  0  0  1  0
 0  0  0  0  0  1
```

We see that $\bar{B}$ is of dimension $\tilde{N} = (N-n)n + n(n-1) \div 2 = 3 + 3 = 6$ and H is of dimension $\tilde{N} \times \tilde{N} = 6 \times 6$.

The Curvature Condition and the Wolfe Conditions

In textbooks [24] an application of the BFGS algorithm typically further involves a line search subject to the *Wolfe conditions*. If these are satisfied the *curvature condition* usually also is.

A condition that is similar to the *secant condition* discussed before is that $R^{(k)}$ has to be positive-definite at point $s^{(k-1)}$:

$$(s^{(k-1)})^T y^{(k-1)} > 0.$$

This is referred to as the *standard curvature condition*. If we impose the *Wolfe conditions*, the *standard curvature condition* holds automatically. The Wolfe conditions are stated with respect to the parameter $\eta^{(k)}$.

Definition 6.2.2. The **Wolfe conditions** are:

$$\begin{aligned} L(x^{(k)} + \eta^{(k)} p^{(k)}) &\leq L(x^{(k)}) + c_1 \eta^{(k)} (\nabla_{x^{(k)}} L)^T p^{(k)} && \text{for } c_1 \in (0,1) \quad \text{and} \\ (\nabla_{(x^{(k)} + \eta^{(k)} p^{(k)})} L)^T p^{(k)} &\geq c_2 (\nabla_{x^{(k)}} L)^T p^{(k)} && \text{for } c_2 \in (c_1, 1). \end{aligned}$$

The two Wolfe conditions above are respectively called the *sufficient decrease condition* and the *curvature condition*.

A possible choice for c_1 and c_2 are 10^{-4} and 0.9 [24]. We further have:

Theorem 6.2.3. *The second Wolfe condition, also called curvature condition, is stronger than the standard curvature condition under the assumption that the first Wolfe condition is true and* $L(x^{(k+1)}) < L(^{(x_k)})$.

Proof. We use the second Wolfe condition to write

$$(\nabla_{x^{(k)}} L)^T p^{(k-1)} - c_2 (\nabla_{x^{(k-1)}} L)^T p^{(k-1)} = (y^{(k-1)})^T p^{(k-1)} + (1 - c_2)(\nabla_{x^{(k-1)}} L)^T p^{(k-1)} \geq 0,$$

and we can apply the first Wolfe condition on the second term in this expression:

$$(1 - c_2)(\nabla_{x^{(k-1)}} L)^T p^{(k-1)} \geq \frac{1 - c_2}{c_1 \eta^{(k-1)}} (L(x^{(k)}) - L(x^{(k-1)})),$$

which is negative if the value of L is decreasing. □

It is noteworthy that line search has not been used a lot in deep learning in the past. This is beginning to change however [71, 72]. We also note that the BFGS optimizer combined with the global tangent space representation (see Chapter 2.11) offers a way of performing second order optimization on manifolds, this is however not the only way to do so [73, 74].

Stability of the Algorithm

Similar to the Adam optimizer (see Chapter 6.1) we also add a δ term for stability to two of the terms appearing in the update rule of the BFGS algorithm in practice.

Library Functions

`GeometricMachineLearning.BFGSOptimizer` – Type.

```
BFGSOptimizer(η, δ)
```

Make an instance of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimizer.

`η` is the *learning rate*. `δ` is a stabilization parameter.

source

`GeometricMachineLearning.BFGSCache` – Type.

```
BFGSCache(B)
```

Make the cache for the BFGS optimizer based on the array `B`.

It stores an array for the gradient of the previous time step `B` and the inverse of the Hessian matrix `H`.

The cache for the inverse of the Hessian is initialized with the idendity. The cache for the previous gradient information is initialized with the zero vector.

Note that the cache for `H` is changed iteratively, whereas the cache for `B` is newly assigned at every time step.

source

`AbstractNeuralNetworks.update!` – Method.

```
update!(o::Optimizer{<:BFGSOptimizer}, C, B)
```

Peform an update with the BFGS optimizer.

`C` is the cache, `B` contains the gradient information (the output of `global_rep` in general).

First we compute the *final velocity* with

```
vecS = -o.method.η * C.H * vec(B)
```

and then we update `H`

```
C.H .= (𝕀 - ρ * SY) * C.H * (𝕀 - ρ * SY') + ρ * vecS * vecS'
```

where `SY` is `vecS * Y'` and $\mathbb{I}$ is the idendity.

Implementation

For stability we use `δ` for computing `ρ`:

```
ρ = 1. / (vecS' * Y + o.method.δ)
```

This is similar to the `AdamOptimizer`

Extended help

If we have weights on a `Manifold` than the updates are slightly more difficult. In this case the `vec` operation has to be generalized to the corresponding *global tangent space*.

source

`Base.vec` – Method.

```
vec(A::StiefelLieAlgHorMatrix)
```

Vectorize `A`.

Examples

```
using GeometricMachineLearning

A = SkewSymMatrix([1, ], 2)
B = [2 3; ]
B̄ = StiefelLieAlgHorMatrix(A, B, 3, 2)
B̄ |> vec

# output

vcat(1-element Vector{Int64}, 2-element Vector{Int64}):
 1
 2
 3
```

Implementation

This is using `Vcat` from the package `LazyArrays`.

source

Chapter Summary

In this chapter we gave explicit examples of neural network optimizers and demonstrated the corresponding application interface; we referred to the related updating rules as *optimizer methods*. An important ingredient for all optimizers was the *optimizer cache*. In the simplest case (for the gradient optimizer) the cache stores nothing; it however contains *first moments* for the momentum optimizer, and *first* and *second moments* for the *Adam optimizer*. When using the more complex *BFGS optimizer* the cache is even more complicated (it stores an approximation to the inverse of the Hessian of the loss). The cache further has to be parallel transported along the optimization trajectory; for vector spaces this is trivial and for homogeneous spaces this is done by utilizing the *global tangent space representation* which was developed in the previous chapter.

References

[1] I. Goodfellow, Y. Bengio and A. Courville. *Deep learning* (MIT press, Cambridge, MA, 2016).

[24] J. N. Stephen J. Wright. *Numerical optimization* (Springer Science+Business Media, New York, NY, 2006).

[70] A.Γ. (math.stackexchange user 253273). *Quasi-newton methods: Understanding DFP updating formula*, https://math.stackexchange.com/q/2279304 (2017). Accessed on September 19, 2024.

[73] W. Huang, P.-A. Absil and K. A. Gallivan. *A Riemannian BFGS method for nonconvex optimization problems*. In: *Numerical Mathematics and Advanced Applications ENUMATH 2015* (Springer, 2016); pp. 627–634.

[74] B. Gao, N. T. Son and T. Stykel. *Symplectic Stiefel manifold: tractable metrics, second-order geometry and Newton's methods*, arXiv preprint arXiv:2406.14299 (2024).

Part III

Special Neural Network Layers and Architectures

Chapter 7

Layers

In this chapter we introduce various special neural network layers. A neural network layer is a parametrized function that is *relatively simple* and serves as a basic building block of a neural network architecture (neural network architectures will be introduced in the next chapter). Some of the neural network layers in this chapter (like SympNet layers and multihead attention) are well-established while others (like volume-preserving attention and linear symplectic attention) constitute novel work.

7.1 SympNet Layers

The *SympNet paper* [10] discusses three different kinds of sympnet layers: *activation layers*, *linear layers* and *gradient layers*. We discuss them below. Because activation layers are just a simplified form of gradient layers those are introduced together. A neural network that consists of many of these layers we call a SympNet (see Chapter 8.5).

SympNet Gradient Layer

The Sympnet gradient layer is based on the following theorem:

Theorem 7.1.1. *Given a symplectic vector space $\mathbb{R}^{2n}$ with coordinates $q_1, \dots, q_n, p_1, \dots, p_n$ and a function $f : \mathbb{R}^n \to \mathbb{R}$ that only acts on the q part, the map $(q, p) \mapsto (q, p + \nabla_q f)$ is symplectic. A similar statement holds if f only acts on the p part.*

Proof. Proofing this is straightforward by looking at the gradient of the mapping:

$$\begin{pmatrix} \mathbb{I} & \mathbb{O} \\ \nabla_q^2 f & \mathbb{I} \end{pmatrix},$$

where $\nabla_q^2 f$ is the Hessian of f. This matrix is symmetric and for any symmetric matrix A we have that:

$$\begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A & \mathbb{I} \end{pmatrix}^T \mathbb{J}_{2n} \begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A & \mathbb{I} \end{pmatrix} = \begin{pmatrix} \mathbb{I} & A \\ \mathbb{O} & \mathbb{I} \end{pmatrix} \begin{pmatrix} \mathbb{O} & \mathbb{I} \\ -\mathbb{I} & \mathbb{O} \end{pmatrix} \begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A & \mathbb{I} \end{pmatrix} = \begin{pmatrix} \mathbb{O} & \mathbb{I} \\ -\mathbb{I} & \mathbb{O} \end{pmatrix} = \mathbb{J}_{2n},$$

thus showing symplecticity. □

If we deal with `GSympNet`s this function f is

$$f(q) = a^T \Sigma(Kq + b),$$

where $a, b \in \mathbb{R}^m$, $K \in \mathbb{R}^{m \times n}$ and Σ is the antiderivative of some common activation function σ. We routinely refer to m as the *upscaling dimension* in `GeometricMachineLearning`. Computing the gradient of f gives:

$$[\nabla_q f]_k = \sum_{i=1}^{m} a_i \sigma(\sum_{j=1}^{n} k_{ij} q_j + b_i) k_{ik} = K^T (a \odot \sigma(Kq + b)),$$

where $\odot$ is the element-wise product, i.e. $[a \odot v]_k = a_k v_k$. This is the form that *gradient layers* take. In addition to gradient layers `GeometricMachineLearning` also has *linear* and *activation* layers implemented. Activation layers are simplified versions of *gradient layers*. These are equivalent to taking $m = n$ and $K = \mathbb{I}$.

SympNet Linear Layer

Linear layers of type p are of the form:

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A & \mathbb{I} \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix},$$

where A is a symmetric matrix. This is implemented very efficiently in `GeometricMachineLearning` with the special matrix `SymmetricMatrix`.

Library Functions

`GeometricMachineLearning.SympNetLayer` – Type.

```
SympNetLayer <: AbstractExplicitLayer
```

Implements the various layers from the SympNet paper [10].

This is a super type of `GradientLayer`, `ActivationLayer` and `LinearLayer`.

See the relevant docstrings of those layers for more information.

source

`GeometricMachineLearning.GradientLayer` – Type.

```
GradientLayer <: SympNetLayer
```

See the docstrings for the constructors `GradientLayerQ` and `GradientLayerP`.

source

`GeometricMachineLearning.GradientLayerQ` – Type.

```
GradientLayerQ(n, upscaling_dimension, activation)
```

Make an instance of a gradient-q layer.

The gradient layer that changes the q component. It is of the form:

$$\begin{bmatrix} \mathbb{I} & \nabla V \\ \mathbb{O} & \mathbb{I} \end{bmatrix},$$

with $V(p) = \sum_{i=1}^{M} a_i \Sigma(\sum_j k_{ij} p_j + b_i)$, where `activation` $\equiv \Sigma$ is the antiderivative of the activation function σ (one-layer neural network). We refer to M as the *upscaling dimension*.

Such layers are by construction symplectic.

source

GeometricMachineLearning.GradientLayerP – Type.

```
GradientLayerP(n, upscaling_dimension, activation)
```

Make an instance of a gradient-p layer.

The gradient layer that changes the p component. It is of the form:

$$\begin{bmatrix} \mathbb{I} & \mathbb{O} \\ \nabla V & \mathbb{I} \end{bmatrix},$$

with $V(p) = \sum_{i=1}^{M} a_i \Sigma(\sum_j k_{ij} q_j + b_i)$, where `activation` $\equiv \Sigma$ is the antiderivative of the activation function σ (one-layer neural network). We refer to M as the *upscaling dimension*.

Such layers are by construction symplectic.

source

GeometricMachineLearning.LinearLayer – Type.

```
LinearLayer <: SympNetLayer
```

See the constructors LinearLayerQ and LinearLayerP.

Implementation

LinearLayer uses the custom matrix SymmetricMatrix for its weight.

source

GeometricMachineLearning.LinearLayerQ – Type.

```
LinearLayerQ(n)
```

Make a linear layer of dimension $n \times n$ that only changes the q component.

This is equivalent to a left multiplication by the matrix:

$$\begin{pmatrix} \mathbb{I} & A \\ \mathbb{O} & \mathbb{I} \end{pmatrix},$$

where A is a `SymmetricMatrix`.

source

`GeometricMachineLearning.LinearLayerP` – Type.

```
LinearLayerP(n)
```

Make a linear layer of dimension $n \times n$ that only changes the p component.

This is equivalent to a left multiplication by the matrix:

$$\begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A & \mathbb{I} \end{pmatrix},$$

where A is a `SymmetricMatrix`.

source

`GeometricMachineLearning.ActivationLayer` – Type.

```
ActivationLayer <: SympNetLayer
```

See the constructors `ActivationLayerQ` and `ActivationLayerP`.

source

`GeometricMachineLearning.ActivationLayerQ` – Type.

```
ActivationLayerQ(n, σ)
```

Make an activation layer of size `n` and with activation `σ` that only changes the q component.

Performs:

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q + \mathrm{diag}(a)\sigma(p) \\ p \end{pmatrix}.$$

This can be recovered from `GradientLayerQ` by setting M equal to `n`, K equal to $\mathbb{I}$ and b equal to zero.

source

`GeometricMachineLearning.ActivationLayerP` – Type.

```
ActivationLayerP(n, σ)
```

Make an activation layer of size `n` and with activation `σ` that only changes the p component.

Performs:

$$\begin{pmatrix} q \\ p \end{pmatrix} \mapsto \begin{pmatrix} q \\ p + \text{diag}(a)\sigma(q) \end{pmatrix}.$$

This can be recovered from `GradientLayerP` by setting M equal to `n`, K equal to $\mathbb{I}$ and b equal to zero.

source

7.2 Volume-Preserving Feedforward Layer

The *volume-preserving feedforward layers* in `GeometricMachineLearning` are closely related to SympNet layers (see Chapter 7.1). We note that [27] proposes more complicated volume-preserving feedforward neural networks than the ones presented here. These are motivated by a classical theorem [75]; but for reasons of simplicity and their similarity to SympNet layers we resort to simpler ones.

Volume-preserving (see Chapter 3.2) feedforward layers in `GeometricMachineLearning` are a special type of ResNet layer for which we restrict the weight matrices to be of a particular form. Each layer computes:

$$\texttt{VPFF}_{A,b} : x \mapsto x + \sigma(Ax + b),$$

where σ is a nonlinearity, A is the weight and b is the bias. The matrix A is either a `LowerTriangular` matrix L or an `UpperTriangular` matrix U. We demonstrate volume-preservation of these layers by considering the case $A = L$. The matrix looks as follows:

$$L = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(n-1)} & 0 \end{pmatrix}.$$

For the jacobian we then have:

$$J = \nabla\texttt{VPFF}_{L,b} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ b_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ b_{n1} & \cdots & b_{n(n-1)} & 1 \end{pmatrix},$$

and the determinant of J is 1, i.e. the map is volume-preserving. A similar statement holds if the matrix A is `UpperTriangular` instead of `LowerTriangular`.

Library Functions

`GeometricMachineLearning.VolumePreservingFeedForwardLayer` – Type.

```
VolumePreservingFeedForwardLayer <: AbstractExplicitLayer
```

Super-type of `VolumePreservingLowerLayer` and `VolumePreservingUpperLayer`. The layers do the following:

$$x \mapsto \begin{cases} \sigma(Lx + b) & \text{where } L \text{ is } \texttt{LowerTriangular}, \\ \sigma(Ux + b) & \text{where } U \text{ is } \texttt{UpperTriangular}. \end{cases}$$

The functor can be applied to a vector, a matrix or a tensor. The special matrices are implemented as `LowerTriangular` and `UpperTriangular`.

source

`GeometricMachineLearning.VolumePreservingLowerLayer` – Type.

```
VolumePreservingLowerLayer(dim)
```

Make an instance of `VolumePreservingLowerLayer` for a specific system dimension.

See the documentation for `VolumePreservingFeedForwardLayer`.

Examples

We apply the `VolumePreservingLowerLayer` with matrix:

$$L = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

to a vector:

$$x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```
using GeometricMachineLearning

l = VolumePreservingLowerLayer(2, identity; use_bias=false)
A = LowerTriangular([1,], 2)
ps = (weight = A, )
x = ones(eltype(A), 2)

l(x, ps)

# output

2×1 Matrix{Int64}:
 1
 2
```

Arguments

The constructor can be called with the optional arguments:

- `activation=tanh`: the activation function.
- `use_bias::Bool=true` (keyword argument): specifies whether a bias should be used.

source

`GeometricMachineLearning.VolumePreservingUpperLayer` – Type.

```
VolumePreservingUpperLayer(dim)
```

Make an instance of `VolumePreservingUpperLayer` for a specific system dimension.

See the documentation for `VolumePreservingFeedForwardLayer`.

Examples

We apply the `VolumePreservingUpperLayer` with matrix:

$$U = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

to a vector:

$$x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```
using GeometricMachineLearning

l = VolumePreservingUpperLayer(2, identity; use_bias=false)
A = UpperTriangular([1,], 2)
ps = (weight = A, )
x = ones(eltype(A), 2)

l(x, ps)

# output

2×1 Matrix{Int64}:
 2
 1
```

Arguments

The constructor can be called with the optional arguments:

- `activation=tanh`: the activation function.
- `use_bias::Bool=true` (keyword argument): specifies whether a bias should be used.

source

7.3 The Attention Layer

The *attention* mechanism was originally developed for image recognition and natural language processing (NLP) tasks. It is motivated by the need to handle time series data in an efficient way[1]. Its essential idea is to compute correlations between vectors in input sequences. So given two sequences

$$(z_q^{(1)}, z_q^{(2)}, \dots, z_q^{(T)}) \text{ and } (z_k^{(1)}, z_k^{(2)}, \dots, z_k^{(T)}),$$

an attention mechanism computes pair-wise correlations between all combinations of two input vectors from these sequences. In [77] "additive" attention is used to compute such correlations:

$$(z_q, z_k) \mapsto v^T \sigma(W z_q + U z_k),$$

where $z_q, z_k \in \mathbb{R}^d$ are elements of the input sequences. The learnable parameters are $W, U \in \mathbb{R}^{n \times d}$ and $v \in \mathbb{R}^n$.

However *multiplicative attention* [9] is more straightforward to interpret and cheaper to handle computationally:

$$(z_q, z_k) \mapsto z_q^T W z_k,$$

where $W \in \mathbb{R}^{d \times d}$ is a learnable weight matrix with respect to which correlations are computed as scalar products. Regardless of the type of attention used, they all compute correlations among input sequences on whose basis further computation is performed. Given two input sequences $Z_q = (z_q^{(1)}, \dots, z_q^{(T)})$ and $Z_k = (z_k^{(1)}, \dots, z_k^{(T)})$, we can arrange the various correlations into a *correlation matrix* $C \in \mathbb{R}^{T \times T}$ with entries $[C]_{ij} = \texttt{attention}(z_q^{(i)}, z_k^{(j)})$, where the *attention* may be additive or multiplicative. In the case of multiplicative attention this matrix is just $C = Z_q^T W Z_k$.

Remark 7.3.1. The notation with designating different vectors with a q and k label comes from natural language processing. These labels stand for *queries* and *keys*.

In the section on multihead attention (see Chapter 7.4) this will be explained further.

Reweighting of the Input Sequence

In `GeometricMachineLearning` we always compute *self-attention*, meaning that the two input sequences Z_q and Z_k are the same, i.e. $Z = Z_q = Z_k$.[2]

These correlations are then used to reweight the columns in the input sequence Z. For this we first apply a nonlinearity σ onto C and then multiply $\sigma(C)$ onto Z from the right, i.e. the output of the attention layer is $Z\sigma(C)$. So we perform the following mappings:

[1] *Recurrent neural networks* [76] have the same motivation. The have however been replaced by transformers (see Chapter 8.7), of which *attention* is the most important component, in many applications.

[2] Multihead attention (see Chapter 7.4) also falls into this category. Here the input Z is multiplied from the left with several *projection matrices* P_i^Q and P_i^K, where i indicates the *head*. For each head we then compute a correlation matrix $(P_i^Q Z)^T (P^K Z)$.

$$Z \xrightarrow{\text{correlations}} C(Z) =: C \xrightarrow{\sigma} \sigma(C) \xrightarrow{\text{right multiplication}} Z\sigma(C).$$

After the right multiplication the outputs is of the following form:

$$[\sum_{i=1}^{T} p_i^{(1)} z^{(i)}, \dots, \sum_{i=1}^{T} p_i^{(T)} z^{(i)}],$$

for $p^{(i)} = [\sigma(C)]_{\bullet i}$. What is *learned* during training are T different linear combinations of the input vectors, where the coefficients $p_j^{(i)}$ in these linear combinations depend on the input Z nonlinearly.

Volume-Preserving Attention

The `VolumePreservingAttention` layer (and the activation function σ defined for it) in `GeometricMachineLearning` was specifically designed to apply it to data coming from physical systems that can be described through a divergence-free or a symplectic vector field. Traditionally the nonlinearity in the attention mechanism is a softmax[3] [9] and the self-attention layer performs the following mapping:

$$Z := [z^{(1)}, \dots, z^{(T)}] \mapsto Z\text{softmax}(Z^T W Z).$$

The softmax activation acts vector-wise, i.e. if we supply it with a matrix C as input it returns:

$$\text{softmax}(C) = [\text{softmax}(c_{\bullet 1}), \dots, \text{softmax}(c_{\bullet T})].$$

The output of a softmax is a *probability vector* (also called *stochastic vector*) and the matrix $Y = [y^{(1)}, \dots, y^{(T)}]$, where each column is a probability vector, is sometimes referred to as a "stochastic matrix" [78]. This attention mechanism finds application in *transformer neural networks* [9]. The problem with this matrix from a geometric point of view is that all the columns are independent of each other and the nonlinear transformation could in theory produce a stochastic matrix for which all columns are identical and thus lead to a loss of information. So the softmax activation function is inherently non-geometric. We visualize this with the figure below:

[3] The softmax acts on the matrix C in a vector-wise manner, i.e. it operates on each column of the input matrix $C = [\text{softmax}(c_{\bullet 1}), \dots, \text{softmax}(c_{\bullet T})] \equiv [c^{(1)}, \dots, c^{(T)}]$. The result is a sequence of probability vectors $[y^{(1)}, \dots, y^{(T)}] = [\text{softmax}(y^{(1)}), \dots, \text{softmax}(y^{(T)})]$ for which $\sum_{i=1}^{T} y_i^{(j)} = 1 \quad \forall j \in \{1, \dots, T\}$.

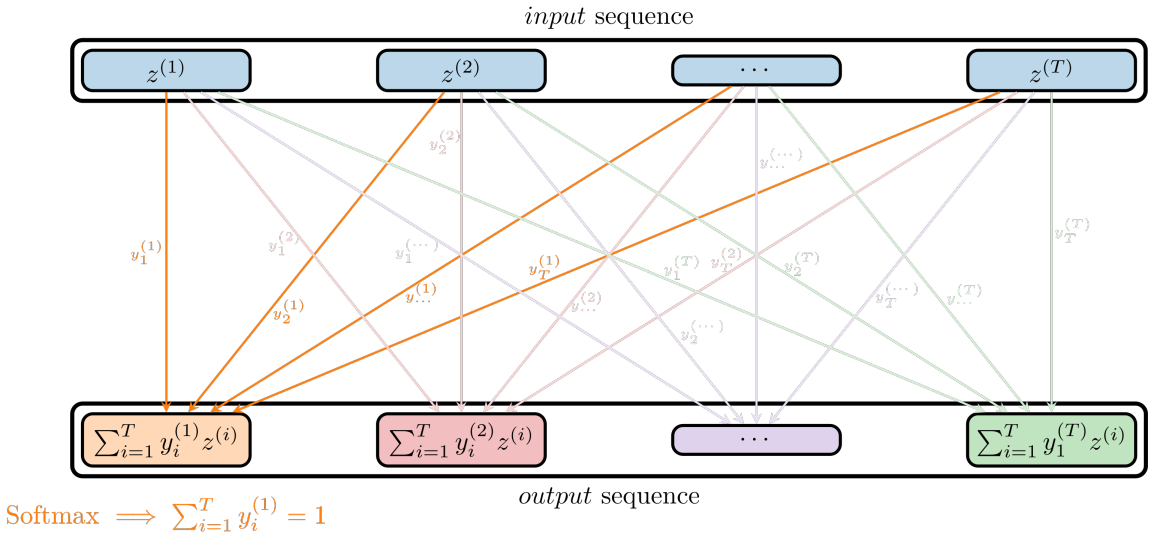


Figure 7.1: Visualization of the reweighting of the input sequence. Here the different coefficients are mostly independent of each other and could in theory produce the same reweighting for each output vector.

So the y coefficients responsible for producing the first output vector are independent from those producing the second output vector etc., they have the condition $\sum_{i=1}^{T} y_i^{(j)} z_\mu^{(i)}$ for each column j imposed on them, but the coefficients for two different columns are independent of each other.

Remark 7.3.2. We said that the coefficients are *independent of each other*, which means that in theory, they could be the same or ill-conditioned, i.e. we could have

$$\det(\text{softmax}(C)) \approx 0.$$

With *volume-preserving attention* we make the coefficients dependent of each other, such that the columns of $\sigma(C)$ are *independent* of each other:

$$\sigma(C)^T \sigma(C) = \mathbb{I},$$

which further implies $\det(\sigma(C)) = 1$.

Besides the traditional attention mechanism `GeometricMachineLearning` therefore also has a *volume-preserving transformation* that fulfills a similar role, but imposes additional structure on the y coefficients. There are two approaches implemented to realize this *volume-preserving transformation*. Both of them however utilize the *Cayley transform* to produce orthogonal matrices instead of stochastic matrices. For an orthogonal matrix Σ we have $\Sigma^T \Sigma = \mathbb{I}$, so all the columns are linearly independent, which is not necessarily true for a stochastic matrix P. In the following we explain how this new activation function is implemented. First we need to briefly discuss the *Cayley transform*.

The Cayley Transform

The Cayley transform maps from skew-symmetric matrices to orthonormal matrices. It takes the form[4]:

$$\text{Cayley} : A \mapsto (\mathbb{I} - A)(\mathbb{I} + A)^{-1}.$$

Analogously to when we used the Cayley transform as a retraction (see Chapter 5.2), we can easily check that $\text{Cayley}(A)$ is orthogonal if A is skew-symmetric. For this consider $\varepsilon \mapsto A(\varepsilon) \in \mathcal{S}_{\text{skew}}$ with $A(0) = \mathbb{O}$ and $A'(0) = B \neq \mathbb{O}$. Then we have:

$$\frac{\delta(\text{Cayley}(A)^T\text{Cayley}(A))}{\delta A} = \frac{d}{d\varepsilon}|_{\varepsilon=0}\text{Cayley}(A(\varepsilon))^T\text{Cayley}(A(\varepsilon)) = A'(0)^T + A'(0) = \mathbb{O},$$

So $\text{Cayley}(A)^T\text{Cayley}(A)$ remains unchanged among ε. In order to use the Cayley transform as an activation function we further need a mapping from the input Z to a skew-symmetric matrix. This is realized in two ways in `GeometricMachineLearning`: via a scalar-product with a skew-symmetric weighting and via a scalar-product with an arbitrary weighting.

First approach: scalar products with a skew-symmetric weighting

For this the attention layer is modified in the following way:

$$Z := [z^{(1)}, \ldots, z^{(T)}] \mapsto Z\sigma(Z^TAZ),$$

where $\sigma(C) = \text{Cayley}(C)$ and A is a matrix of type `SkewSymMatrix` that is learnable, i.e. the parameters of the attention layer are stored in A.

Second approach: scalar products with an arbitrary weighting

For this approach we compute correlations between the input vectors based on scalar product with an arbitrary weighting. This arbitrary $T \times T$ matrix A constitutes the learnable parameters of the attention layer. The correlations we consider here are based on:

$$(z^{(2)})^TAz^{(1)}, (z^{(3)})^TAz^{(1)}, \ldots, (z^{(d)})^TAz^{(1)}, (z^{(3)})^TAz^{(2)}, \ldots, (z^{(d)})^TAz^{(2)}, \ldots, (z^{(d)})^TAz^{(d-1)}.$$

So we consider correlations $(z^{(i)})^Tz^{(j)}$ for which $i > j$. We now arrange these correlations into a skew-symmetric matrix:

$$C = \begin{bmatrix} 0 & -(z^{(2)})^TAz^{(1)} & -(z^{(3)})^TAz^{(1)} & \ldots & -(z^{(d)})^TAz^{(1)} \\ (z^{(2)})^TAz^{(1)} & 0 & -(z^{(3)})^TAz^{(2)} & \ldots & -(z^{(d)})^TAz^{(2)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ (z^{(d)})^TAz^{(1)} & (z^{(d)})^TAz^{(2)} & (z^{(d)})^TAz^{(3)} & \ldots & 0 \end{bmatrix}.$$

[4]The Cayley transform here does not have the factor 1/2 hat we used when talking about the Cayley retraction (see Chapter 5.2). This is because now we do not need the retraction property $d/dt\text{Cayley}(tV)|_{t=0} = V$, but only a map $\mathfrak{g} \to G = SO(N)$.

This correlation matrix can now again be used as an input for the Cayley transform to produce an orthogonal matrix. Mathematically this is also equivalent to first computing all correlations Z^TAZ and then mapping the lower triangular to the upper triangular and negating these elements. This is visualized below:

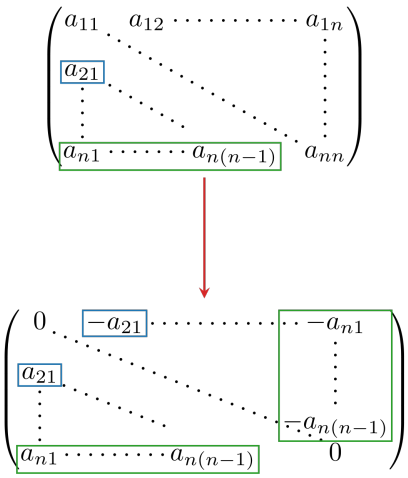


Figure 7.2: The lower-triangular part is copied to the upper-triangular part and the respective entries are negated.

Internally `GeometricMachineLearning` computes this more efficiently with the function `GeometricMachineLearning.tensor_mat_skew_sym_assign`. We show a comparison of the two approaches in the examples section (see Chapter 11.3).

How is Structure Preserved?

To discuss structure preservation, we must first define the notion of volume on the space of input matrices. A mapping $f : \mathbb{R}^{d\times T} \to \mathbb{R}^{d\times T}$ is said to be *volume-preserving* if the determinant of its Jacobian operator $Df(Z)$ is ± 1.

We focus on the first approach, where the map is defined as:

$$f(Z) = Z \cdot \sigma(Z^TAZ),$$

with $\sigma(C) = (\mathbb{I} - C)(\mathbb{I} + C)^{-1}$. Here Z is a rectangular matrix of full row rank d (with $T \geq d$).

The volume-preserving property is proved by analyzing the geometry of the tangent space $T_Z\mathbb{R}^{d\times T} = \mathbb{R}^{d\times T}$. We decompose any tangent vector into two orthogonal components: a *vertical* component and a *horizontal* component. The vertical component is the kernel of the tangent map of $\pi : \mathbb{R}^{d\times T} \to \mathfrak{so}(T), Z \mapsto Z^TAZ$ and the horizontal component is its orthogonal complement (with respect to the canonical metric).

In this context it is important to mention that the fibers defined by π, i.e. $\mathcal{F}_C := \pi^{-1}\{C\}$, are preserved under f, i.e. for $Z \in \mathcal{F}_C$ we have:

$$\pi \circ f(Z) = \sigma(C)^TZ^TAZ\sigma(C) = \sigma(C)^TC\sigma(C) = C$$

and hence $f(Z) \in \mathcal{F}_C$. Note that $\sigma(C)$ commutes with C because the former is a rational function of the latter.

We now split the proof of volume preservation into three parts: one for the fiber directions, one for the base directions and one that puts everything together. We also note that we follow [79] for this proof.

1. Vertical Space (Fiber Directions)

The vertical subspace V_Z consists of directions that do not change the correlation matrix $C = Z^T AZ$, i.e. $V_Z = T_Z \mathcal{F}_C$. A vector $v \in V_Z$ must satisfy the following:

$$V_Z = \{v \in \mathbb{R}^{d \times T} : Z^T Av + v^T AZ = 0\}.$$

We now compute the dimension of V_Z. For this consider the linear map $\mathcal{L}(v) = \text{Skew}(Z^T Av)$. The vertical space corresponds to the kernel of $\mathcal{L}$. Because the sum of the dimension of the image and the kernel give the total space, we have: $\dim(V_Z) = dT - \dim(\text{Im}(\mathcal{L}))$. Assuming that AZ has rank d we can find a transformation[5] $AZ \mapsto \begin{bmatrix}\mathbb{I} & 0\end{bmatrix}$ and then write the image under $\mathcal{L}$ as:

$$\mathcal{L}(v) = \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} \begin{bmatrix}\mathbb{I} & 0\end{bmatrix} - \begin{bmatrix}\mathbb{I} \\ 0\end{bmatrix} \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} v_1^T - v_1 & -v_2 \\ v_2^T & 0 \end{bmatrix},$$

where $v = [v_1, v_2]$ with $v_1 \in \mathbb{R}^{d \times d}$ and $v_2 \in \mathbb{R}^{d \times (T-d)}$. Dimension counting then yields $\dim(\text{Im}(\mathcal{L})) = d(d-1)/2 + d(T-d)$ and subsequently:

$$\dim(V_Z) = dT - \left(dT - \frac{d(d+1)}{2}\right) = \frac{d(d+1)}{2}.$$

Remark 7.3.3. We note the similarity in the argumentation for proving the dimensionality of the image of $\mathcal{L}$ and that of the *Stiefel manifold* $St(d, T)$.

We also note that the map f acts on a vertical vector $v \in V_Z$ simply by rotating it:

$$T_Z f(v) = v \cdot \sigma(C).$$

Since $\sigma(C)$ is an orthogonal matrix, this rotation has a determinant of 1. This is further outlined below (see Chapter 7.3).

2. Horizontal Space (Base Directions)

The horizontal subspace H_Z consists of directions orthogonal to V_Z. These directions are responsible for changing the correlation matrix C.

H_Z is generated by applying the *Lie algebra of skew-symmetric matrices* onto AZ from the right:

[5] This transformation could be absorbed into v.

$$H_Z = \{AZ\Lambda : \Lambda \in \mathbb{R}^{T\times T}, \Lambda^T = -\Lambda\}.$$

We show that H_Z is orthogonal to V_Z. For $h \in H_Z$ we have $h = AZ\Lambda$ and further

$$\mathrm{Tr}(v^T h) = \mathrm{Tr}(v^T AZ\Lambda) = \mathrm{Tr}(\Lambda^T (v^T AZ)^T) = -\mathrm{Tr}(\Lambda(v^T AZ)) = -\mathrm{Tr}((v^T AZ)\Lambda) = 0,$$

by the properties of the trace and the symmetry of $v^T AZ$. In order to prove that we have an orthogonal decomposition $\mathbb{R}^{d\times T} = V_Z \oplus H_Z$ we further have to show that the dimension of H_Z is $d(d-1)/2 + d(T-d)$.

Remark 7.3.4. We remark the analogies to the geometry of the Stiefel manifold. The horizontal component H_Z is almost equivalent to the tangent space to the Stiefel manifold $T_Y St(d,T)$ (bar a matrix transpose), where $Y \in St(d,T)$ is an orthogonal matrix that spans the same space as Z^T.

We focus on the case $T > d$, for which the map $L_Z : \Lambda \mapsto AZ\Lambda$ is not injective. Its kernel is:

$$\ker(L_Z) = \{\Lambda \in \mathfrak{so}(T) : Z\Lambda = 0\}.$$

In a basis where $Z = \begin{bmatrix}\mathbb{I}_d & 0\end{bmatrix}$, the kernel again consists of skew-symmetric matrices supported on the null space of Z (the bottom-right $(T-d)\times(T-d)$ block[6]). The dimension of the horizontal space is:

$$\dim(H_Z) = \dim(\mathfrak{so}(T)) - \dim(\mathfrak{so}(T-d)) = \frac{T(T-1)}{2} - \frac{(T-d)(T-d-1)}{2}$$

Simplifying yields $\dim(H_Z) = dT - \frac{d(d+1)}{2}$. Summing dimensions: $\dim(V_Z) + \dim(H_Z) = dT$. The decomposition is valid.

We further decompose the action of f on a horizontal vector $h = AZ\Lambda \in H_Z$ into vertical and horizontal component:

$$T_{f(Z)}\pi(T_Z f(h)) = T_Z(\pi \circ f)h = T_Z\pi(h).$$

This implies that the horizontal component of the output is generated by the *same* effective parameter Λ:

$$T_Z f(AZ\Lambda) \simeq AZ'\Lambda + \mathcal{V}_{\mathrm{ver}}$$

where $Z' = Z\sigma$. The term $\mathcal{V}_{\mathrm{ver}}$ refers to the vertical component of $T_Z f(h)$ in $V_{f(Z)}$ which is not needed for proving that f is volume-preserving.

[6]See the application of $\mathcal{L}$ onto V_Z outlined above (see Chapter 7.3).

3. The Jacobian Determinant

When represented in a coordinate system aligned with this Vertical/Horizontal decomposition, the Jacobian matrix J of f has a *block lower-triangular* structure:

$$J = \begin{pmatrix} J_{HH} & 0 \\ J_{VH} & J_{VV} \end{pmatrix}.$$

- The top-right block is 0 because moving along a vertical direction (fiber) does not change the horizontal coordinate (base).
- The bottom-right block J_{VV} represents the rotation $v \mapsto v\alpha$. Its determinant is 1.
- The top-left block J_{HH} represents the identity map on the effective parameters of the horizontal space. Its determinant is 1.

Consequently, the total determinant is $\det(Df) = \det(J_{HH}) \cdot \det(J_{VV}) = 1 \cdot 1 = 1$. This rigorous decomposition confirms that the layer preserves the volume of the input data.

Historical Note

Attention was used before the transformer (see Chapter 8.7) was introduced, but mostly in connection with *recurrent neural networks* see [77, 80].

Library Functions

`GeometricMachineLearning.tensor_mat_skew_sym_assign` – Function.

```
tensor_mat_skew_sym_assign(Z::AbstractArray{<:Number, 3}, A::AbstractMatrix)
```

Compute scalar products of columns of Z along the second axis.

The scalar products are weighted by A.

Scalar products are computed for any two vectors of the form `Z[:, i, k]` and `Z[:, j, k]`, i.e.

$$(z^{(i)}, z^{(j)}) \mapsto (z^{(i)})^T A z^{(j)} \text{ for } i > j.$$

The result of this are $n(n-2) \div 2$ scalar products for each index `k` from the third axis.

These scalar products are written into a lower-triangular matrix and the final output of the function is a tensor of these lower-triangular matrices.

This is used in `VolumePreservingAttention` when `skew_sym` is set to `false`.

Examples

Here we consider a weighting

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

and three sequences:

$$Z_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}, \quad Z_2 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad Z_3 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

The result of applying `tensor_mat_skew_sym_assign` is a tensor $\in \mathbb{R}^{2\times2\times3}$:

```
using GeometricMachineLearning: tensor_mat_skew_sym_assign

A = [1 0 0; 0 2 0; 0 0 3]
Z = [1; 0; 0;; 1; 1; 1;;; 0; 1; 0;; 1; 1; 1;;; 0; 0; 1;; 1; 1; 1]

tensor_mat_skew_sym_assign(Z, A)

# output

2×2×3 Array{Int64, 3}:
[:, :, 1] =
 0  0
 1  0

[:, :, 2] =
 0  0
 2  0

[:, :, 3] =
 0  0
 3  0
```

source

`GeometricMachineLearning.VolumePreservingAttention` – Type.

```
VolumePreservingAttention(dim, seq_length)
```

Make an instance of `VolumePreservingAttention` for a specific dimension and sequence length.

The sequence length is `0` by default.

Setting `seq_length` to `0` for all sequence lengths but does not apply the fast Cayley activation.

Arguments

The constructor can be called with an optional keyword argument:

- `skew_sym::Bool = false`: specifies if the weight matrix is skew symmetric (`true`) or arbitrary (`false`).

Functor

Applying a layer of type `VolumePreservingAttention` does the following:

- First we perform the operation $Z \mapsto Z^T AZ =: C$, where $Z \in \mathbb{R}^{N \times \texttt{seq\_length}}$ is a vector containing time series data and A is the skew symmetric matrix associated with the layer (if `skew_sym = true`).
- In a second step we compute the Cayley transform of C; $\Lambda = \text{Cayley}(C)$.
- The output of the layer is then $Z\Lambda$.

Implementation

The fast activation is only implemented for sequence lengths of 2, 3, 4 and 5. Other sequence lengths only work on CPU (for now).

The fast Cayley activation is using inverses that have been computed symbolically.

source

7.4 Multihead Attention

In order to arrive from the attention layer (see Chapter 7.3) at the *multihead attention layer* we have to do a few modifications. Here note that these neural networks were originally developed for natural language processing (NLP) tasks and the terminology used here bears some resemblance to that field. The input to a multihead attention layer typicaly comprises three components:

1. Values $V \in \mathbb{R}^{N \times T}$: a matrix whose columns are *value vectors*,
2. Queries $Q \in \mathbb{R}^{N \times T}$: a matrix whose columns are *query vectors*,
3. Keys $K \in \mathbb{R}^{N \times T}$: a matrix whose columns are *key vectors*.

Regular attention performs the following operation[1]:

$$\text{Attention}(Q, K, V) = V\text{softmax}\left(\frac{K^T Q}{\sqrt{N}}\right),$$

where N is the dimension of the vectors in V, Q and K. The softmax activation function here acts column-wise:

$$\text{softmax} : \mathbb{R}^T \to \mathbb{R}^T \text{ with } [\text{softmax}(v)]_i = e^{v_i} / \left(\sum_{j=1} e^{v_j}\right).$$

The $K^T Q$ term is a similarity matrix between the queries and the vectors.

The transformer contains a *self-attention mechanism*, i.e. takes an input X and then transforms it linearly to V, Q and K via $V = P^V X$, $Q = P^Q X$ and $K = P^K X$. What distinguishes the multihead attention layer from the singlehead attention layer is that there is not just one P^V, P^Q and P^K, but there are several: one for each *head* of the multihead attention layer. After computing the individual values, queries and vectors, and after applying the softmax, the outputs are then concatenated together in order to obtain again an array that is of the same size as the input array:

[1] The division by $\sqrt{N}$ is optional here and sometimes left out.

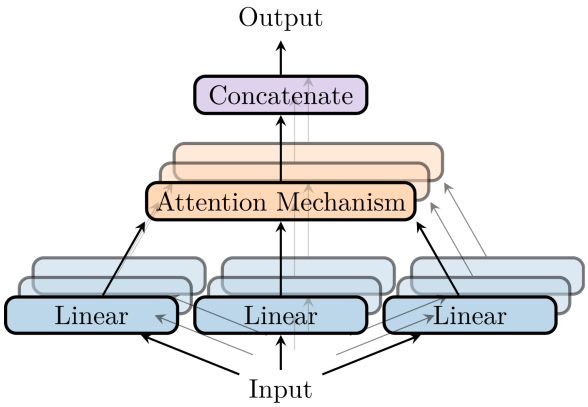


Figure 7.3: A representation of a multihead attention layer with three heads.

Written as an equation we get:

$$\text{MultiHeadAttention}(Z) = \begin{pmatrix} \text{Attention}(P_1^Q Z, P_1^K Z, P_1^V Z) \\ \text{Attention}(P_2^Q Z, P_2^K Z, P_2^V Z) \\ \cdots \\ \text{Attention}(P_{\texttt{n\_heads}}^Q Z, P_{\texttt{n\_heads}}^K Z, P_{\texttt{n\_heads}}^V Z) \end{pmatrix},$$

where $P_i^{(\cdot)} \in \mathbb{R}^{(N \div \texttt{n\_heads}) \times N}$ for $Z \in \mathbb{R}^{N \times T}$. Note that we implicitly require that N is divisible by `n_heads` here.

Here the various P matrices can be interpreted as being projections onto lower-dimensional subspaces, hence the designation by the letter P. The columns of the projection matrices span smaller spaces that should *capture features in the input data*. We will show in an example (see Chapter 11.1) how training of a neural network can benefit from putting the $P_i^{(\cdot)}$ matrices on the Stiefel manifold.

Remark 7.4.1. The `MultiHeadAttention` implemented in `GeometricMachineLearning` has an optional keyword `add_connection`. If this is set to `true` then the output of the `MultiHeadAttention` layer is:

$$\text{MultiHeadAttention}(Z) = Z + \begin{pmatrix} \text{Attention}(P_1^Q Z, P_1^K Z, P_1^V Z) \\ \text{Attention}(P_2^Q Z, P_2^K Z, P_2^V Z) \\ \cdots \\ \text{Attention}(P_{\texttt{n\_heads}}^Q Z, P_{\texttt{n\_heads}}^K Z, P_{\texttt{n\_heads}}^V Z) \end{pmatrix},$$

so we add the input again to the output.

Computing Correlations in the Multihead-Attention Layer

The attention mechanism describes a reweighting of the "values" V_i based on correlations between the "keys" K_i and the "queries" Q_i. First note the structure of these matrices: they are all a collection of T ($N \div$ `n_heads`)-dimensional vectors, i.e. $V_i = [v_i^{(1)}, \ldots, v_i^{(T)}], K_i = [k_i^{(1)}, \ldots, k_i^{(T)}], Q_i = [q_i^{(1)}, \ldots, q_i^{(T)}]$ with $i = 1, \ldots,$ `n_heads`. Those vectors have been obtained by applying the respective projection matrices onto the original input.

When performing the *reweighting* of the columns of V_i we first compute the correlations between the vectors in K_i and in Q_i and store the results in a *correlation matrix* C_i:

$$[C_i]_{mn} = \left(k_i^{(m)}\right)^T q_i^{(n)}.$$

The columns of this correlation matrix are than rescaled with a softmax function, obtaining a matrix of *probability vectors*[2] $\mathcal{P}_i$:

$$[\mathcal{P}_i]_{\bullet n} = \mathrm{softmax}\left(\frac{[C_i]_{\bullet n}}{\sqrt{N \div \texttt{n\_heads}}}\right).$$

Finally the matrix $\mathcal{P}_i$ is multiplied onto V_i from the right, resulting in T convex combinations of the T vectors $v_i^{(m)}$ with $m = 1, \dots, T$:

$$V_i\mathcal{P}_i = \left[\sum_{m=1}^{T} [\mathcal{P}_i]_{m,1} v_i^{(m)}, \dots, \sum_{m=1}^{T} [\mathcal{P}_i]_{m,T} v_i^{(m)}\right].$$

With this we can now give a better interpretation of what the projection matrices W_i^V, W_i^K and W_i^Q should do: they map the original data to lower-dimensional subspaces. We then compute correlations between the representation in the K and in the Q basis and use this correlation to perform a convex reweighting of the vectors in the V basis. These reweighted *values* are then fed into a standard feedforward neural network as is further explained in the section on the standard transformer (see Chapter 8.7).

Because the main task of the W_i^V, W_i^K and W_i^Q matrices here is for them to find bases, it makes sense to constrain them onto the Stiefel manifold; they do not and should not have the maximum possible generality.

Using a Matrix Softmax

Usually the attention layer is using a `VectorSoftmax`, i.e. one that produces a series of probability vectors. In `GeometricMachineLearning` we can also use a `MatrixSoftmax` instead. An example application of this is shown in the tutorials section (see Chapter 11.3).

Library Functions

`GeometricMachineLearning.MultiHeadAttention` – Type.

```
MultiHeadAttention(dim, n_heads)
```

Make a `MultiHeadAttention` layer with `n_heads` for a system of dimension `dim`.

Note that the `dim` has to be divisible by `n_heads`.

[2] Also called a *stochastic matrix*.

MultiHeadAttention (MHA) serves as a preprocessing step in the transformer.

It reweights the input vectors bases on correlations within those data.

This is used for the neural networks `StandardTransformerIntegrator` and `ClassificationTransformer`.

Arguments

The optional keyword arguments to `MultiHeadAttention` are:

- `Stiefel::Bool=false`
- `add_connection::Bool=true`
- `activation::AbstractSoftmax=VectorSoftmax`.

`Stiefel` indicates whether weights are put on the `StiefelManifold` $St(\dim, \dim \div \text{n\_heads})$.

`add_connection` indicates whether the input is again added to the output.

source

7.5 Linear Symplectic Attention

The attention layer introduced here can be seen as an extension of the SympNet gradient layer (see Chapter 7.1) to the setting where we deal with time series data. Before we introduce the `LinearSymplecticAttention` layer we first define a notion of symplecticity for multi-step methods (see Chapter 8.3).

This definition is different from [81, 82], but similar to the definition of volume-preservation for product spaces (see Chapter 7.3) in [30].

Definition 7.5.1. A multi-step method $\varphi \times_T \mathbb{R}^{2n} \to \times_T \mathbb{R}^{2n}$ is called **symplectic** if it preserves the the symplectic product structure, i.e. if $\hat{\varphi}$ is symplectic.

Remark 7.5.2. The **symplectic product structure** is the following skew-symmetric non-degenerate bilinear form:

$$\hat{\mathbb{J}}([z^{(1)}, \dots, z^{(T)}], [\tilde{z}^{(1)}, \dots, \tilde{z}^{(T)}]) := \sum_{i=1}^{T} (z^{(i)})^T \mathbb{J}_{2n} \tilde{z}^{(i)}.$$

$\hat{\mathbb{J}}$ is defined through the isomorphism between the product space and the space of big vectors

$$\hat{}: \times_{(T \text{ times})} \mathbb{R}^d \xrightarrow{\approx} \mathbb{R}^{dT},$$

so we induce the symplectic structure on the product space through the pullback of this isomorphism.

In order to construct a symplectic attention mechanism we extend the principle behind the SympNet gradient layer (see Chapter 7.1), i.e. we construct scalar functions that only depend on $[q^{(1)}, \dots, q^{(T)}]$ or $[p^{(1)}, \dots, p^{(T)}]$. The specific choice we make here is the following:

$$F(q^{(1)}, \dots, q^{(T)}) = \frac{1}{2} \text{Tr}(QAQ^T),$$

where $Q := [q^{(1)}, \ldots, q^{(T)}]$ is the concatenation of the vectors into a matrix. We therefore have for the gradient:

$$\nabla_Q f = \frac{1}{2} Q(A + A^T) =: Q\bar{A},$$

where $\bar{A} \in \mathcal{S}_{\text{sym}}(T)$ is a symmetric matrix. So the map performs:

$$[q^{(1)}, \ldots, q^{(T)}] \mapsto \left[\sum_{i=1}^{T} a_{1i} q^{(i)}, \ldots, \sum_{i=1}^{T} a_{Ti} q^{(i)} \right] \text{ for } a_{ji} = [\bar{A}]_{ji}.$$

Note that there is still a reweighting of the input vectors performed with this linear symplectic attention, like in standard attention (see Chapter 7.3) and volume-preserving attention (see Chapter 7.3), but the crucial difference is that the coefficients a_{ji} here are fixed and not computed as the result of a softmax or a Cayley transform (see Chapter 7.3). We hence call this attention mechanism *linear symplectic attention* as it performs a linear reweighting of the input vectors. We distinguish it from the standard attention mechanism (see Chapter 7.3), which computes coefficients that depend on the input nonlinearly.

Library Functions

`GeometricMachineLearning.LinearSymplecticAttention` – Type.

```
LinearSymplecticAttention
```

Implements the linear symplectic attention layers. Analogous to `GradientLayer` it performs mappings that only change the Q or the P part.

This layer preserves symplecticity in the *product-space sense*.

For more information see `LinearSymplecticAttentionQ` and `LinearSymplecticAttentionP`.

Implementation

The coefficients of a `LinearSymplecticAttention` layer is a `SymmetricMatrix`:

```
using GeometricMachineLearning
using GeometricMachineLearning: params

l = LinearSymplecticAttentionQ(3, 5)
ps = params(NeuralNetwork(Chain(l))).L1

typeof(ps.A) <: SymmetricMatrix

# output

true
```

source

`GeometricMachineLearning.LinearSymplecticAttentionQ` – Type.

```
LinearSymplecticAttentionQ(sys_dim, seq_length)
```

Make an instance of `LinearSymplecticAttentionQ` for a specific dimension and sequence length.

Performs:

$$\begin{pmatrix} Q \\ P \end{pmatrix} \mapsto \begin{pmatrix} Q + \nabla_P F \\ P \end{pmatrix},$$

where Q, $P \in \mathbb{R}^{n \times T}$ and $F(P) = \frac{1}{2}\mathrm{Tr}(PAP^T)$.

The parameters of this layer are $\bar{A} = \frac{1}{2}(A + A^T)$.

source

`GeometricMachineLearning.LinearSymplecticAttentionP` – Type.

```
LinearSymplecticAttentionP(sys_dim, seq_length)
```

Make an instance of `LinearSymplecticAttentionP` for a specific dimension and sequence length.

Performs:

$$\begin{pmatrix} Q \\ P \end{pmatrix} \mapsto \begin{pmatrix} Q \\ P + \nabla_Q F \end{pmatrix},$$

where Q, $P \in \mathbb{R}^{n \times T}$ and $F(Q) = \frac{1}{2}\mathrm{Tr}(QAQ^T)$.

The parameters of this layer are $\bar{A} = \frac{1}{2}(A + A^T)$.

source

Chapter Summary

In this chapter we discussed various neural network layers and the corresponding application interface in `GeometricMachineLearning`. Some of these layers constitute novel work (like the volume-preserving attention layer and the linear symplectic layer) and others were established before (such as SympNet layers and multihead attention). Volume-preserving attention and linear symplectic attention were designed as a modification of standard attention in order to imbue the corresponding neural network with structure (volume preservation and symplecticity respectively). Volume-preserving attention was achieved by exchanging the softmax activation function with a *Cayley activation function*, but otherwise keeping the operations the same, i.e. we still perform *multiplicative attention*. In order to make the attention layer symplectic we had to impose more severe restrictions on it: the attention mechanism does not compute a scalar product in this case. We therefore call it *linear*.

In the next chapter we use these neural network layers to build *neural network architectures*.

References

[10] P. Jin, Z. Zhang, A. Zhu, Y. Tang and G. E. Karniadakis. *SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems.* Neural Networks **132**, 166–179 (2020).

[77] D. Bahdanau, K. Cho and Y. Bengio. *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473 (2014).

[80] M.-T. Luong, H. Pham and C. D. Manning. *Effective approaches to attention-based neural machine translation*, arXiv preprint arXiv:1508.04025 (2015).

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. *Attention is all you need.* Advances in neural information processing systems **30** (2017).

7.6 Symplectic Attention

There are two ways of constructing symplectic attention: with a matrix-like softmax and a classical softmax.

Both of these approaches are however based on taking the derivative of an expression with respect to its input arguments (similar to gradient layers (see Chapter 7.1)).

We first start with singlehead attention[1]. We start with the correlation matrix (see Chapter 7.3):

$$C = Z^T A Z \implies C_{mn} = \sum_{k\ell} Z_{km} A_{k\ell} Z_{\ell n}.$$

Its element-wise derivative is:

$$\frac{\partial C_{mn}}{\partial Z_{ij}} = \sum_{\ell} (\delta_{jm} A_{i\ell} Z_{\ell n} + \delta_{jn} Z_{\ell m} A_{\ell i}).$$

Matrix Softmax

Here we take as a staring point the expression:

$$\Sigma(Z) = \log(1 + \exp(\sum_{m,n} C_{mn})).$$

Its gradient (with respect to Z) is:

$$\begin{aligned}\frac{\partial \Sigma(Z)}{\partial Z_{ij}} &= \frac{1}{1 + \sum m, n \exp(C_{mn})} \sum_{m'n'} \exp(C_{m'n'}) \sum_{\ell} (\delta_{jm'} A_{i\ell} Z_{\ell n'} + \delta_{jn'} Z_{\ell m'} A_{\ell i}) \\ &= \frac{1}{1 + \sum_{m,n} \exp(C_{mn})} \{[AZ \exp.(C)^T]_{ij} + [A^T Z \exp.(C)]_{ij}\}.\end{aligned}$$

[1] The multihead attention case is in principle not much more difficult, but care has to be taken with regards to the projection spaces.

Note that if `A` is a `SymmetricMatrix` the expression than simplifies to:

$$\frac{\partial \Sigma(Z)}{\partial Z_{ij}} = 2\frac{1}{1+\sum_{m,n}\exp(C_{mn})}[AZ\exp.(C)^T]_{ij},$$

or written in matrix notation:

$$\nabla_Z \Sigma(Z) = 2\frac{1}{1+\sum_{m,n}\exp(C_{mn})}AZ\exp.(C).$$

Whether we use a `SymmetricMatrix` for A or not can be set with the keyword `symmetric` in `SymplecticAttention`.

Vector Softmax

Here we take as a staring point the expression:

$$\Sigma(Z) = \sum_n \log(1+\exp(\sum_m C_{mn})).$$

We then get:

$$\frac{\partial \Sigma(Z)}{\partial Z_{ij}} = \sum_n \frac{\exp(C_{jn})}{1+\sum_m \exp(C_{mn})}[AZ]_{in} + \frac{1}{1+\sum_m \exp(C_{mj})}[A^T Z\exp(C)]_{ij}.$$

The second term in this expression is equivalent to a *standard attention step*:

$$\text{TermII}: \qquad A^T Z\text{softmax}_1(C).$$

The first term is equivalent to:

$$\text{TermI}: \qquad \sum_n [AZ]_{in}[\text{softmax}_1(C)^T]_{nj} \equiv AZ(\text{softmax}_1(C))^T.$$

If we again assume that the matrix `A` is a `SymmetricMatrix` then the expression simplifies to:

$$\nabla_Z \Sigma(Z) = AZ\text{softmax}_1(C).$$

Info

Note that we used the "one softmax" here instead of the standard softmax. The one softmax has been shown to be favorable to the standard softmax in some cases.

A discussion of the one softmax can be found in [83].

Library Functions

`GeometricMachineLearning.VectorSoftmax` – Type.

```
VectorSoftmax <: AbstractSoftmax
```

Turn an arbitrary vector into a probability vector with:

$$[\text{softmax}(a)]_i = \frac{e^{a_i}}{\sum_{i'=1}^{d} e^{a_i}}.$$

This is what is most often understood under the name "softmax". `MatrixSoftmax` is the matrix version.

source

`GeometricMachineLearning.MatrixSoftmax` – Type.

```
MatrixSoftmax
```

Like `VectorSoftmax` but for matrices:

$$[\text{softmax}(A)]_{ij} = \frac{e^{A_{ij}}}{\sum_{i'=1,j'=1}^{d,\bar{d}} e^{A_{ij}}}.$$

source

`GeometricMachineLearning.SymplecticAttention` – Type.

```
SymplecticAttention
```

Implements the symplectic attention layers. See `LinearSymplecticAttention`.

Keys

It stores the following key:

- `activation::AbstractSoftmax`

Constructors

See `SymplecticAttentionQ` and `SymplecticAttentionP`.

Implementation

`SymplecticAttention` is similar to `MultiHeadAttention` or `VolumePreservingAttention` in that it computes the scalar products of all vectors in a sequence of input vectors:

$$C = Q^T A Q,$$

where Q is the q-part of an input Z (see QPT). The matrix A is a weighting that can either be symmetric or skew-symmetric (this can be adjusted with the key-word `symmetric::Bool`).

Extended help

The symplectic attention mechanism is derived via computing the gradient of a separable Hamiltonian, as is also done in GSympNets.

source

`GeometricMachineLearning.SymplecticAttentionQ` – Type.

```
SymplecticAttentionQ
```

A constant that is derived from `SymplecticAttention`. This only changes the q-part of the input.

Constructor

```
SymplecticAttentionQ(M; symmetric::Bool, activation)
```

The default for the keywords are true and MatrixSoftmax().

You may want to alter the activation function (either `MatrixSoftmax` or `VectorSoftmax`), but its almost always better to set the keyword `symmetric` to `true`.

source

`GeometricMachineLearning.SymplecticAttentionP` – Type.

```
SymplecticAttentionP
```

A constant that is derived from `SymplecticAttention`. This only changes the `p`-part of the input.

Constructor

```
SymplecticAttentionP(M; symmetric::Bool, activation)
```

The default for the keywords are true and MatrixSoftmax().

You may want to alter the activation function (either `MatrixSoftmax` or `VectorSoftmax`), but its almost always better to set the keyword `symmetric` to `true`.

source

Chapter 8

Architectures

In this chapter we build, starting from the neural network layers introduced in the previous chapter, *neural network architectures*. Here these are always understood as a composition of neural network layers. We start by shortly explaining the application interface for neural networks used in `GeometricMachineLearning` and then introduce symplectic autoencoders, SympNets, volume-preserving feedforward neural networks, standard transformers, volume-preserving transformers and linear symplectic transformers. All of these architectures, except SympNets and standard transformers (and arguably volume-preserving feedforward neural networks), constitute novel work. As we will see all of them, except symplectic autoencoders, can be interpreted as *neural network-based integrators*.

8.1 `NeuralNetworks` in `GeometricMachineLearning`

`GeometricMachineLearning` inherits some functionality from another `Julia` package called `AbstractNeuralNetworks`. How these two packages interact is shown in the figure below for the example of the SympNet (see Chapter 8.5)[1]:

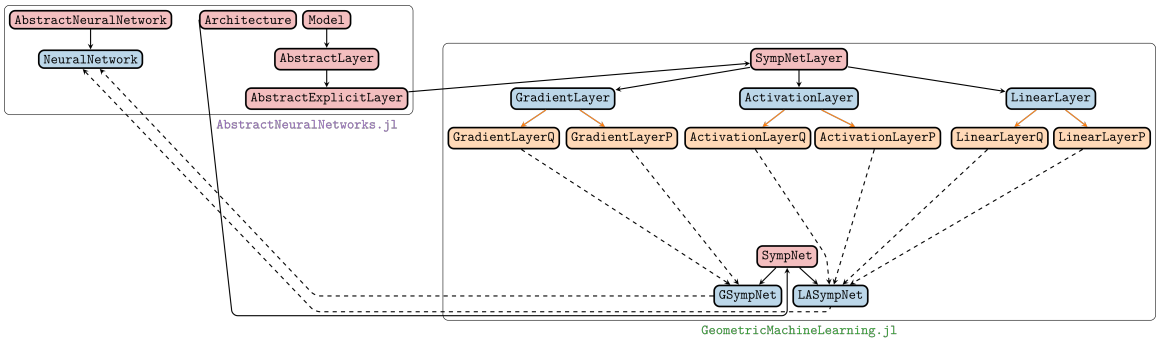


Figure 8.1: Visualization of how the packages interact.

The red color indicates an `abstract type`, blue indicates a `struct` and orange indicates a `const` (derived from a `struct`). Solid black arrows indicate direct dependencies, i.e. we have

```
GradientLayer <: SympNetLayer <: AbstractExplicitLayer
GSympNet <: SympNet <: Architecture
```

[1] The section on SympNets also contains an explanation of all the `structs` and `types` described in this section here.

Dashed black arrows indicate a derived neural network architecture. A `GSympNet` (which is an `Architecture`) is derived from `GradientLayerQ` and `GradientLayerP` (which are `AbstractExplicitLayer`s) for example. An `Architecture` can be turned into a `NeuralNetwork` by calling the associated constructor

```
arch = GSympNet(3)
nn = NeuralNetwork(arch, CPU(), Float64)
```

Such a neural network has four fields:

- `architecture`: the `Architecture` we supplied the constructor with,
- `model`: a *translation* of the supplied architecture into specific neural network layers,
- `params`: the neural network parameters,
- `backend`: this indicates on which *device* we allocate the neural network parameters. In this case it is `CPU()`.

We can get the associated model to `GSympNet` by calling:

```
nn.model.layers
```

```
(GradientLayerQ{3, 3, typeof(tanh)}(6, tanh), GradientLayerP{3, 3, typeof(tanh)}(6, tanh))
```

and we see that it consists of two layers: a `GradientLayerQ` and a `GradientLayerP`.

8.2 The Symplectic Autoencoder

Symplectic autoencoders offer a structure-preserving way of mapping a high-dimensional system to a low-dimensional system. Concretely this means that if we obtain a reduced system by means of a symplectic autoencoder, this system will again be symplectic; we can thus model a symplectic FOM with a symplectic ROM (see Chapter 4.5).

The architecture is represented by the figure below[1]:

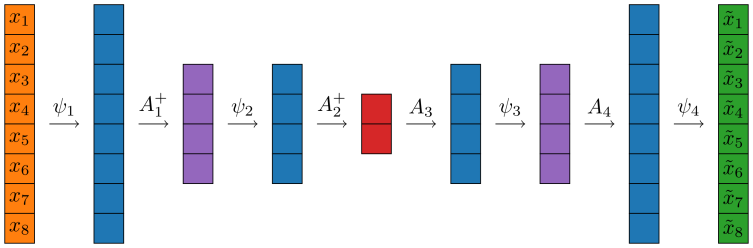


Figure 8.2: A visualization of the symplectic autoencoder architecture. It is a composition of SympNet layers and PSD-like layers.

[1]For the symplectic autoencoder we only use SympNet gradient layers (see Chapter 7.1) because they seem to outperform LA-SympNets in many cases and are easier to interpret: their nonlinear part is the gradient of a function that only depends on half the coordinates.

It is a composition of SympNet gradient layers (see Chapter 7.1) and PSD-like matrices (see Chapter 4.5), so a matrix A_i (respectively A_i^+) is of the form

$$A_i^{(+)} = \begin{bmatrix} \Phi_i & \mathbb{O} \\ \mathbb{O} & \Phi_i \end{bmatrix} \text{ where } \begin{cases} \Phi_i \in St(d_i, d_{i+1}) \subset \mathbb{R}^{d_{i+1} \times d_i} & \text{if } d_{i+1} > d_i \\ \Phi_i \in St(d_{i+1}, d_i) \subset \mathbb{R}^{di \times d_{i+1}} & \text{if } d_i > d_{i+1}, \end{cases}$$

where $A_i^{(+)} = A_i$ if $d_{i+1} > d_i$ and $A_i^{(+)} = A_i^+$ if $d_{i+1} < d_i$. Also note that for cotangent lift-like matrices we have

$$\begin{aligned} A_i^+ = \mathbb{J}_{2N} A_i^T \mathbb{J}_{2n}^T &= \begin{bmatrix} \mathbb{O}_{n\times n} & \mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{O}_{n\times n} \end{bmatrix} \begin{bmatrix} \Phi_i^T & \mathbb{O}_{n\times N} \\ \mathbb{O}_{n\times N} & \Phi_i^T \end{bmatrix} \begin{bmatrix} \mathbb{O}_{N\times N} & -\mathbb{I}_N \\ \mathbb{I}_N & \mathbb{O}_{N\times N} \end{bmatrix} \\ &= \begin{bmatrix} \Phi_i^T & \mathbb{O}_{n\times N} \\ \mathbb{O}_{n\times N} & \Phi_i^T \end{bmatrix} = A_i^T, \end{aligned}$$

so the symplectic inverse is equivalent to a matrix transpose in this case. In the symplectic autoencoder we use SympNets as a form of *symplectic preprocessing* before the linear symplectic reduction (i.e. the PSD layer) is employed. The resulting neural network has some of its weights on manifolds, which is why we cannot use standard neural network optimizers, but have to resort to manifold optimizers (see Chapter 5.1). Note that manifold optimization is not necessary for the weights corresponding to the SympNet layers, these are still updated with standard neural network optimizers during training. Also note that SympNets are nonlinear and preserve symplecticity, but they cannot change the dimension of a system while PSD layers can change the dimension of a system and preserve symplecticity, but are strictly linear. Symplectic autoencoders have all three properties: they preserve symplecticity, can change dimension and are nonlinear mappings. We can visualize this in a Venn diagram:

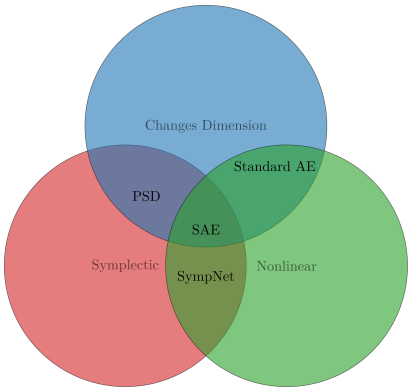


Figure 8.3: Venn diagram visualizing that a symplectic autoencoder (SAE) is symplectic, can change dimension and is nonlinear.

We now show the proof that shows $\nabla_{\mathcal{R}(z)}\psi = (\nabla_z \mathcal{R})^+$ which was used when showing the equivalence between Hamiltonian systems on the full and the reduced space (see Chapter 4.5):

Proof. The symplectic autoencoder is a composition of G-SympNet layers and PSD-like matrices:

$$\Psi^d = A_n \circ \psi_n \circ \cdots \circ A_1 \circ \psi_1.$$

It's local inverse is

$$(\Psi^d)^{-1} = \psi_1^{-1} \circ A_1^+ \circ \ldots \circ \psi_n^{-1} \circ A_n^+.$$

The jacobian of Ψ^d is:

$$\nabla_z \Psi^d = A_n \nabla_{A_{n-1} \cdots A_1 \psi_1(z)} \psi_n \cdots A_1 \nabla_z \psi_1,$$

and thus

$$(\nabla_z \Psi^d)^+ = (\nabla \psi)^+ A_1^+ \cdots (\nabla \psi_n)^+ A_n^+,$$

where we dropped the argument in the derivative of the nonlinear parts. We further have

$$A^+ = A^T$$

for PSD-like matrices and

$$(\nabla_z \psi)^+ = \begin{pmatrix} \mathbb{O} & \mathbb{I} \\ -\mathbb{I} & \mathbb{O} \end{pmatrix} \begin{pmatrix} \mathbb{I} & \nabla_p f \\ \mathbb{O} & \mathbb{I} \end{pmatrix}^T \begin{pmatrix} \mathbb{O} & -\mathbb{I} \\ \mathbb{I} & \mathbb{O} \end{pmatrix} = \begin{pmatrix} \mathbb{I} & -\nabla_p f \\ \mathbb{O} & \mathbb{I} \end{pmatrix},$$

for the G-SympNet layers, where we assumed that ψ only changes the q component. Because these matrices are square, the inverse $(\nabla \psi)^+ = (\nabla \psi)^{-1}$ is unique. □

The SympNet layers in the symplectic autoencoder operate in *intermediate dimensions* (as well as the input and output dimensions). In the following we explain how `GeometricMachineLearning` computes those intermediate dimensions.

Intermediate Dimensions

For a high-fidelity system of dimension $2N$ and a reduced system of dimension $2n$, the intermediate dimensions in the symplectic encoder and the decoder are computed according to:

```
iterations = Vector{Int}(n : (N - n) ÷ (n_blocks - 1) : N)
iterations[end] = full_dim2
iterations * 2
```

So for e.g. $2N = 100$, $2n = 10$ and `n_blocks` $= 3$ we get

$$\text{iterations} = 5{:}(45 \div 2){:}50 = 5{:}22{:}50 = (5, 27, 49).$$

We still have to perform the two other modifications in the algorithm above:

1. `iterations[end] = full_dim2` ... assign `full_dim2` to the last entry,
2. `iterations * 2` ... multiply all the intermediate dimensions by two.

The resulting dimensions are:

$$(10, 54, 100).$$

The second step (the multiplication by two) is needed to arrive at intermediate dimensions that are even. This is necessary to preserve the canonical symplectic structure of the system (see Chapter 3.1).

Example

A visualization of an instance of `SymplecticAutoencoder` is shown below:

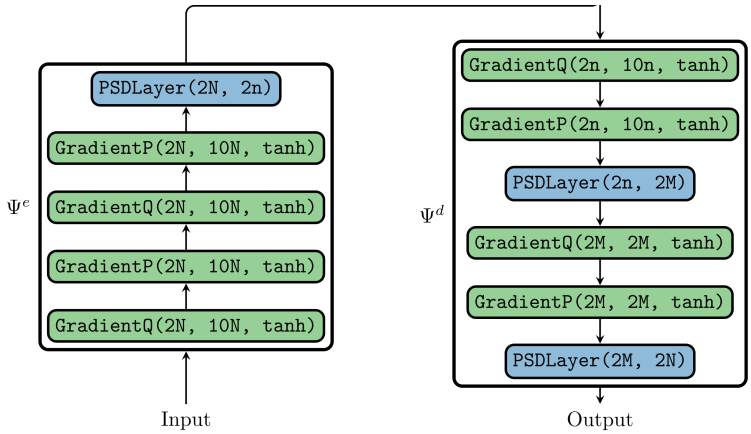


Figure 8.4: Example of a symplectic autoencoder. The SympNet layers are in green, the PSD-like layers are in blue.

In this figure we have the following configuration: `n_encoder_blocks` is two, `n_encoder_layers` is four, `n_decoder_blocks` is three and `n_decoder_layers` is two. For a full dimension of 100 and a reduced dimension of ten we can build such an instance of a symplectic autoencoder by calling:

```
using GeometricMachineLearning

const full_dim = 100
const reduced_dim = 10
```

```
model = SymplecticAutoencoder(full_dim, reduced_dim;
                                                    n_encoder_blocks = 2,
                                                    n_encoder_layers = 4,
                                                    n_decoder_blocks = 3,
                                                    n_decoder_layers = 2)

for layer in Chain(model)
    println(stdout, layer)
end
```

```
GradientLayerQ{100, 100, typeof(tanh)}(500, tanh)
GradientLayerP{100, 100, typeof(tanh)}(500, tanh)
GradientLayerQ{100, 100, typeof(tanh)}(500, tanh)
GradientLayerP{100, 100, typeof(tanh)}(500, tanh)
PSDLayer{100, 10}()
GradientLayerQ{10, 10, typeof(tanh)}(50, tanh)
GradientLayerP{10, 10, typeof(tanh)}(50, tanh)
PSDLayer{10, 54}()
GradientLayerQ{54, 54, typeof(tanh)}(270, tanh)
GradientLayerP{54, 54, typeof(tanh)}(270, tanh)
PSDLayer{54, 100}()
```

We also see that the intermediate dimension in the decoder is `54` for the specified dimensions and `n_decoder_blocks = 3` as was outlined before.

Library Functions

`GeometricMachineLearning.SymplecticAutoencoder` – Type.

```
SymplecticAutoencoder(full_dim, reduced_dim)
```

Make an instance of `SymplecticAutoencoder` for dimensions `full_dim` and `reduced_dim`.

The architecture

The symplectic autoencoder architecture was introduced in [33]. Like any other autoencoder it consists of an *encoder* $\Psi^e : \mathbb{R}^{2N} \to \mathbb{R}^{2n}$ and a *decoder* $\Psi^d : \mathbb{R}^{2n} \to \mathbb{R}^{2N}$ with $n \ll N$. These satisfy the following properties:

$$\nabla_z \Psi^e \mathbb{J}_{2N} (\nabla_z \Psi^e \mathbb{J}_{2N})^T = \mathbb{J}_{2n} \quad \text{and} \\ (\nabla_\xi \Psi^d)^T \mathbb{J}_{2N} \nabla_\xi \Psi^d = \mathbb{J}_{2n}.$$

Because the decoder has this particular property, the reduced system can be described by the Hamiltonian $H \circ \Psi^d$:

$$\mathbb{J}_{2n} \nabla_\xi (H \circ \Psi^d) = \mathbb{J}_{2n} (\nabla_\xi \Psi^d)^T \nabla_{\Psi^d(\xi)} H = \mathbb{J}_{2n} (\nabla_\xi \Psi^d)^T \mathbb{J}_{2N}^T \mathbb{J}_{2N} \nabla_{\Psi^d(\xi)} H = (\nabla_\xi \Psi^d)^+ X_H(\Psi^d(\xi)),$$

where $(\nabla_\xi \Psi^d)^+$ is the *symplectic inverse* of $\nabla_\xi \Psi^d$ (for more details see the docs on the `AutoEncoder` type).

Arguments

Besides the required arguments `full_dim` and `reduced_dim` you can provide the following keyword arguments:

- `n_encoder_layers::Integer = 4`: The number of layers in one encoder block.
- `n_encoder_blocks::Integer = 2`: The number of encoder blocks.
- `n_decoder_layers::Integer = 1`: The number of layers in one decoder block.
- `n_decoder_blocks::Integer = 3`: The number of decoder blocks.
- `sympnet_upscale::Integer = 5`: The *upscaling dimension* of the GSympNet. See `GradientLayerQ` and `GradientLayerP`.
- `activation = tanh`: The activation in the gradient layers.
- `encoder_init_q::Bool = true`: Specifies if the first layer in each encoder block should be of q type.
- `decoder_init_q::Bool = true`: Specifies if the first layer in each decoder block should be of p type.

source

8.3 Neural Network Integrators

In `GeometricMachineLearning` we can divide most neural network architectures (that are used for applications to physical systems) into two categories: autoencoders and integrators. This is also closely related to the application of reduced order modeling where *autoencoders are used in the offline phase* and *integrators are used in the online phase*.

The term *integrator* in its most general form refers to an approximation of the flow of an ODE (see Chapter 2.6) by a numerical scheme. Traditionally, for so called *one-step methods*, these numerical schemes are constructed by defining certain relationships between a known time step $z^{(t)}$ and a future unknown one $z^{(t+1)}$ [3, 84]:

$$f(z^{(t)}, z^{(t+1)}) = 0.$$

One usually refers to such a relationship as an *integration scheme*. If this relationship can be reformulated as

$$z^{(t+1)} = g(z^{(t)}),$$

then we refer to the scheme as *explicit*, if it cannot be reformulated in such a way then we refer to it as *implicit*. Implicit schemes are typically more expensive to solve than explicit ones. The `Julia` library `GeometricIntegrators` [44] offers a wide variety of integration schemes both implicit and explicit.

The neural network integrators in `GeometricMachineLearning` (the corresponding type is `NeuralNetworkIntegrator`) are all explicit integration schemes where the function g above is modeled with a neural network.

Neural networks, as an alternative to traditional methods, are employed because of (i) potentially superior performance and (ii) an ability to learn unknown dynamics from data.

The simplest of such a neural network for modeling an explicit integrator is the `ResNet`. SympNets (see Chapter 8.5) can be seen as the symplectic (see Chapter 3.1) version of the ResNet. There is an example demonstrating the performance of SympNets (see Chapter 10.1). This example demonstrates the advantages of symplectic neural networks.

Multi-step methods

Multi-step method [81, 82] refers to schemes that are of the form[1]:

$$f(z^{(t-\texttt{sl}+1)}, z^{(t-\texttt{sl}+2)}, \ldots, z^{(t)}, z^{(t+1)}, \ldots, z^{(\texttt{pw}+1)}) = 0,$$

where `sl` is short for *sequence length* and `pw` is short for *prediction window*. Note that we can recover traditional one-step methods by setting `sl` and `pw` equal to 1. We can also formulate explicit mulit-step methods. They are of the form:

$$[z^{(t+1)}, \ldots, z^{(t+\texttt{pw})}] = g(z^{(t-\texttt{sl}+1)}, \ldots, z^{(t)}).$$

In `GeometricMachineLearning` all multi-step methods, as is the case with one-step methods, are explicit. There are essentially two ways to construct multi-step methods with neural networks: the older one is using recurrent neural networks such as long short-term memory cells (LSTMs) [85] and the newer one is using transformer neural networks [9]. Both of these approaches have been successfully employed to learn multi-step methods (see [54, 55] for the former and [30, 86, 87] for the latter), but because the transformer architecture exhibits superior performance on modern hardware and can be imbued with geometric properties we almost always use a transformer-derived architecture when dealing with time series[2].

Explicit multi-step methods derived from the transformer are always subtypes of the type `TransformerIntegrator` in `GeometricMachineLearning`. In `GeometricMachineLearning` the standard transformer (see Chapter 8.7), the volume-preserving transformer (see Chapter 8.8) and the linear symplectic transformer (see Chapter 8.9) are implemented.

Remark 8.3.1. For standard multi-step methods (that are not neural network-based) `sl` is generally a number greater than one whereas `pw = 1` in most cases. For the `TransformerIntegrator`s in `GeometricMachineLearning` however we usually have:

$$\texttt{pw} = \texttt{sl},$$

so the number of vectors in the input sequence is equal to the number of vectors in the output sequence. This makes it easier to define structure-preservation for these architectures and improves stability.

[1] We again assume that all the steps up to and including t are known.

[2] `GeometricMachineLearning` also has an LSTM implementation, but this may be deprecated in the future.

Library Functions

`GeometricMachineLearning.NeuralNetworkIntegrator` – Type.

`NeuralNetworkIntegrator` is a super type of various neural network architectures such as `SympNet` and `ResNet`.

The purpose of such neural networks is to approximate the flow of an ordinary differential equation (ODE).

`NeuralNetworkIntegrator`s can be seen as modeling traditional one-step methods with neural networks, i.e. for a fixed time step they perform:

$$\texttt{NeuralNetworkIntegrator} : z^{(t)} \mapsto z^{(t+1)},$$

to try to approximate the flow of some ODE:

$$||\texttt{Integrator}(z^{(t)}) - \varphi^h(z^{(t)})|| \approx \mathcal{O}(h),$$

where φ^h is the flow map of the ODE for a time step h.

source

`GeometricMachineLearning.ResNet` – Type.

```
ResNet(dim, n_blocks, activation)
```

Make an instance of a `ResNet`.

A ResNet is a neural network that realizes a mapping of the form:

$$x = \mathcal{NN}(x) + x,$$

so the input is again added to the output (a so-called add connection). In `GeometricMachineLearning` the specific ResNet that we use consists of a series of simple `ResNetLayer`s.

Constructor

`ResNet` can also be called with the constructor:

```
ResNet(dl, n_blocks)
```

where `dl` is an instance of `DataLoader`.

See `iterate` for an example of this.

source

`GeometricMachineLearning.ResNetLayer` – Type.

```
ResNetLayer(dim)
```

Make an instance of the resnet layer.

The `ResNetLayer` is a simple feedforward neural network to which we add the input after applying it, i.e. it realizes $x \mapsto x + \sigma(Ax + b)$.

Arguments

The ResNet layer takes the following arguments:

1. `dim::Integer`: the system dimension.
2. `activation = identity`: The activation function.

The following is a keyword argument:

- `use_bias::Bool = true`: This determines whether a bias b is used.

source

`Base.iterate` – Method.

```
iterate(nn, ics)
```

This function computes a trajectory for a `NeuralNetworkIntegrator` that has already been trained for valuation purposes.

It takes as input:

1. `nn`: a `NeuralNetwork` (that has been trained).
2. `ics`: initial conditions (a `NamedTuple` of two vectors)

Examples

To demonstrate `iterate` we use a simple ResNet that does:

$$\text{ResNet} : x \mapsto \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

and we iterate three times with

$$\texttt{ics} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

```
using GeometricMachineLearning

model = ResNet(3, 0, identity)
weight = [1 0 0; 0 2 0; 0 0 1]
bias = [0, 0, 1]
ps = NeuralNetworkParameters((L1 = (weight = weight, bias = bias), ))
nn = NeuralNetwork(model, Chain(model), ps, CPU())

ics = [1, 1, 1]
iterate(nn, ics; n_points = 4)

# output

3×4 Matrix{Int64}:
 1  2  4   8
 1  3  9  27
 1  3  7  15
```

Arguments

The optional keyword argument is

- `n_points = 100`

The number of integration steps that should be performed.

source

`GeometricMachineLearning.TransformerIntegrator` – Type.

```
TransformerIntegrator <: Architecture
```

Encompasses various transformer architectures, such as the `VolumePreservingTransformer` and the `LinearSymplecticTransformer`.

The central idea behind this is to construct an explicit multi-step integrator:

$$\texttt{Integrator} : [z^{(t-\texttt{sl}+1)}, z^{(t-\texttt{sl}+2)}, \ldots, z^{(t)}] \mapsto [z^{(t+1)}, z^{(t+2)}, \ldots, z^{(t+\texttt{pw})}],$$

where `sl` stands for *sequence length* and `pw` stands for *prediction window*, so the numbers of input and output vectors respectively.

source

`Base.iterate` – Method.

```
iterate(nn, ics)
```

Iterate the neural network of type `TransformerIntegrator` for initial conditions `ics`.

The initial condition is a matrix $\in \mathbb{R}^{n \times \texttt{seq\_length}}$ or `NamedTuple` of two matrices).

This function computes a trajectory for a Transformer that has already been trained for valuation purposes.

Parameters

The following are optional keyword arguments:

- `n_points::Int=100`: The number of time steps for which we run the prediction.
- `prediction_window::Int=size(ics.q, 2)`: The prediction window (i.e. the number of steps we predict into the future) is equal to the sequence length (i.e. the number of input time steps) by default.

source

8.4 Hamiltonian Neural Network

The Hamiltonian Neural Network (HNN) [49] aims at building a Hamiltonian vector field (see Chapter 3.1) with a neural network. We recall that a *canonical Hamiltonian vector field* on $\mathbb{R}^{2d}$ is one that can be written as:

$$X_H(z) = \mathbb{J}_{2d} \nabla_z H,$$

where $\mathbb{J}_{2d}$ is the `PoissonTensor`. The idea behind a Hamiltonian neural network is to learn a vector field of this form, i.e. to learn:

$$X_{\mathcal{NN}}(z) = \mathbb{J}_{2d} \nabla_z \mathcal{NN},$$

where $\mathcal{NN} : \mathbb{R}^{2d} \to \mathbb{R}$ is a neural network that approximates the Hamiltonian. There are then two different options to define a *HNN loss*, depending on the format in which the data are given.

HNN Loss for Vector Field Data

For the first loss, we assume that the given data describe the vector field of the HNN:

$$\mathcal{L}_{\text{HNN}} = \sqrt{\sum_{i=1}^{d} \left(\left|\left| \frac{\partial \mathcal{NN}}{\partial q_i} + \dot{p}_i \right|\right|_2^2 + \left|\left| \frac{\partial \mathcal{NN}}{\partial p_i} - \dot{q}_i \right|\right|_2^2 \right)}$$

HNN Loss for Phase Space Data

For the second loss, we assume that the given data describe points in phase space associated to a Hamiltonian system. For this approach we also need to specify a *symplectic integrator* [3] in order to train the neural network. In the following we use SymplecticEulerB to define this loss. This integrator does the following:

$$\text{SymplecticEulerB} : (q^{(t)}, p^{(t)}) \mapsto (q^{(t+1)}, p^{(t+1)})$$

Note that this integrator is implicit in general.

$$\mathcal{L}_{\text{HNN}} = \sqrt{\sum_t \sum_{i=1}^{d} \left|\left| \frac{\partial \mathcal{NN}}{\partial q_i}(q^{(t)}, p^{(t+1)}) + \frac{p_i^{t+1} - p_i^{(t)}}{\Delta t} \right|\right|_2^2 + \left|\left| \frac{\partial \mathcal{NN}}{\partial p_i}(q^{(t)}, p^{(t+1)}) + \frac{q_i^{t+1} - q_i^{(t)}}{\Delta t} \right|\right|_2^2}$$

Here the derivatives (i.e. vector field data) $\dot{q}_i^{(t)}$ and $\dot{p}_i^{(t)}$ are approximated with finite differences:

Info

Usually we use `Zygote` for computing derivatives in `GeometricMachineLearning`, but as the `Zygote` documentation itself points out: "Often using a different AD system over Zygote is a better solution [for computing second-order derivatives]." For this reason we compute the loss of the HNN with `SymbolicNeuralNetworks` and optionally also its gradient.

Library Functions

`GeometricMachineLearning.hamiltonian_vector_field` – Method.

```
hamiltonian_vector_field(arch::HamiltonianArchitecture)
```

Compute an executable expression of the Hamiltonian vector field of a `HamiltonianArchitecture`.

Implementation

This first computes a symbolic expression of the vector field using `symbolic_hamiltonian_vector_field`.

source

`GeometricMachineLearning.HamiltonianArchitecture` – Type.

```
HamiltonianArchitecture <: Architecture
```

See `StandardHamiltonianArchitecture` and `GeneralizedHamiltonianArchitecture`.

source

`GeometricMachineLearning.StandardHamiltonianArchitecture` – Type.

```
StandardHamiltonianArchitecture <: HamiltonianArchitecture
```

A realization of the standard Hamiltonian neural network (HNN) [49].

Also see `GeneralizedHamiltonianArchitecture`.

Constructor

The constructor takes the following input arguments:

1. `dim`: system dimension,
2. `width = dim`: width of the hidden layer. By default this is equal to `dim`,
3. `nhidden = 1`: the number of hidden layers,
4. `activation = AbstractNeuralNetworks.TanhActivation()`: the activation function used in the HNN.

source

`GeometricMachineLearning.HNNLoss` – Type.

```
HNNLoss <: NetworkLoss
```

The loss for a Hamiltonian neural network.

Constructor

This can be called with a `NeuralNetwork`, built with a `HamiltonianArchitecture`, as the only input arguemtn, i.e.:

```
HNNLoss(nn)
```

where `nn` is a `NeuralNetwork`, that is built with a `HamiltonianArchitecture`, gives the corresponding Hamiltonian loss.

Functor

```
loss(c, ps, input, output)
loss(ps, input, output) # equivalent to the above
```

source

`GeometricMachineLearning.symbolic_hamiltonian_vector_field` – Method.

```
symbolic_hamiltonian_vector_field(nn::SymbolicNeuralNetwork)
```

Get the symbolic expression for the vector field belonging to the HNN `nn`.

Implementation

This is calling `SymbolicNeuralNetworks.Jacobian` and then multiplies the result with a Poisson tensor.

source

`SymbolicNeuralNetworks.SymbolicPullback` – Method.

```
SymbolicPullback(arch::HamiltonianArchitecture)
```

Make a `SymbolicPullback` based on a `HamiltonianArchitecture`.

Implementation

Internally this is calling `SymbolicNeuralNetwork` and `HNNLoss`.

source

`GeometricMachineLearning.GeneralizedHamiltonianArchitecture` – Type.

```
GeneralizedHamiltonianArchitecture <: HamiltonianArchitecture
```

A realization of generalized Hamiltonian neural networks (GHNNs) as introduced in [51].

Also see `StandardHamiltonianArchitecture`.

Constructor

The constructor takes the following input arguments:

1. `dim`: system dimension,
2. `width = dim`: width of the hidden layer. By default this is equal to `dim`,
3. `nhidden = 1`: the number of hidden layers,
4. `activation = AbstractNeuralNetworks.TanhActivation()`: the activation function used in the GHNN,
5. `integrator = nothing`: the integrator that is used to design the GHNN.

source

`GeometricMachineLearning._processing` – Function.

```
_processing(returned_pullback)
```

Strip `returned_pullback` from unnecessary `Zygote`-induces garbage.

Also see the docs for `ZygotePullback`.

source

8.5 SympNet Architecture

This section discusses the symplectic neural network (SympNet) architecture and its implementation in `GeometricMachineLearning`.

Principle

SympNets [10] are a type of neural network that can model the trajectory of a canonical Hamiltonian system (see Chapter 3.1) in phase space. Take $(q^T, p^T)^T = (q_1, \ldots, q_d, p_1, \ldots, p_d)^T \in \mathbb{R}^{2d}$ as the coordinates in phase space, where $q = (q_1, \ldots, q_d)^T \in \mathbb{R}^d$ is refered to as the *position* and $p = (p_1, \ldots, p_d)^T \in \mathbb{R}^d$ the *momentum*. Given a point

$$\begin{pmatrix} q^{(m)} \\ p^{(m)} \end{pmatrix} \in \mathbb{R}^{2d}$$

the SympNet aims to compute the *next position*

$$\text{SympNet}\left(\begin{pmatrix} q^{(m)} \\ p^{(m)} \end{pmatrix}\right) = \begin{pmatrix} \tilde{q}^{(m+1)} \\ \tilde{p}^{(m+1)} \end{pmatrix} \in \mathbb{R}^{2d}$$

and thus predicts the trajectory while preserving the *symplectic structure* of the system. SympNets are enforcing symplecticity strongly, meaning that this property is hard-coded into the network architecture. The layers are reminiscent of traditional neural network feedforward layers, but have a strong restriction imposed on them in order to be symplectic.

SympNets can be viewed as a *symplectic integrator* or symplectic one-step method[1] [3, 84]. Their goal is to predict, based on an initial condition $((q^{(0)})^T, (p^{(0)})^T)^T$, a sequence of points in phase space that fit the training data as well as possible:

$$\begin{pmatrix} q^{(0)} \\ p^{(0)} \end{pmatrix}, \begin{pmatrix} \tilde{q}^{(1)} \\ \tilde{p}^{(1)} \end{pmatrix}, \cdots \begin{pmatrix} \tilde{q}^{(n)} \\ \tilde{p}^{(n)} \end{pmatrix}.$$

The tilde in the above equation indicates *predicted data*. With standard SympNets[2] the time step between predictions is not a parameter we can choose but is related to the *temporal frequency of the training data*. This means that if data is recorded in an interval of e.g. 0.1 seconds, then this will be the time step of our integrator.

SympNets preserve symplecticity by exploiting the (q, p) structure of the system. This is visualized below:

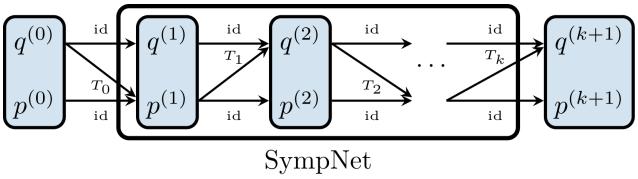


Figure 8.5: Visualization of the SympNet architecture.

In the figure above we see that an update for q is based on data coming from p and an update for p is based on data coming from q. $T_i : \mathbb{R}^d \to \mathbb{R}^d$ is an operation that changes p when i is even and changes q when odd. It has the special property that its Jacobian is a symmetric matrix. There are two types of SympNet architectures: LA-SympNets and G-SympNets.

[1] *Symplectic multi-step methods* can be modeled with transformers (see Chapter 8.9).

[2] Recently an approach [51] has been proposed that makes explicitly specifying the time step possible by viewing SympNets as a subclass of so-called "Generalized Hamiltonian Neural Networks".

LA-SympNet

The first type of SympNets, LA-SympNets, are obtained from composing two types of layers: symplectic linear layers and symplectic activation layers (see Chapter 7.1). For a given integer w, the *linear part* of an LA-SympNet is

$$\mathcal{L}^{w,\text{up}}\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} \mathbb{I} & A^w/\mathbb{O} \\ \mathbb{O}/A^w & \mathbb{I} \end{pmatrix} \cdots \begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A^2 & \mathbb{I} \end{pmatrix} \begin{pmatrix} \mathbb{I} & A^1 \\ \mathbb{O} & \mathbb{I} \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix} + b,$$

or

$$\mathcal{L}^{w,\text{low}}\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} \mathbb{I} & \mathbb{O}/A^w \\ A^w/\mathbb{O} & \mathbb{I} \end{pmatrix} \cdots \begin{pmatrix} \mathbb{I} & A^2 \\ \mathbb{O} & \mathbb{I} \end{pmatrix} \begin{pmatrix} \mathbb{I} & \mathbb{O} \\ A^1 & \mathbb{I} \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix} + b.$$

The superscripts up and low indicate whether the q or the p part is first changed[3]. The learnable parameters are the symmetric matrices $A^i \in \mathcal{S}_{\text{sym}}(d)$ and the bias $b \in \mathbb{R}^{2d}$. The integer w is the number of linear layers in one block. It can be shown that five of these layers, i.e. $w \geq 5$, can represent any linear symplectic map [88], so w need not be larger than five. We denote the set of symplectic linear layers by $\mathcal{M}^L$.

The second type of layer needed for LA-SympNets are activation layers (see Chapter 7.1).

An LA-SympNet is a mapping of the form $\Psi = l_k \circ a_k \circ l_{k-1} \circ \cdots \circ a_1 \circ l_0$ where $(l_i)_{0\leq i\leq k} \subset \mathcal{M}^L$ and $(a_i)_{1\leq i\leq k} \subset \mathcal{M}^A$. We will refer to k as the *number of hidden layers* of the SympNet[4] and the number w above as the *depth* of the linear layer.

We give an example of calling LA-SympNet:

```
using GeometricMachineLearning

k = 1
w = 2
arch = LASympNet(4;
                    nhidden = k,
                    depth = 2,
                    init_upper_linear = true,
                    init_upper_act = true,
                    activation = tanh)

model = Chain(arch).layers
```

```
(LinearLayerQ{4, 4}(), LinearLayerP{4, 4}(), GeometricMachineLearning.BiasLayer{4, 4}(),
↪  ActivationLayerQ{4, 4, typeof(tanh)}(tanh), ActivationLayerP{4, 4, typeof(tanh)}(tanh),
↪  LinearLayerQ{4, 4}(), LinearLayerP{4, 4}(), GeometricMachineLearning.BiasLayer{4, 4}())
```

[3] "up" means we first change the q part and "low" means we first change the p part. This can be set via the keyword `init_upper_linear` in `LASympNet`.

[4] Note that if $k = 0$ then the LA-SympNet consists of only one linear layer.

The keywords `init_upper_linear` and `init_upper_act` indicate whether the first linear (respectively activation) layer is of q type[5].

G-SympNets

G-SympNets are an alternative to LA-SympNets. They are built with only one kind of layer, the gradient layer (see Chapter 7.1). If we denote by $\mathcal{M}^G$ the set of gradient layers, a G-SympNet is a function of the form $\Psi = g_k \circ g_{k-1} \circ \cdots \circ g_1$ where $(g_i)_{1 \leq i \leq k} \subset \mathcal{M}^G$. The index k here is the *number of layers* in the SympNet.

```
using GeometricMachineLearning

k = 2
n = 10
arch = GSympNet(4; upscaling_dimension = n, n_layers = k, init_upper = true, activation = tanh)

model = Chain(arch).layers
```

```
(GradientLayerQ{4, 4, typeof(tanh)}(10, tanh), GradientLayerP{4, 4, typeof(tanh)}(10, tanh))
```

The keyword `init_upper` for `GSympNet` is similar as in the case for `LASympNet`. The keyword `upscaling_dimension` is explained in the section on the SympNet gradient layer (see Chapter 7.1).

Universal Approximation Theorems

In order to state the *universal approximation theorem* for both architectures we first need a few definitions:

Let U be an open set of $\mathbb{R}^{2d}$, and let us denote by $\mathcal{SP}^r(U)$ the set of C^r smooth symplectic maps on U. We now define a topology on $C^r(K, \mathbb{R}^{2d})$, the set of C^r-smooth maps from a compact set $K \subset U$ to $\mathbb{R}^{2d}$ through the norm

$$||f||_{C^r(K,\mathbb{R}^{2d})} = \sum_{|\alpha| \leq r} \max_{1 \leq i \leq 2d} \sup_{x \in K} |D^\alpha f_i(x)|,$$

where the differential operator D^α is defined by

$$D^\alpha f = \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} ... x_n^{\alpha_n}},$$

with $|\alpha| = \alpha_1 + ... + \alpha_{2d}$. We impose the following condition (r finiteness) on the activation function:

Definition 8.5.1. σ is **r-finite** if $\sigma \in C^r(\mathbb{R}, \mathbb{R})$ and $\int |D^r \sigma(x)| dx < \infty$.

[5] Similarly to `init_upper_linear`, if `init_upper_act = true` then the first activation layer is of q type, i.e. changes the q component and leaves the p component unchanged.

We further consider the topology on $C^r(U, \mathbb{R}^d)$ induced by $||\cdot||_{C^r(\cdot,\mathbb{R}^d)}$ and the associated notion of denseness (see Chapter 2.1):

Definition 8.5.2. Let $m, d, r \in \mathbb{N}$ with $m, d > 0$ be given, U an open subset of $\mathbb{R}^m$, and $I, J \subset C^r(U, \mathbb{R}^d)$. We say J is **r-uniformly dense on compacta in** I if $J \subset I$ and for any $f \in I$, $\epsilon > 0$, and any compact $K \subset U$, there exists $g \in J$ such that $||f - g||_{C^r(K,\mathbb{R}^d)} < \epsilon$.

Remark 8.5.3. The associated topology to this notion of denseness is the **compact-open topology**. It is generated by the following sets:

$$V(K, U) := \{f \in C^r(\mathbb{R}^m, \mathbb{R}^d) : \text{ such that } f(K) \subset U\},$$

i.e. the *compact-open topology* is the smallest topology that contains all sets of the form $V(K, U)$.

We can now state the universal approximation theorems:

Theorem 8.5.4 (Approximation theorem for LA-SympNets)**.** *For any positive integer $r > 0$ and open set $U \in \mathbb{R}^{2d}$, the set of LA-SympNet is r-uniformly dense on compacta in $SP^r(U)$ if the activation function σ is r-finite.*

and

Theorem 8.5.5 (Approximation theorem for G-SympNets)**.** *For any positive integer $r > 0$ and open set $U \in \mathbb{R}^{2d}$, the set of G-SympNet is r-uniformly dense on compacta in $SP^r(U)$ if the activation function σ is r-finite.*

There are many r-finite activation functions commonly used in neural networks, for example:

- The sigmoid activation function: $\sigma(x) = 1/(1 + e^{-x})$,
- The hyperbolic tangent function: $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$.

The universal approximation theorems state that we can, in principle, get arbitrarily close to any symplectomorphism defined on $\mathbb{R}^{2d}$. But this does not tell us anything about how to optimize the network. This is can be done with any common neural network optimizer (see Chapter 5.1) and these neural network optimizers always rely on a corresponding loss function.

Loss function

To train the SympNet, one needs data along a trajectory such that the model is trained to perform an integration. The loss function is defined as[6]:

$$\text{loss}(z^{\text{c}}, z^{\text{p}}) = \frac{||z^{\text{c}} - z^{\text{p}}||}{||z^{\text{c}}||},$$

where

[6]This loss function is implemented as `FeedForwardLoss` in `AbstractNeuralNetworks`.

$$z^{\mathrm{c}} = \begin{pmatrix} q^{\mathrm{c}} \\ p^{\mathrm{c}} \end{pmatrix}$$

is the current state and

$$z^{\mathrm{p}} = \begin{pmatrix} q^{\mathrm{p}} \\ p^{\mathrm{p}} \end{pmatrix}$$

is the predicted state. In the example section (see Chapter 10.1) we show how to use SympNets in `GeometricMachineLearning.jl` and how to modify the loss function (see Chapter C.1).

Library Functions

`GeometricMachineLearning.SympNet` – Type.

```
SympNet <: NeuralNetworkIntegrator
```

The `SympNet` type encompasses `GSympNet`s and `LASympNet`s. SympNets [10] are universal approximators of *canonical symplectic flows*. This means that for every map

$$\varphi : \mathbb{R}^{2n} \to \mathbb{R}^{2n} \text{ with } (\nabla\varphi)^T \mathbb{J} \nabla\varphi = \mathbb{J},$$

we can find a SympNet that approximates φ arbitrarily well.

source

`GeometricMachineLearning.LASympNet` – Type.

```
LASympNet(d)
```

Make an LA-SympNet with dimension d.

There exists an additional constructor that can be called by supplying an instance of `DataLoader`.

Examples

```
using GeometricMachineLearning
dl = DataLoader(rand(2, 20); suppress_info = true)
LASympNet(dl)

# output

LASympNet{typeof(tanh), true, true}(2, 5, 1, tanh)
```

Arguments

Keyword arguments are:

- `depth::Int = 5`: The number of linear layers that are applied.
- `nhidden::Int = 1`: The number of hidden layers (i.e. layers that are *not* output layers).
- `activation = tanh`: The activation function that is applied.
- `init_upper_linear::Bool = true`: Initialize the linear layer so that it first modifies the q-component.
- `init_upper_act::Bool = true`: Initialize the activation layer so that it first modifies the q-component.

source

`GeometricMachineLearning.GSympNet` – Type.

```
GSympNet(d)
```

Make a G-SympNet with dimension d.

There exists an additional constructor that can be called by supplying an instance of `DataLoader` (see `LASympNet` for an example of using this constructor).

Arguments

Keyword arguments are:

- `upscaling_dimension::Int = 2d`: The *upscaling dimension* of the gradient layer. See the documentation for `GradientLayerQ` and `GradientLayerP` for further explanation.
- `n_layers::Int2`: The number of layers (i.e. the total number of `GradientLayerQ` and `GradientLayerP`).
- `activationtanh`: The activation function that is applied.
- `init_upper::Booltrue`: Initialize the gradient layer so that it first modifies the q-component.

source

8.6 Volume-Preserving Feedforward Neural Network

The volume-preserving feedforward neural network presented here can be seen as an adaptation of an LA-SympNet to the setting when we deal with a divergence-free vector field (see Chapter 3.2). It also serves as the feedforward module in the volume-preserving transformer (see Chapter 8.8).

Neural network architecture

The constructor for `VolumePreservingFeedForward` produces the following architecture[1]:

[1] Based on the input arguments `n_linear` and `n_blocks`. In this example `init_upper` is set to false, which means that the first layer is of type *lower* followed by a layer of type *upper*.

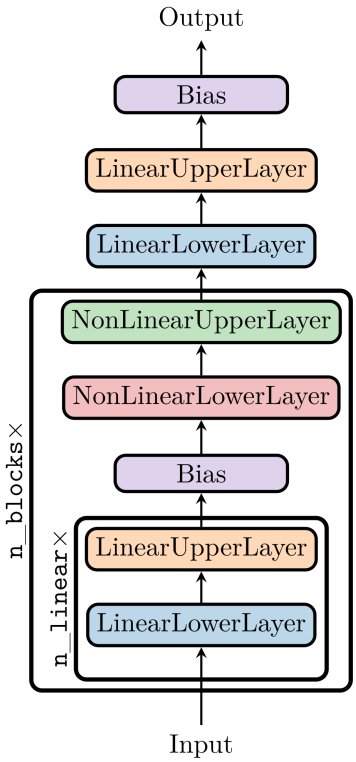


Figure 8.6: Visualization of how the keywords in the constructor are interpreted.

Here "LinearLowerLayer" performs

$$\text{LinearLowerLayer}_L : x \mapsto x + Lx$$

and "NonLinearLowerLayer" performs

$$\text{NonLinearLowerLayer}_L : x \mapsto x + \sigma(Lx + b).$$

The activation function σ is the forth input argument to the constructor and `tanh` by default. We can make an instance of a `VolumePreservingFeedForward` neural network:

```
using GeometricMachineLearning

const d = 3

arch = VolumePreservingFeedForward(d)


for layer in Chain(arch)
    println(stdout, layer)
end
```

```
VolumePreservingLowerLayer{3, 3, :no_bias, typeof(identity)}(identity)
VolumePreservingUpperLayer{3, 3, :bias, typeof(identity)}(identity)
VolumePreservingLowerLayer{3, 3, :bias, typeof(tanh)}(tanh)
VolumePreservingUpperLayer{3, 3, :bias, typeof(tanh)}(tanh)
VolumePreservingLowerLayer{3, 3, :no_bias, typeof(identity)}(identity)
VolumePreservingUpperLayer{3, 3, :bias, typeof(identity)}(identity)
```

And we see that we get the same architecture as shown in the figure above, with the difference that the bias has been subsumed in the previous layers. Note that the nonlinear layers also contain a bias vector.

Note on Sympnets

In the general framework of feedforward neural networks SympNets (see Chapter 8.5) are more restrictive than volume-preserving neural networks as symplecticity is a stronger property than volume-preservation:

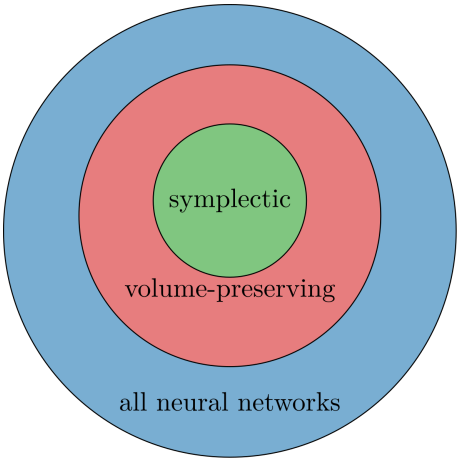


Figure 8.7: Symplectic neural networks are a more restrictive class of architectures than volume-preserving ones. But by construction they only work in even dimensions.

Note however that SympNets rely on data in canonical form, i.e. data that is of (q, p) type (called `GeometricMachineLearning.QPT` in `GeometricMachineLearning`), so those data need to come from a vector space $\mathbb{R}^{2n}$ of even dimension. Volume-preserving feedforward neural networks also work for odd-dimensional spaces. This is also true for transformers: the volume-preserving transformer (see Chapter 8.8) works in spaces of arbitrary dimension $\mathbb{R}^{n\times T}$, whereas the linear symplectic transformer (see Chapter 8.9) only works in even-dimensional spaces $\mathbb{R}^{2n\times T}$.

Library Functions

`GeometricMachineLearning.VolumePreservingFeedForward` – Type.

```
VolumePreservingFeedForward(dim)
```

Make an instance of a volume-preserving feedforward neural network for a specific system dimension.

This architecture is a composition of `VolumePreservingLowerLayer` and `VolumePreservingUpperLayer`.

Arguments

You can provide the constructor with the following additional arguments:

1. `n_blocks::Int = 1`: The number of blocks in the neural network (containing linear layers and nonlinear layers).
2. `n_linear::Int = 1`: The number of linear `VolumePreservingLowerLayer`s and `VolumePreservingUpperLayer`s in one block.
3. `activation = tanh`: The activation function for the nonlinear layers in a block.

The following is a keyword argument:

- `init_upper::Bool = false`: Specifies if the first layer is lower or upper.

source

8.7 Standard Transformer

The transformer is a relatively modern neural network architecture [9] that has come to dominate the field of natural language processing (NLP, [28]) and replaced the previously dominant long-short term memory cells (LSTM, [85]). Its success is due to a variety of factors:

- unlike LSTMs it consists of very simple building blocks and hence is easier to interpret mathematically,
- it is very flexible in its application and the data it is fed with do not have to conform to a rigid pattern,
- transformers utilize modern hardware (especially GPUs) very effectively.

The transformer architecture is sketched below:

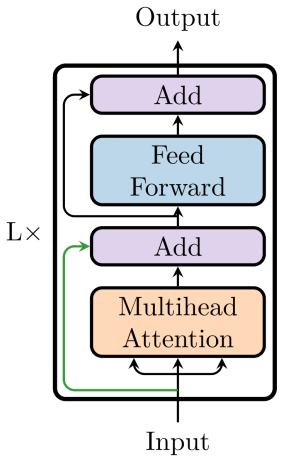


Figure 8.8: Visualization of the standard transformer. It consists of two components: a mulithead attention layer and a feedforward neural network.

It is nothing more than a combination of a multihead attention layer (see Chapter 7.4) and a residual neural network[1] (ResNet).

As was explained when we talked about the attention module (see Chapter 7.3), the attention layer performs a convex reweighting of the input sequence:

$$\text{Attention} : Z \equiv [z^{(1)}, \dots, z^{(T)}] \mapsto [\sum_{i=1}^{T} p_i^{(1)} z^{(i)}, \dots, \sum_{i=1}^{T} p_i^{(T)} z^{(i)}] = \text{Attention}(Z),$$

where the coefficients $p^{(i)}$ depend on Z and are *learnable*. In the case of multihead attention (see Chapter 7.4) a greater number of these *reweighting coefficients* are learned, but it is otherwise not much more complicated than single-head attention.

The green arrow in the figure above indicates that this first *add connection* can be left out. This can be specified via the keyword argument `add_connection` in `MultiHeadAttention` layer and the `StandardTransformerIntegrator`.

We should also note that such transformers have been used for the online phase in reduced order modeling (see Chapter 4.1) before [87].

Classification Transformer

Instead of using the transformer for integration, it can also be used as a image classifier. In this case it is often referred to as "vision transformer" [29]. In this case we append a `ClassificationLayer` to the output of the transformer. This will be used in one of the examples (see Chapter 11.1).

[1] A layer of type GeometricMachineLearning.ResNetLayer is nothing more than a neural network to whose output we again add the input, i.e. every ResNet is of the form $\text{ResNet}(x) = x + \mathcal{NN}(x)$.

The Upscaling

When using the transformer one typically also benefits from defining a `transformer_dim` that is greater than the system dimension and a corresponding `upscaling_activation` (see the docstring of `StandardTransformerIntegrator`).

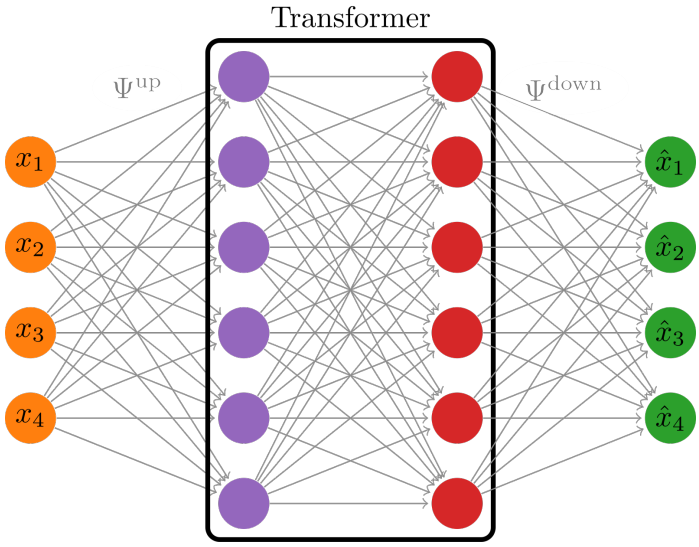


Figure 8.9: If the transformer dimension is not equal to the system dimension, then we add two more neural network layers. One that maps up to the space whose dimension is the transformer dimension and one that maps down again to the space whose dimension is the system dimension.

In the figure above we call

$$\Psi^{\text{up}} : \mathbb{R}^{\texttt{sys\_dim}} \to \mathbb{R}^{\texttt{transformer\_dim}}$$

the *upscaling layer* and

$$\Psi^{\text{down}} : \mathbb{R}^{\texttt{transformer\_dim}} \to \mathbb{R}^{\texttt{sys\_dim}}$$

the *downscaling layer*. Both of these layers are dense layers with the activation function for the downscaling layer being the identity (for better expressivity) and the activation function for the upscaling layer can be specified via the keyword `upscaling_activation`.

`GeometricMachineLearning` does not have an implementation of such an upscaling for the volume-preserving transformer (see Chapter 8.8) and the linear symplectic transformer (see Chapter 8.9). Symplectic liftings have however recently been discussed to learn higher-dimensional Hamiltonian representations of given data [63].

Library Functions

`GeometricMachineLearning.StandardTransformerIntegrator` – Type.

```
StandardTransformerIntegrator(sys_dim)
```

Make an instance of `StandardTransformerIntegrator` for a specific system dimension.

Here the standard transformer used as an integrator (see `TransformerIntegrator`).

It is a composition of `MultiHeadAttention` layers and `ResNetLayer` layers.

Arguments

The following are optional keyword arguments:

- `transformer_dim::Int = sys_dim`: this is the dimension *after the upscaling*.
- `n_blocks::Int = 1` : the number of `ResNetLayer` blocks.
- `n_heads::Int = sys_dim`: the number of heads in the multihead attention layer.
- `L::Int = 2`: the number of transformer blocks.
- `upscaling_activation = identity`: the activation used in the upscaling layer.
- `resnet_activation = tanh`: the activation used for the `ResNetLayer`.
- `attention_activation = GeometricMachineLearning.VectorSoftmax()` : the activation used for the `MultiHeadAttention` layer.
- `add_connection:Bool = true`: specifies if the input should be added to the output.

source

`GeometricMachineLearning.Transformer` – Function.

```
Transformer(dim, n_heads, L)
```

Make an instance of the Transformer with `n_heads` for dimension `dim` and `L` blocks.

Arguments

`Transformer` takes the following optional keyword arguments:

- `activation = tanh`: the activation function used for the `ResNetLayer`.
- `Stiefel::Bool = false`: if the matrices P^V, P^Q and P^K should live on a manifold.
- `add_connection::Bool = false`: if the input should by added to the ouput after the `MultiHeadAttention` layer.
- `use_bias::Bool = true`: Specifies if the `ResNetLayer` should use a bias.

source

`GeometricMachineLearning.ClassificationTransformer` – Type.

```
ClassificationTransformer(dl)
```

Make an instance of the `ClassificationTransformer` based on an instance of `DataLoader`.

This is a transformer neural network for classification purposes. At the moment this is only used for training on MNIST, but can in theory be used for any classification problem.

Arguments

The optional keyword arguments are:

- `n_heads::Int=7`: The number of heads in the `MultiHeadAttention` (mha) layers.
- `L::Int=16`: The number of transformer blocks.
- `activation=softmax`: The activation function.
- `Stiefel::Bool=true`: Whether the matrices in the mha layers are on the `StiefelManifold`.
- `add_connection::Bool=true`: Whether the input is appended to the output of the mha layer. (skip connection)

source

`GeometricMachineLearning.assign_output_estimate` – Function.

```
assign_output_estimate(full_output, prediction_window)
```

Crop the output to get the correct number of output vectors.

The function `assign_output_estimate` is closely related to the `Transformer`. It takes the last `prediction_window` columns of the output and uses them for the prediction.

i.e.

$$\mathbb{R}^{N\times T} \to \mathbb{R}^{N\times \texttt{pw}}, \begin{bmatrix} z_1^{(1)} & \cdots & z_1^{(T)} \\ \cdots & \cdots & \cdots \\ z_n^{(1)} & \cdots & z^{(T)}{}_n \end{bmatrix} \mapsto \begin{bmatrix} z_1^{(T-\texttt{pw})} & \cdots & z_1^{(T)} \\ \cdots & \cdots & \cdots \\ z_n^{(T-\texttt{pw})} & \cdots & z^{(T)}{}_n \end{bmatrix}$$

If `prediction_window` is equal to `sequence_length`, then this is not needed.

source

8.8 Volume-Preserving Transformer

The volume-preserving transformer [30] is, similar to the standard transformer, a combination of two different neural networks: a volume-preserving attention layer (see Chapter 7.3) and a volume-preserving feedforward layer (see Chapter 8.6). It is visualized below:

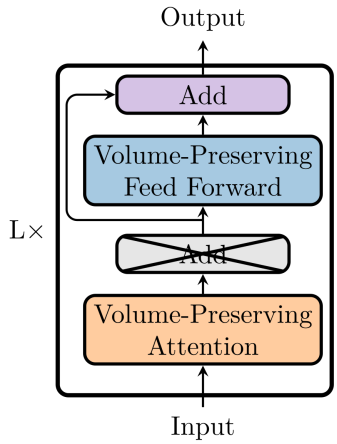


Figure 8.10: Visualization of the Volume-Preserving Transformer architecture.

In the figure we indicate that we leave out the *add connection*. When talking about the standard transformer (see Chapter 8.7) we said that the add connection is optional and can be included via the keyword argument `add_connection`. For the volume-preserving transformer this is not true: it is always excluded.

Note that the volume-preserving transformer preserves the volume in the sense of the product spaces. That the `VolumePreservingAttention` layer preserves this structure was discussed when we introduced it (see Chapter 7.3). That the `VolumePreservingFeedForwardLayer`s preserve this structure on the product space is also easy to see. We take a `VolumePreservingFeedForwardLayer`, e.g.

$$\psi : z \mapsto \sigma(Lz + b),$$

and look at its action on the element of the product space $[z^{(1)}, \dots, z^{(T)}] = Z \in \mathbb{R}^{d \times T}$:

$$\psi_T : [z^{(1)}, \dots, z^{(T)}] \mapsto [\psi(z^{(1)}), \dots, \psi(z^{(T)})].$$

The jacobian of $\hat{\psi}_T : \mathbb{R}^{dT} \to \mathbb{R}^{dT}$, the representation of ψ_T in the *coordinate system* of the big vector (see Chapter 7.3), is of the form[1]

$$\nabla\hat{\psi}_T = \begin{bmatrix} J & \mathbb{O} & \cdots & \mathbb{O} \\ \mathbb{O} & J & \cdots & \mathbb{O} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbb{O} & \mathbb{O} & \cdots & J \end{bmatrix} = J \otimes \mathbb{I}_T,$$

where J is the jacobian of ψ. We now see that $\det(\nabla\hat{\psi}_T) = 1$ and volume in the product space is preserved.

[1] In order to arrive at this representation we possibly have to exchange the order of the rows in the matrix. This is however not critical since it may only cause a sign change in the determinant.

Library Functions

`GeometricMachineLearning.VolumePreservingTransformer` – Type.

```
VolumePreservingTransformer(sys_dim, seq_length)
```

Make an instance of the volume-preserving transformer for a given system dimension and sequence length.

Arguments

The following are keyword argumetns:

- `n_blocks::Int = 1`: The number of blocks in one transformer unit (containing linear layers and nonlinear layers).
- `n_linear::Int = 1`: The number of linear `VolumePreservingLowerLayer`s and `VolumePreservingUpperLayer`s in one block.
- `L::Int = 1`: The number of transformer units.
- `activation = tanh`: The activation function.
- `init_upper::Bool = false`: Specifies if the network first acts on the q component.
- `skew_sym::Bool = false`: specifies if we the weight matrix is skew symmetric or arbitrary.

source

8.9 Linear Symplectic Transformer

The linear symplectic transformer consists of a combination of linear symplectic attention (see Chapter 7.5) and gradient layers (see Chapter 7.1) and is visualized below.

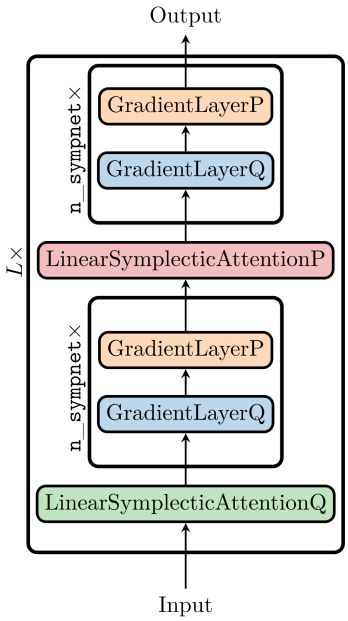


Figure 8.11: Visualization of the linear symplectic transformer architecutre. In this figure the number of SympNet layers per transformer block is two.

In this picture we also visualize the keywords `n_sympnet` and L for `LinearSymplecticTransformer`.

What we discussed for the volume-preserving transformer (see Chapter 8.8) also applies here: the attention mechanism acts on all the input vectors at once and is designed such that it preserves the product structure (here this is the symplectic product structure). The attention mechanism serves as a *preprocessing step* after which we apply a regular feedforward neural network; here this is a SympNet (see Chapter 8.5).

Why use Transformers for Model Order Reduction

The standard transformer (see Chapter 8.7), the volume-preserving transformer (see Chapter 8.8) and the linear symplectic transformer are suitable for model order reduction for a number of reasons. Besides their improved accuracy [87] their ability to resolve time series data also makes it possible to deal with data that come from multiple parameters. For this consider the following two trajectories:

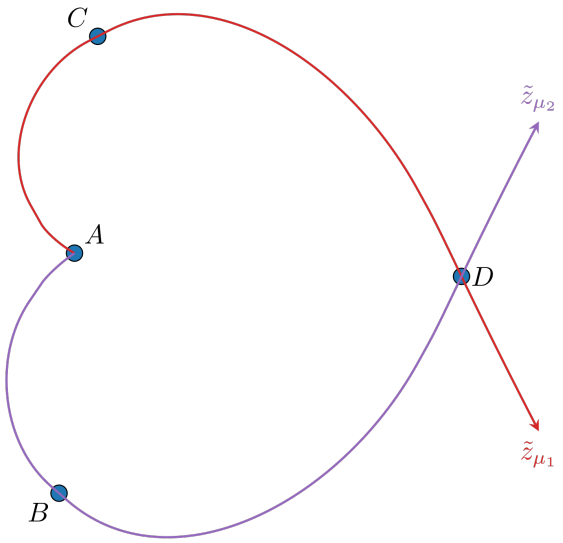


Figure 8.12: Two trajectories of a parameter-dependent ODE with the same initial condition.

The trajectories come from a parameter-dependent ODE (see Chapter 2.6) in two dimensions. As initial condition we take $A \in \mathbb{R}^2$ and we look at two different parameter instances: μ_1 and μ_2. As we can see the curves $\tilde{z}_{\mu_1}$ and $\tilde{z}_{\mu_2}$ both start out at A, then go into different directions but cross again at D. If we used a standard feedforward neural network to treat this system it would not be able to resolve those training data as the information would be ambiguous at points A and D, i.e. the network would not know what it should predict. If we however consider the information coming from points three points, either (A, B, D) or (A, C, D), then the network can learn to predict the next time step. We will elaborate more on this in the tutorial section (see Chapter 11.3).

Library Functions

`GeometricMachineLearning.LinearSymplecticTransformer` – Type.

```
LinearSymplecticTransformer(sys_dim, seq_length)
```

Make an instance of `LinearSymplecticTransformer` for a specific system dimension and sequence length.

Arguments

You can provide the additional optional keyword arguments:

- `n_sympnet::Int = (2)`: The number of sympnet layers in the transformer.
- `upscaling_dimension::Int = 2*dim`: The upscaling that is done by the gradient layer.
- `L::Int = 1`: The number of transformer units.
- `activation = tanh`: The activation function for the SympNet layers.
- `init_upper::Bool=true`: Specifies if the first layer is a q-type layer (`init_upper=true`) or if it is a p-type layer (`init_upper=false`).

The number of SympNet layers in the network is `2n_sympnet`, i.e. for `n_sympnet = 1` we have one `GradientLayerQ` and one `GradientLayerP`.

source

Chapter Summary

In this chapter we discussed various neural network architectures and presented the corresponding application interface in `GeometricMachineLearning`. Of the presented architectures the *symplectic autoencoder*, the *volume-preserving transformer* and the *linear symplectic transformer* constitute novel architectures. We construced the symplectic autoencoder as a composition of *PSD-like layers* and *SympNet* layers. For optimizing this architecture we need to resort to manifold optimization. The volume-preserving transformer is a composition of *volume-preserving attention layers* and *volume-preserving feedforward layers*; this is similar for the liner symplectic transformer: it is a composition of *linear symplectic attention layers* and *SympNets*.

We discussed how all three transformer neural networks can be seen as *multi-step integrators*. This was elaborated on in an extra section on *neural network-based integrators*.

References

[33] B. Brantner and M. Kraus. *Symplectic autoencoders for Model Reduction of Hamiltonian Systems*, arXiv preprint arXiv:2312.10004 (2023).

[44] M. Kraus. *GeometricIntegrators.jl: Geometric Numerical Integration in Julia*, https://github.com/JuliaGNI/GeometricIntegrators.jl (2020).

[126] K. Feng. *The step-transition operators for multi-step methods of ODE's.* Journal of Computational Mathematics, 193–202 (1998).

[30] B. Brantner, M. Kraus, Z. Li and G. de Romemont. *Volume-preserving transformers for learning time series data with structure.* ESAIM: Proceedings and Surveys **81**, 123–144 (2025).

8.10 The Symplectic Transformer

The symplectic transformer is, like the standard transformer (see Chapter 8.7), a combination of *attention layers* and *feedforward layers*. The difference is that the attention layers are not multihead attention layers (see Chapter 7.4) and the feedforward layers are not standard `GeometricMachineLearning.ResNetLayer`s, but symplectic attention layers (see Chapter 7.6) and sympnet layers (see Chapter 7.1).

Library Functions

`GeometricMachineLearning.SymplecticTransformer` – Type.

```
SymplecticTransformer <: TransformerIntegrator
```

Constructors

```
SymplecticTransformer(sys_dim)
```

Make an instance of `SymplecticTransformer` for a specific system dimension and sequence length. Also see `LinearSymplecticTransformer`.

Arguments

You can provide the additional optional keyword arguments:

- `n_sympnet::Int = (2)`: The number of sympnet layers in the transformer.
- `upscaling_dimension::Int = 2*dim`: The upscaling that is done by the gradient layer.
- `L::Int = 1`: The number of transformer units.
- `sympnet_activation = tanh`: The activation function for the SympNet layers.
- `attention_activation = GeometricMachineLearning.MatrixSoftmax()`: The activation function for the Attention layers.
- `init_upper::Bool=true`: Specifies if the first layer is a q-type layer (`init_upper=true`) or if it is a p-type layer (`init_upper=false`).
- `symmetric::Bool=false`:

The number of SympNet layers in the network is `2n_sympnet`, i.e. for `n_sympnet = 1` we have one `GradientLayerQ` and one `GradientLayerP`.

source

Part IV

Experiments and Applications

Chapter 9

Learning a Reduced Model with Symplectic Autoencoders

In this chapter we show how to use symplectic autoencoders for the offline phase of reduced order modeling. The example we consider here is the *Toda lattice*. We further discuss the online phase and compare a standard integrator (implicit midpoint) with a transformer neural network.

9.1 Symplectic Autoencoders and the Toda Lattice

In this tutorial we use a symplectic autoencoder (see Chapter 8.2) to approximate the solution of the Toda lattice with a lower-dimensional Hamiltonian model and compare it with standard proper symplectic decomposition (see Chapter 4.5).

Remark 9.1.1. As with any neural network we have to make the following choices:

1. specify the *architecture*,
2. specify the *type* and *backend*,
3. pick an *optimizer* for training the network,
4. specify how you want to perform *batching*,
5. choose a *number of epochs*,

where points 1 and 3 depend on a variable number of hyperparameters.

For the symplectic autoencoder point 1 is done by calling `SymplecticAutoencoder`, point 2 is done by calling `NeuralNetwork`, point 3 is done by calling `Optimizer` and point 4 is done by calling `Batch`.

The system

The Toda lattice [31] is a prototypical example of a Hamiltonian PDE. It is described by

$$H(q,p) = \sum_{n\in\mathbb{Z}} \left(\frac{p_n^2}{2} + \alpha e^{q_n - q_{n+1}} \right).$$

Starting from this equation we further assume a finite number of particles N and impose periodic boundary conditions:

$$q_{n+N} \equiv q_n$$
$$p_{n+N} \equiv p_n.$$

In this tutorial we want to reduce the dimension of the big system by a significant factor with (i) proper symplectic decomposition (PSD) and (ii) symplectic autoencoders (SAE). The first approach is strictly linear whereas the second one allows for more general mappings.

Using the Toda lattice in numerical experiments

In order to use the Toda lattice in numerical experiments we have to pick suitable initial conditions. For this, consider the third-degree spline:

$$h(s) = \begin{cases} 1 - \frac{3}{2}s^2 + \frac{3}{4}s^3 & \text{if } 0 \leq s \leq 1 \\ \frac{1}{4}(2-s)^3 & \text{if } 1 < s \leq 2 \\ 0 & \text{else.} \end{cases}$$

Plotted on the relevant domain it looks like this:

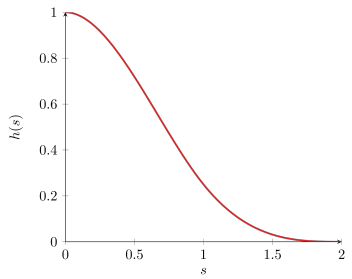


Figure 9.1: Third degree spline.

We end up with the following choice of parametrized initial conditions:

$$u_0(\mu)(\omega) = h(s(\omega, \mu)), \quad s(\omega, \mu) = 20\mu|\omega + \frac{\mu}{2}|,$$

where the ω is an element of the domain $\Omega = [-0.5, 0.5]$. For the purposes of this tutorial we will use the default value for μ provided in `GeometricProblems`:

```
import GeometricProblems.TodaLattice as tl

tl.μ
```

```
0.3
```

We thus look at the displacement of $N = 200$ particles on the periodic domain $\Omega = [-0.5, 0.5]/\sim \simeq S^1$ where the equivalence relation $\sim$ indicates that we associate the points -0.5 and 0.5 with each other.

Get the data

The training data can very easily be obtained by using the packages `GeometricProblems` and `GeometricIntegrators`:

```
using GeometricIntegrators: integrate, ImplicitMidpoint
using GeometricMachineLearning

pr = tl.hodeproblem(; timespan = (0.0, 800.))
sol = integrate(pr, ImplicitMidpoint())
```

We then put the data in the correct format by calling `DataLoader`[1]:

```
dl_cpu = DataLoader(sol; autoencoder = true, suppress_info = true)
```

Also note that the keyword `autoencoder` was set to true when calling `DataLoader`. The keyword argument `supress_info` determines whether data loader provides some additional information on the data it is called on.

Train the network

We now want to compare two different approaches: `PSDArch` and `SymplecticAutoencoder`. For this we first have to set up the networks:

```
const reduced_dim = 2

psd_arch = PSDArch(dl_cpu.input_dim, reduced_dim)
sae_arch = SymplecticAutoencoder(dl_cpu.input_dim, reduced_dim; n_encoder_blocks = 4,
                                                                n_decoder_blocks = 4,
                                                                n_encoder_layers = 2,
                                                                n_decoder_layers = 2)
```

Training a neural network is usually done by calling an instance of `Optimizer` in `GeometricMachineLearning`. `PSDArch` however can be solved directly by using singular value decomposition and this is done by calling solve!:

```
psd_nn_cpu = NeuralNetwork(psd_arch, CPU(), eltype(dl_cpu))

solve!(psd_nn_cpu, dl_cpu)
```

```
0.6946587819588325
```

The `SymplecticAutoencoder` we train with the `AdamOptimizerWithDecay` however[2]:

[1] For more information on `DataLoader` see the corresponding section (see Chapter A.3).

[2] It is not feasible to perform the training on CPU, which is why we use `CUDA` [89] here. We further perform the training in single precision.

```
using CUDA

const n_epochs = 262144
const batch_size = 4096

backend = CUDABackend()
dl = DataLoader(dl_cpu, backend, Float32)


sae_nn_gpu = NeuralNetwork(sae_arch, CUDADevice(), Float32)
o = Optimizer(sae_nn_gpu, AdamOptimizerWithDecay(integrator_train_epochs))

# train the network
o(sae_nn_gpu, dl, Batch(batch_size), n_epochs)
```

After training we map the network parameters to cpu:

```
const mtc = GeometricMachineLearning.map_to_cpu
```

```
sae_nn_cpu = mtc(sae_nn_gpu)
```

The online stage with a standard integrator

After having trained our neural network we can now evaluate it in the online stage of reduced complexity modeling:

```
psd_rs = HRedSys(pr, encoder(psd_nn_cpu), decoder(psd_nn_cpu); integrator = ImplicitMidpoint())
sae_rs = HRedSys(pr, encoder(sae_nn_cpu), decoder(sae_nn_cpu); integrator = ImplicitMidpoint())
```

We integrate the full system (again) as well as the two reduced systems[3]:

```
FOM + Implicit Midpoint: 93.321223 seconds (6.61 M allocations: 535.713 MiB, 0.10% gc time)
PSD + Implicit Midpoint: 5.269227 seconds (20.60 M allocations: 10.385 GiB, 67.62% gc time)
SAE + Implicit Midpoint: 355.384054 seconds (230.79 M allocations: 261.337 GiB, 17.84% gc time)
```

And plot the solutions for

```
time_steps = (0, 300, 800)
```

[3] All of this is done with `ImplicitMidpoint` as integrator.

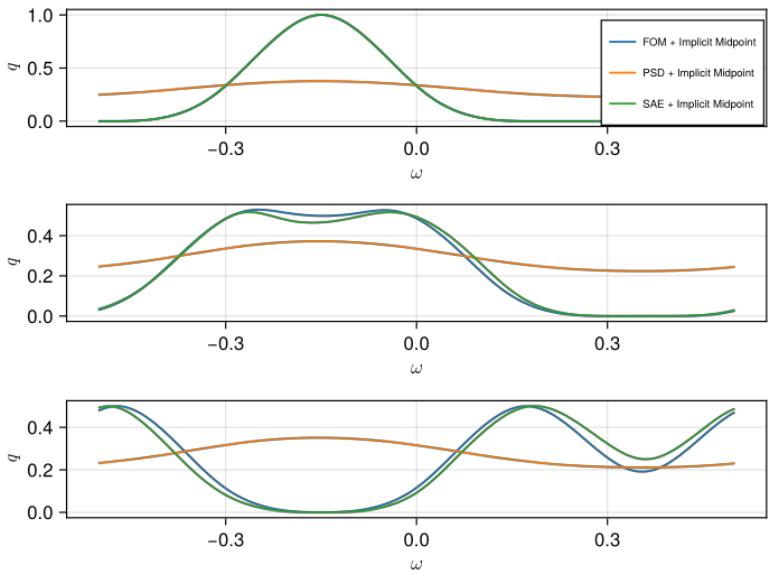


Figure 9.2: Comparison between FOM (blue), PSD with implicit midpoint (orange) and SAE with implicit midpoint (green).

We can see that the SAE has much more approximation capabilities than the PSD. But even though the SAE reasonably reproduces the full-order model (FOM), we see that the online stage of the SAE takes even longer than evaluating the FOM. In order to solve this problem we have to make the *online stage more efficient*.

The online stage with a neural network

Instead of using a standard integrator we can also use a neural network that is trained on the reduced data. For this:

```
const integrator_train_epochs = 65536
const integrator_batch_size = 4096
const seq_length = 4

integrator_architecture = StandardTransformerIntegrator(reduced_dim;
                                                        transformer_dim = 20,
                                                        n_blocks = 3,
                                                        n_heads = 5,
                                                        L = 3,
                                                        upscaling_activation = tanh)

integrator_nn = NeuralNetwork(integrator_architecture, backend)

integrator_method = AdamOptimizerWithDecay(integrator_train_epochs)

o_integrator = Optimizer(integrator_method, integrator_nn)

# map from autoencoder type to integrator type
dl_integration = DataLoader(dl; autoencoder = false)
```

```
integrator_batch = Batch(integrator_batch_size, seq_length)
```

```
loss = GeometricMachineLearning.ReducedLoss(encoder(sae_nn_gpu), decoder(sae_nn_gpu))

train_integrator_loss = o_integrator(   integrator_nn,
                                        dl_integration,
                                        integrator_batch,
                                        integrator_train_epochs,
                                        loss)
```

We can now evaluate the solution:

```
@time "time stepping with transformer" time_series = iterate(mtc(integrator_nn), ics; n_points =
↪  length(sol.t), prediction_window = seq_length)
```

```
time stepping with transformer: 0.197708 seconds (1.89 M allocations: 331.746 MiB, 67.05% gc time)
```

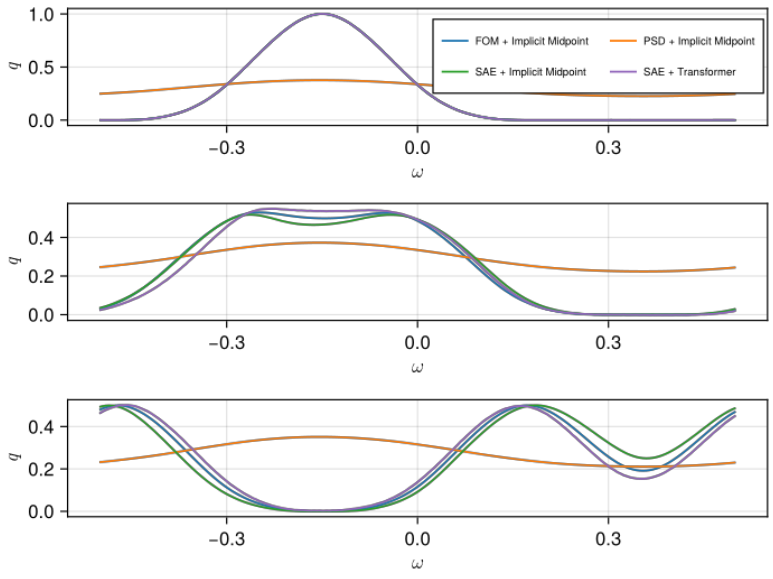


Figure 9.3: Comparison between FOM (blue), PSD with implicit midpoint (orange), SAE with implicit midpoint (green) and SAE with transformer (purple).

Note that integration of the system with the transformer is orders of magnitudes faster than any comparable method and also leads to an improvement in accuracy over the case where we build the reduced space with the symplectic autoencoder and use implicit midpoint in the online phase.

Remark 9.1.2. While training the symplectic autoencoder we completely ignore the online phase, but only aim at finding a good low-dimensional approximation to the solution manifold. This is why we observe that the approximated solution differs somewhat form the actual one when using implicit midpoint for integrating the low-dimensional system (blue line vs. green line).

Chapter Summary

We showed that for the Toda lattice we can achieve a very good approximation to the full-order system by using a two-dimensional reduced space; we also showed that for such a small space proper symplectic decomposition utterly fails.

We however also saw that when we use standard integrators in the online phase, the very low dimension of the reduced space may not mean fast computation. But by using a transformer as a neural network-based integrator we could circumvent this problem and achieve a speed-up of a factor of approximately 1000.

References

[11] L. Peng and K. Mohseni. *Symplectic model reduction of Hamiltonian systems.* SIAM Journal on Scientific Computing **38**, A1–A27 (2016).

[58] C. Greif and K. Urban. *Decay of the Kolmogorov N-width for wave problems.* Applied Mathematics Letters **96**, 216–222 (2019).

[62] P. Buchfink, S. Glas and B. Haasdonk. *Symplectic model reduction of Hamiltonian systems on nonlinear manifolds and approximation with weakly symplectic autoencoder.* SIAM Journal on Scientific Computing **45**, A289–A311 (2023).

[33] B. Brantner and M. Kraus. *Symplectic autoencoders for Model Reduction of Hamiltonian Systems*, arXiv preprint arXiv:2312.10004 (2023).

Chapter 10

Neural Networks as Symplectic Integrators

In this chapter we use neural networks as symplectic integrators. We first show SympNets as an example of a neural network-based one-step method and then show linear symplectic transformers as an example of a symplectic multi-step method.

10.1 SympNets with `GeometricMachineLearning`

This section serves as a short introduction into using SympNets (see Chapter 8.5) with `GeometricMachineLearning`.

Loss function

The `FeedForwardLoss` is the default choice used in `GeometricMachineLearning` for training SympNets, this can however be changed or tweaked (see Chapter C.1).

Training a Harmonic Oscillator

Here we begin with a simple example, the harmonic oscillator, the Hamiltonian of which is

$$H : (q, p) \in \mathbb{R}^2 \mapsto \frac{1}{2}p^2 + \frac{1}{2}q^2 \in \mathbb{R}.$$

Here we take the ODE from `GeometricProblems` and integrate it with `GeometricIntegrators` [44]:

```
import GeometricProblems.HarmonicOscillator as ho
using GeometricIntegrators: ImplicitMidpoint, integrate

# the problem is the ODE of the harmonic oscillator
ho_problem = ho.hodeproblem(; timespan = 500)

# integrate the system
solution = integrate(ho_problem, ImplicitMidpoint())
```

We call `DataLoader` in order to conveniently handle the data:

```
dl_raw = DataLoader(solution; suppress_info = true)
```

We have not yet specified the type and backend that we want to use. We do this now:

```
# specify the data type and the backend
type = Float16
backend = CPU()

# we can then make a new instance of `DataLoader` with this backend and type.
dl = DataLoader(dl_raw, backend, type)
```

Next we specify the architectures[1]:

```
const upscaling_dimension = 2
const nhidden = 1
const activation = tanh
const n_layers = 4 # number of layers for the G-SympNet
const depth = 4 # number of layers in each linear block in the LA-SympNet

# calling G-SympNet architecture
gsympnet = GSympNet(dl; upscaling_dimension = upscaling_dimension,
                        n_layers = n_layers,
                        activation = activation)

# calling LA-SympNet architecture
lasympnet = LASympNet(dl;   nhidden = nhidden,
                            activation = activation,
                            depth = depth)

# initialize the networks
la_nn = NeuralNetwork(lasympnet, backend, type)
g_nn = NeuralNetwork(gsympnet, backend, type)
```

If we want to obtain information on the number of parameters in a neural network, we can do that with the function `parameterlength`. For the `LASympNet`:

```
parameterlength(la_nn.model)
```

```
14
```

And for the `GSympNet`:

```
parameterlength(g_nn.model)
```

[1] `GeometricMachineLearning` provides useful defaults for all parameters, but they can still be specified manually; which is what we are doing here. Details on these parameters can be found in the docstrings for `GSympNet` and `LASympNet`.

```
12
```

Remark 10.1.1. We can also specify whether we would like to start with a layer that changes the q-component or one that changes the p-component. This can be done via the keywords `init_upper` for the `GSympNet`, and `init_upper_linear` and `init_upper_act` for the `LASympNet`.

We have to define an optimizer (see Chapter 6.1) which will be used in training of the SympNet. In this example we use Adam (see Chapter 6.1):

```
# set up optimizer; for this we first need to specify the optimization method
opt_method = AdamOptimizer(type)
# we then call the optimizer struct which allocates the cache
la_opt = Optimizer(opt_method, la_nn)
g_opt = Optimizer(opt_method, g_nn)
```

We can now perform the training of the neural networks:

```
# determine the batch size (the number of samples in one batch)
const batch_size = 16

batch = Batch(batch_size)

# number of training epochs
const nepochs = 100

# perform training (returns array that contains the total loss for each training step)
g_loss_array = g_opt(g_nn, dl, batch, nepochs; show_progress = false)
la_loss_array = la_opt(la_nn, dl, batch, nepochs; show_progress = false)
```

We plot the training errors against the epoch (here the y-axis is in log-scale):

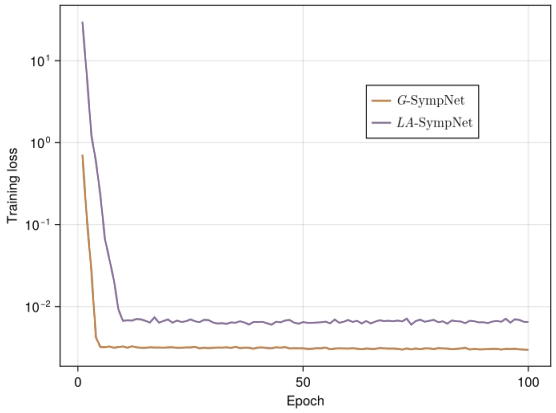


Figure 10.1: Training loss for SympNet.

Now we can make a prediction. We compare the initial data with a prediction starting from the same phase space point using the function `GeometricMachineLearning.iterate`:

```
ics = (q=dl.input.q[:, 1, 1], p=dl.input.p[:, 1, 1])

steps_to_plot = 200

#predictions
la_trajectory = iterate(la_nn, ics; n_points = steps_to_plot)
g_trajectory =  iterate(g_nn, ics; n_points = steps_to_plot)
```

We now plot the result:

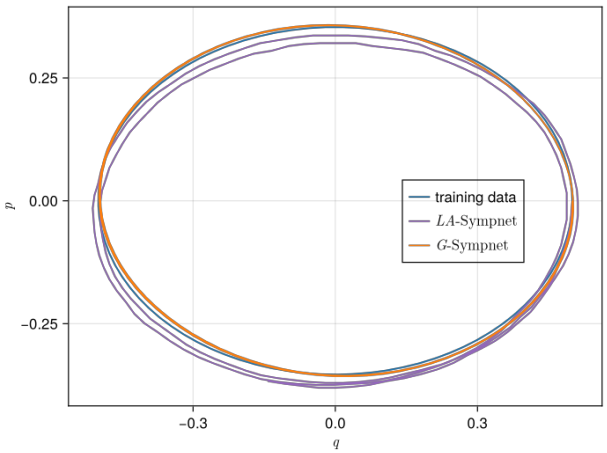


Figure 10.2: Trajectory prediction with the SympNet.

We see that `GSympNet` outperforms `LASympNet` on this problem; the blue line (reference) and the orange line (G-SympNet) are in fact almost indistinguishable.

Remark 10.1.2. We have actually never observed a scenario in which the LA-SympNet can outperform the G-SympNet. The G-SympNet seems usually trains faster, is more accurate and less sensitive to the chosen hyperparameters and initialization of the weights. They are also more straightforward to interpret. We therefore use the G-SympNet as a basis for the *linear symplectic transformer.*

Comparison with a ResNet

We want to show the advantages of using a SympNet over a standard ResNet that is not symplectic. For this we make a ResNet with a similar size of parameters as the two SympNets have:

```
resnet = ResNet(dl, n_layers ÷ 2; activation = activation)

rn_nn = NeuralNetwork(resnet, backend, type)

parameterlength(rn_nn)
```

```
18
```

We now train the network ...

```
rn_opt = Optimizer(opt_method, rn_nn)

rn_loss_array = rn_opt(rn_nn, dl, batch, nepochs; show_progress = false)
```

and plot the loss:

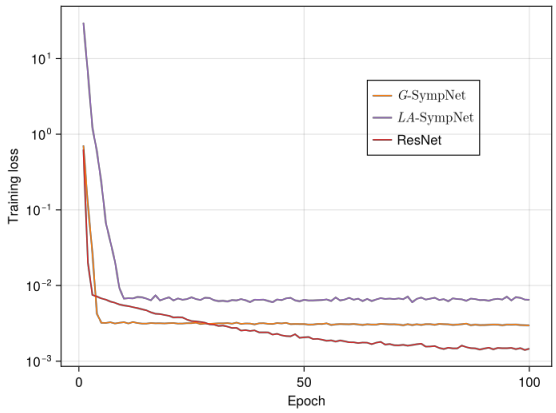


Figure 10.3: Training loss for SympNet and ResNet.

And we see that the loss is significantly lower than for both the LA-SympNet and the G-SympNet. We can also plot the prediction:

```
rn_trajectory = iterate(rn_nn, ics; n_points = steps_to_plot)
```

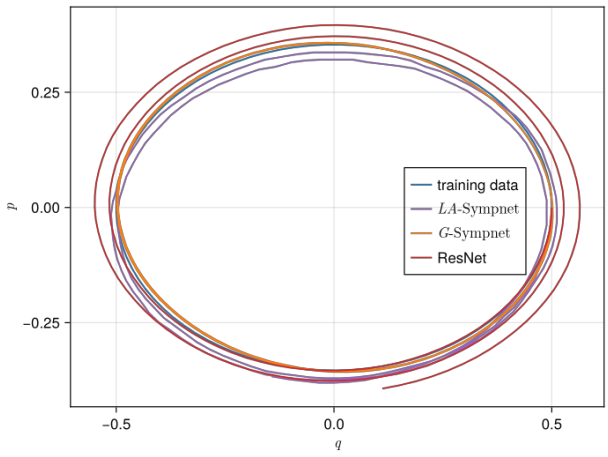


Figure 10.4: Trajectory prediction with SympNet and ResNet.

We see that the ResNet is slowly gaining energy which consitutes unphysical behaviour. If we let this simulation run for even longer, this effect gets more pronounced:

```
steps_to_plot = 400

#predictions
la_trajectory = iterate(la_nn, ics; n_points = steps_to_plot)
g_trajectory =  iterate(g_nn, ics; n_points = steps_to_plot)
rn_trajectory = iterate(rn_nn, ics; n_points = steps_to_plot)
```

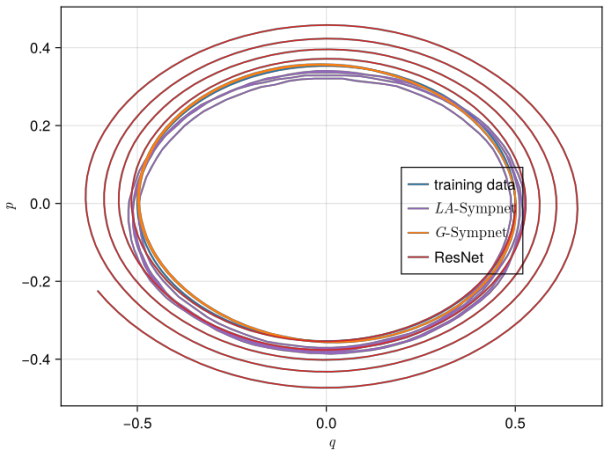


Figure 10.5: Comparison of long-time predictions for SympNet and ResNet.

The behavior the ResNet exhibits is characteristic of integration schemes that do not preserve structure: the error in a single time step can be made very small, but for long-time simulations one typically has to consider symplecticity or other properties. Also note that the curves produced by the LA-SympNet and the G-SympNet are closed (or nearly closed). This is a property of symplectic maps in two dimensions that is preserved by construction [3].

10.2 Linear Symplectic Transformer

In this section we compare the linear symplectic transformer (see Chapter 8.9) to the standard transformer (see Chapter 8.7). The example we treat here is the *coupled harmonic oscillator*:

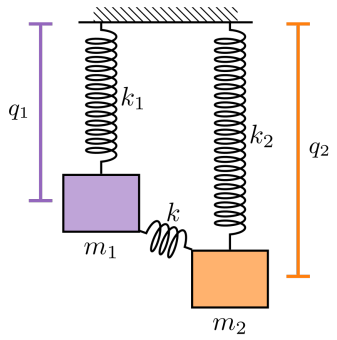


Figure 10.6: Visualization of the coupled harmonic oscillator.

It is a Hamiltonian system (see Chapter 3.1) with

$$H(q_1, q_2, p_1, p_2) = \frac{q_1^2}{2m_1} + \frac{q_2^2}{2m_2} + k_1\frac{q_1^2}{2} + k_2\frac{q_2^2}{2} + k\sigma(q_1)\frac{(q_2 - q_1)^2}{2},$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the sigmoid activation function. The system parameters are:

- k_1: spring constant belonging to m_1,
- k_2: spring constant belonging to m_2,
- m_1: mass 1,
- m_2: mass 2,
- k: coupling strength between the two masses.

To demonstrate the efficacy of the linear symplectic transformer here we will leave the parameters fixed but alter the initial conditions[1]:

```
using GeometricProblems.CoupledHarmonicOscillator: hodeensemble, default_parameters

const timestep = .3
const n_init_con = 5

# ensemble problem
ep = hodeensemble([rand(2) for _ in 1:n_init_con], [rand(2) for _ in 1:n_init_con]; timestep =
↪  timestep)
dl = DataLoader(integrate(ep, ImplicitMidpoint()); suppress_info = true)
```

We now define the architectures and train them:

```
const seq_length = 4
const batch_size = 1024
const n_epochs = 2000

arch_standard = StandardTransformerIntegrator(dl.input_dim; n_heads = 2,
                                                            L = 1,
                                                            n_blocks = 2)
arch_symplectic = LinearSymplecticTransformer(  dl.input_dim,
                                                seq_length; n_sympnet = 2,
                                                L = 1,
                                                upscaling_dimension = 2 * dl.input_dim)
arch_sympnet = GSympNet(dl.input_dim;   n_layers = 4,
                                        upscaling_dimension = 2 * dl.input_dim)

nn_standard = NeuralNetwork(arch_standard)
nn_symplectic = NeuralNetwork(arch_symplectic)
nn_sympnet = NeuralNetwork(arch_sympnet)
```

[1] We here use the implementation of the coupled harmonic oscillator from GeometricProblems.

```
o_method = AdamOptimizerWithDecay(n_epochs, Float64)

o_standard = Optimizer(o_method, nn_standard)
o_symplectic = Optimizer(o_method, nn_symplectic)
o_sympnet = Optimizer(o_method, nn_sympnet)

batch = Batch(batch_size, seq_length)
batch2 = Batch(batch_size)

loss_array_standard = o_standard(nn_standard, dl, batch, n_epochs; show_progress = false)
loss_array_symplectic = o_symplectic(nn_symplectic, dl, batch, n_epochs; show_progress = false)
loss_array_sympnet = o_sympnet(nn_sympnet, dl, batch2, n_epochs; show_progress = false)
```

And the corresponding training losses look as follows:

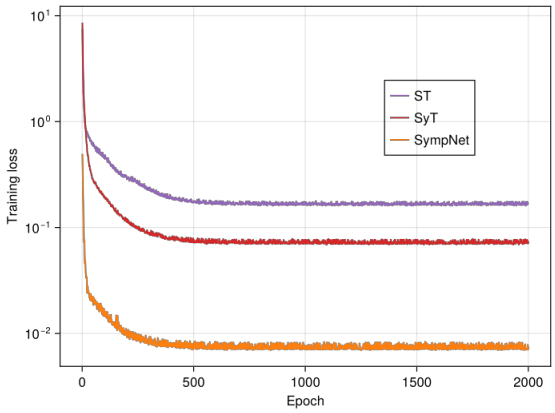


Figure 10.7: Training loss for the different networks.

We further evaluate a trajectory with the trained networks for thirty time steps:

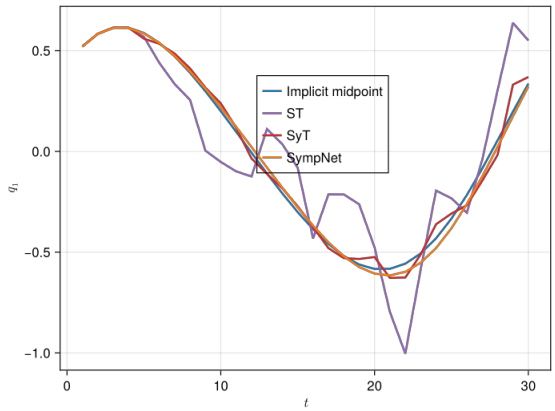


Figure 10.8: Validation of the different networks.

We can see that the standard transformer is not able to stay close to the trajectory coming from implicit midpoint very well. The linear symplectic transformer outperforms the standard transformer while needing fewer parameters:

```
parameterlength(nn_standard), parameterlength(nn_symplectic), parameterlength(nn_sympnet)
```

```
(108, 84, 64)
```

We also note that the linear symplectic transformer can be made *nonlinear* by using symplectic attention (see Chapter 7.6) layers.

Chapter Summary

In this chapter we demonstrated the efficacy of neural network-based symplectic integrators. We showed two examples: SympNets as an example of a symplectic one-step method and linear symplectic transformers as an example of a symplectic multi-step method. We compared the two different SympNet architectures (LA-SympNets and G-SympNets) with a standard ResNet and gave an example where symplecticity is important to achieve long-term stability: the ResNet was shown to fail on longer time scales even though it outperforms the LA-SympNet on smaller ones; this is because the ResNet does not have any structure encoded into it. The linear symplectic transformer was compared to the standard transformer and a SympNet (on the example of a complex coupled harmonic oscillator) and shown to outperform these two networks.

Of the two symplectic neural networks shown here the linear symplectic transformer constitutes a novel neural network architecture which was introduced in this dissertation.

Chapter 11

Transformers with Structure

In this chapter we discuss examples of improving transformers by imbuing them with structure[1]. We show two examples: an example of using the vision transformer where we put orthogonality constraints on some of the weights (which effectively leads to manifold optimization) and using the volume-preserving transformer to learn the dynamics of a rigid body. At the end we further compare two different approaches to realizing the volume-preserving transformer.

11.1 MNIST Tutorial

In this tutorial we show how we can use `GeometricMachineLearning` to build a vision transformer and apply it for MNIST [32], while also putting some of the weights on a manifold. This is also the result presented in [26].

We get the dataset from `MLDatasets`. Before we use it we allocate it on gpu with `cu` from `CUDA.jl` [89]:

```
using MLDatasets
using CUDA
train_x, train_y = MLDatasets.MNIST(split=:train)[:]
test_x, test_y = MLDatasets.MNIST(split=:test)[:]

train_x = train_x |> cu
train_y = train_y |> cu
test_x = test_x |> cu
test_y = test_y |> cu
```

Next we call `DataLoader` on these data. For this we first need to specify a *patch length*[1].

Remark 11.1.1. In order to apply the transformer to a data set we should typically cast these data into a *time series format*. MNIST images are pictures with 28×28 pixels. Here we cast these images into *time series* of length 16, so one image is represented by a matrix $\in \mathbb{R}^{49\times16}$.

```
const patch_length = 7
dl = DataLoader(train_x, train_y, patch_length = patch_length; suppress_info = true)
```

[1] Technically the linear symplectic transformer from the previous chapter could also be included here, but because of the very severe modification/limitation of the attention layer we performed there, we decided against it.

[1] When `DataLoader` is called this way it uses `split_and_flatten` internally.

Here we called `DataLoader` on a tensor and a vector of integers (targets) as input. `DataLoader` automatically converts the data to the correct input format for easy handling. This is visualized below:

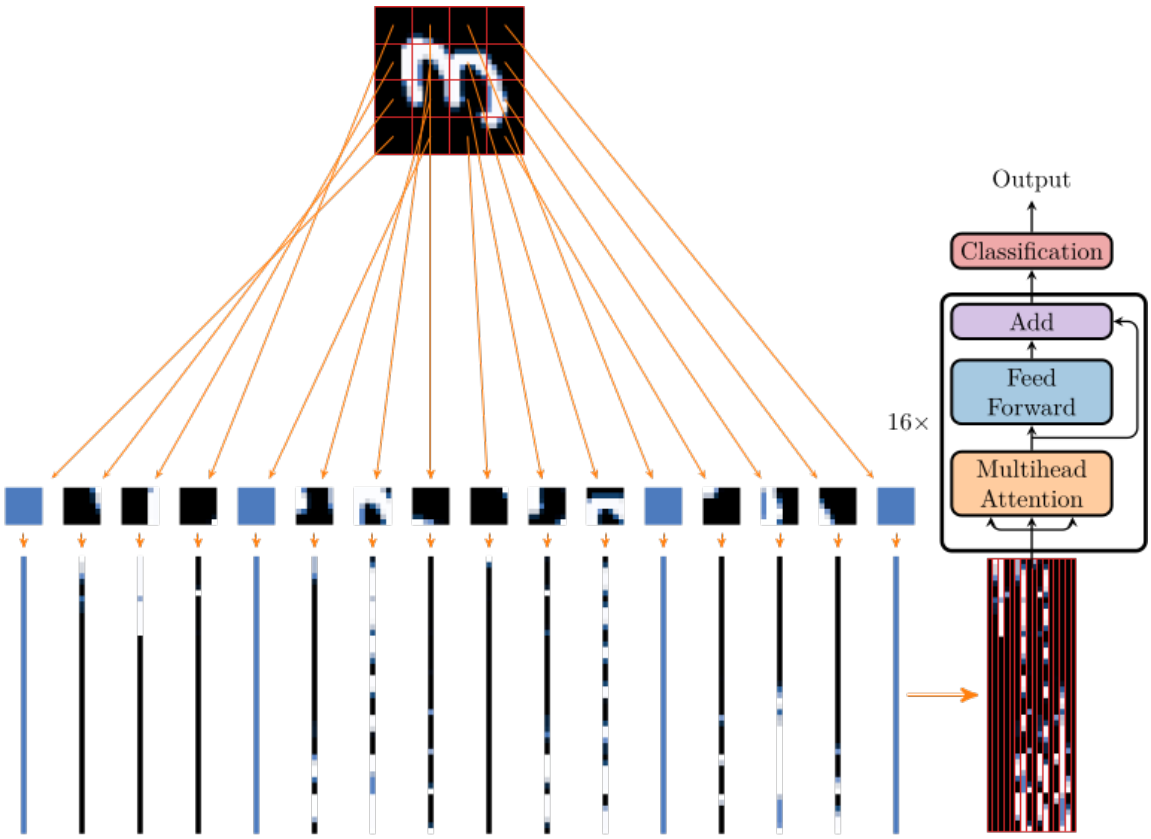


Figure 11.1: Visualization of how the data are preprocessed. An image is first split and then flattened.

Internally `DataLoader` calls `split_and_flatten` which splits each image into a number of *patches* according to the keyword arguments `patch_length` and `number_of_patches`. We also load the test data with `DataLoader`:

```
dl_test = DataLoader(test_x, test_y, patch_length=patch_length)
```

```
[ Info: You provided a tensor and a vector as input. This will be treated as a classification
↪  problem (MNIST). Tensor axes: (i) & (ii) image axes and (iii) parameter dimension.
```

We next define the model with which we want to train:

```
const n_heads = 7
const L = 16
const add_connection = false
```

```
model1 = ClassificationTransformer(dl;
                                   n_heads = n_heads,
                                   L = L,
                                   add_connection = add_connection,
                                   Stiefel = false)
model2 = ClassificationTransformer(dl;
                                   n_heads = n_heads,
                                   L = L,
                                   add_connection = add_connection,
                                   Stiefel = true)
```

Here we have chosen a `ClassificationTransformer`, i.e. a composition of a specific number of transformer layers composed with a classification layer. We also set the *Stiefel option* to `true`, i.e. we are optimizing on the Stiefel manifold.

We now have to initialize the neural network weights. This is done with the constructor for `NeuralNetwork`:

```
backend = GeometricMachineLearning.networkbackend(dl)
T = eltype(dl)
nn1 = NeuralNetwork(model1, backend, T)
nn2 = NeuralNetwork(model2, backend, T)
```

We still have to initialize the optimizers:

```
const batch_size = 2048
const n_epochs = 500
# an instance of batch is needed for the optimizer
batch = Batch(batch_size, dl)

opt1 = Optimizer(AdamOptimizer(T), nn1)
opt2 = Optimizer(AdamOptimizer(T), nn2)
```

And with this we can finally perform the training:

```
loss_array1 = opt1(nn1, dl, batch, n_epochs, FeedForwardLoss())
loss_array2 = opt2(nn2, dl, batch, n_epochs, FeedForwardLoss())
```

We furthermore optimize the second neural network (with weights on the manifold) with the `GradientOptimizer` and the `MomentumOptimizer`:

```
nn3 = NeuralNetwork(model2, backend, T)
nn4 = NeuralNetwork(model2, backend, T)

opt3 = Optimizer(GradientOptimizer(T(0.001)), nn3)
opt4 = Optimizer(MomentumOptimizer(T(0.001), T(0.5)), nn4)
```

For training we use the same data, the same batch and the same number of epochs:

```
loss_array3 = opt3(nn3, dl, batch, n_epochs, FeedForwardLoss())
loss_array4 = opt4(nn4, dl, batch, n_epochs, FeedForwardLoss())
```

And we get the following result:

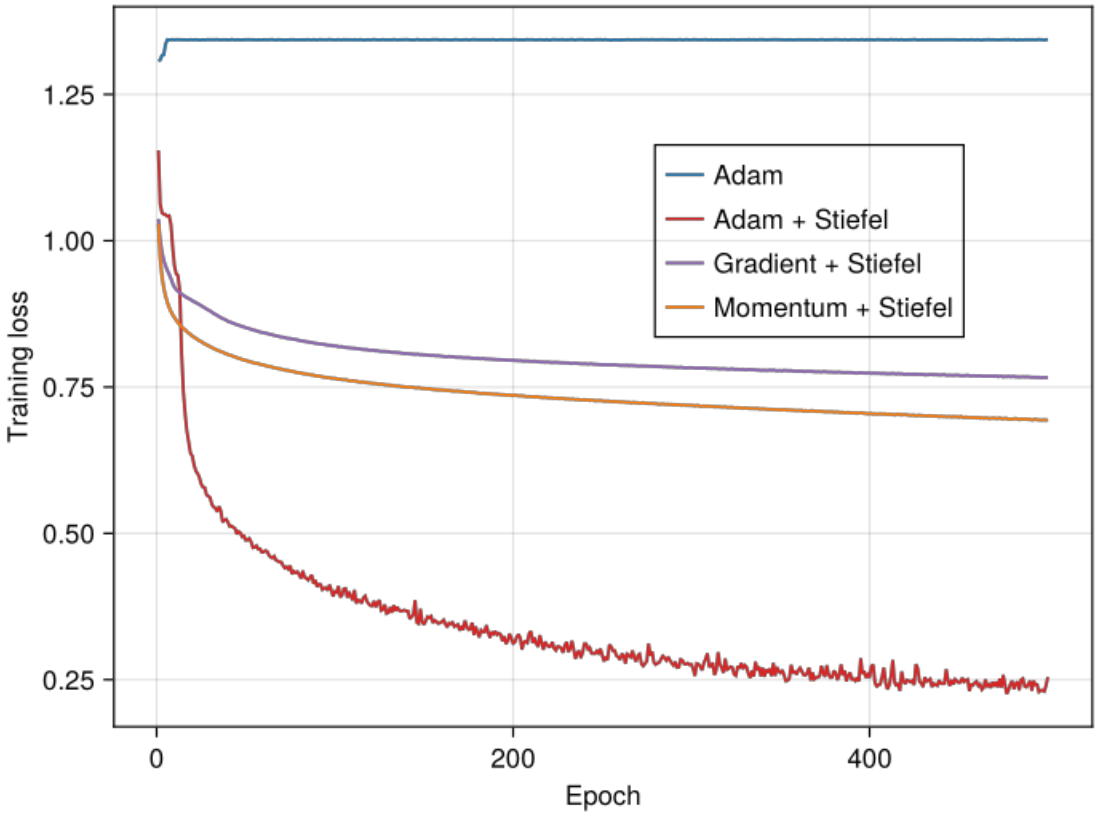


Figure 11.2: Comparison between the standard Adam optimizer (blue), the Adam optimizer with weights on the Stiefel manifold (purple), the gradient optimizer with weights on the Stiefel manifold (purple) and the momentum optimizer with weights on the Stiefel manifold (orange).

Remark 11.1.2. We see that the loss value for the Adam optimizer without parameters on the Stiefel manifold is stuck at around 1.34 which means that it *always predicts the same value*. So in 1 out of ten cases we have error 0 and in 9 out of ten cases we have error $\sqrt{2}$, giving

$$\sqrt{2\frac{9}{10}} = 1.342,$$

which is what we see in the error plot.

We can also call `GeometricMachineLearning.accuracy` to obtain the test accuracy instead of the training error:

```
(accuracy(nn1, dl_test), accuracy(nn2, dl_test), accuracy(nn3, dl_test), accuracy(nn4, dl_test))
```

```
(0.0974, 0.8613, 0.5518, 0.6351)
```

Remark 11.1.3. We note here that conventional convolutional neural networks and other vision transformers achieve much better accuracy on MNIST in a training time that is often shorter than what we presented here. Our aim here is not to outperform existing neural networks in terms of accuracy on image classification problems, but to demonstrate two things: (i) in many cases putting weights on the Stiefel manifold (which is a compact space) can enable training that would otherwise not be possible and (ii) as is the case with standard Adam, the manifold version also seems to achieve similar performance gain over the gradient and momentum optimizer. Both of these observations are demonstrated figure above.

Library Functions

`GeometricMachineLearning.split_and_flatten` – Function.

```
split_and_flatten(input::AbstractArray)::AbstractArray
```

Perform a preprocessing of an image into *flattened patches*.

This rearranges the input data so that it can easily be processed with a transformer.

Examples

Consider a matrix of size 6×6 which we want to divide into patches of size 3×3.

```
using GeometricMachineLearning

input = [ 1  2  3  4  5  6;
          7  8  9 10 11 12;
         13 14 15 16 17 18;
         19 20 21 22 23 24;
         25 26 27 28 29 30;
         31 32 33 34 35 36]

split_and_flatten(input; patch_length = 3, number_of_patches = 4)

# output

9×4 Matrix{Int64}:
  1  19   4  22
  7  25  10  28
 13  31  16  34
  2  20   5  23
  8  26  11  29
 14  32  17  35
  3  21   6  24
  9  27  12  30
 15  33  18  36
```

Here we see that `split_and_flatten`:

1. *splits* the original matrix into four 3×3 matrices and then
2. *flattens* each matrix into a column vector of size 9.

After this all the vectors are put together again to yield a 9×4 matrix.

Arguments

The optional keyword arguments are:

- `patch_length`: by default this is 7.
- `number_of_patches`: by default this is 16.

The sizes of the first and second axis of the output of `split_and_flatten` are

1. `path_length`^{2} and
2. `number_of_patches`.

source

`GeometricMachineLearning.accuracy` – Function.

```
accuracy(model, ps, dl)
```

Compute the accuracy of a neural network classifier.

This needs an instance of `DataLoader` that stores the *test data*.

source

```
accuracy(nn, dl)
```

Compute the accuracy of a neural network classifier.

This is like `accuracy(::Chain, ::Tuple, ::DataLoader)`, but for a `NeuralNetwork`.

source

`GeometricMachineLearning.onehotbatch` – Function.

```
onehotbatch(target)
```

Performs a one-hot-batch encoding of a vector of integers: $input \in \{0, 1, \dots, 9\}^\ell$.

The output is a tensor of shape $10 \times 1 \times \ell$.

If the input is 0, this function produces:

$$0 \mapsto \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix}^T.$$

In more abstract terms: $i \mapsto e_i$.

Examples

```
using GeometricMachineLearning: onehotbatch

target = [0]
onehotbatch(target)

# output

10×1×1 Array{Int64, 3}:
[:, :, 1] =
 1
 0
 0
 0
 0
 0
 0
 0
 0
 0
```

source

`GeometricMachineLearning.ClassificationLayer` – Type.

```
ClassificationLayer(input_dim, output_dim, activation)
```

Make an instance of `ClassificationLayer`.

`ClassificationLayer` takes a matrix as an input and returns a vector that is used for classification.

It does:

$$X \mapsto \sigma(\texttt{compute\_vector}(AX)),$$

where X is a matrix and `compute_vector` specifices how this matrix is turned into a vector.

`compute_vector` can be specified with the keyword `average`.

Arguments

`ClassificationLayer` has the following optional keyword argument:

- `average:Bool=false`.

If this keyword argument is set to `true`, then the output is computed as

$$input \mapsto \frac{1}{N}\sum_{i=1}^{N}[\mathcal{NN}(input)]_{\bullet i}.$$

If set to `false` (the default) it picks the last column of the input.

Examples

```
using GeometricMachineLearning

l = ClassificationLayer(2, 2, identity; average = true)
ps = (weight = [1 0; 0 1], )

input = [1 2 3; 1 1 1]

l(input, ps)

# output

2×1 Matrix{Float64}:
 2.0
 1.0
```

```
using GeometricMachineLearning

l = ClassificationLayer(2, 2, identity; average = false)
ps = (weight = [1 0; 0 1], )

input = [1 2 3; 1 1 1]

l(input, ps)

# output

2×1 Matrix{Int64}:
 3
 1
```

source

11.2 The Volume-Preserving Transformer for the Rigid Body

Here we train a volume-preserving feedforward neural network (see Chapter 8.6), a standard transformer (see Chapter 8.7) and a volume-preserving transformer (see Chapter 8.8) on a rigid body [3, 42]. These are also the results presented in [30]. The ODE that describes the rigid body is:

$$\frac{d}{dt}\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} Az_2z_3 \\ Bz_1z_3 \\ Cz_1z_2 \end{pmatrix}.$$

In the following we use $A = 1$, $B = 1/2$ and $C = -1/2$. For a derivation of this equation see [30].

We first generate the data. The initial conditions that we use are:

$$\texttt{ics} = \left\{ \begin{pmatrix} \sin(\alpha) \\ 0 \\ \cos(\alpha) \end{pmatrix}, \begin{pmatrix} 0 \\ \sin(\alpha) \\ \cos(\alpha) \end{pmatrix} : \alpha = 0.1{:}0.01{:}2\pi \right\}.$$

We build these initial conditions by concatenating $\texttt{ics}_1$ and $\texttt{ics}_2$:

```
const ics₁ = [[sin(val), 0., cos(val)] for val in .1:.01:(2*π)]
const ics₂ = [[0., sin(val), cos(val)] for val in .1:.01:(2*π)]
const ics = [ics₁..., ics₂...]
```

We now generate the data by integrating with:

```
const timestep = .2
const timespan = (0., 20.)
```

The rigid body is implemented in `GeometricProblems`:

```
using GeometricIntegrators: integrate, ImplicitMidpoint
using GeometricProblems.RigidBody: odeproblem, odeensemble, default_parameters

ensemble_problem = odeensemble(ics; timespan = timespan, timestep = timestep, parameters =
↪  default_parameters)
ensemble_solution = integrate(ensemble_problem, ImplicitMidpoint())
dl_cpu = DataLoader(ensemble_solution; suppress_info = true)
```

We plot the trajectories for some of the initial conditions to get and idea of what the data look like:

```
const n_trajectories_to_plot = 5
indices = Int.(ceil.(size(dl_cpu.input, 3) * rand(n_trajectories_to_plot)))
trajectories = [dl_cpu.input[:, :, index] for index in indices]
```

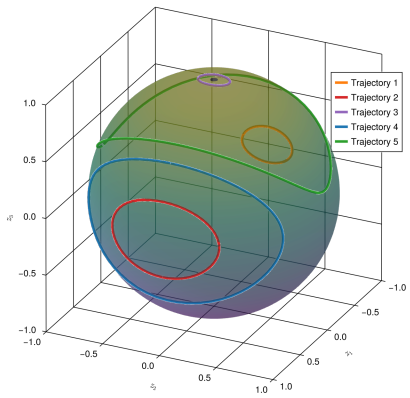


Figure 11.3: A sample of rigid body trajectories. This system has two conserved quantities.

The rigid body has two conserved quantities:

1. one conserved quantity is the Hamiltonian of the system (see Chapter 3.1): $H(z_1, z_2, z_3) = \frac{1}{2}\left(\frac{z_1^2}{I_1} + \frac{z_2^2}{I_2} + \frac{z_3^2}{I_3}\right)$,
2. the second one is the quadratic invariant: $I(z_1, z_2, z_3) = z_1^2 + z_2^2 + z_3^2$.

The coefficients I_1, I_2 and I_3 can be obtained through

$$A = \frac{I_2 - I_3}{I_2 I_3},$$
$$B = \frac{I_3 - I_1}{I_3 I_1},$$
$$C = \frac{I_1 - I_2}{I_1 I_2}.$$

The second conserved invariant $I(\cdot, \cdot, \cdot)$ is visualized through the sphere in the figure above. The conserved Hamiltonian is the reason for why the curves are closed.

The rigid body has Poisson structure [3], but does not have canonical Hamiltonian structure. We can thus not use SympNets (see Chapter 8.5) or symplectic transformers (see Chapter 8.9) here, but the ODE is clearly divergence-free. We use this to demonstrate the efficacy of the volume-preserving transformer (see Chapter 8.8). We set up our networks:

```
# hyperparameters concerning the architectures
const sys_dim = size(dl_cpu.input, 1)
const n_heads = 1
const L = 3 # transformer blocks
const activation = tanh
const resnet_activation = tanh
const n_linear = 1
const n_blocks = 2
const skew_sym = false
const seq_length = 3

arch_vpff = VolumePreservingFeedForward(sys_dim, n_blocks * L, n_linear, resnet_activation)
arch_vpt = VolumePreservingTransformer(sys_dim, seq_length;
                                       n_blocks = n_blocks,
                                       n_linear = n_linear,
                                       L = L,
                                       activation = resnet_activation,
                                       skew_sym = skew_sym)
arch_st = StandardTransformerIntegrator(sys_dim;    n_heads = n_heads,
                                                    transformer_dim = sys_dim,
                                                    n_blocks = n_blocks,
                                                    L = L,
                                                    resnet_activation = resnet_activation,
                                                    add_connection = false)
```

Note that we set the keyword `skew_sym` to `false` here. This is different from what we did in [30], where it was set to true[1]. We allocate the networks on GPU:

[1] A detailed discussion of the consequences of setting this keyword is presented as a separate example (see Chapter 11.3).

```
using CUDA
backend = CUDABackend()
```

```
T = Float32

dl = DataLoader(dl_cpu, backend, T)
nn_vpff = NeuralNetwork(arch_vpff, backend, T)
nn_vpt = NeuralNetwork(arch_vpt, backend, T)
nn_st = NeuralNetwork(arch_st, backend, T)

(parameterlength(nn_vpff), parameterlength(nn_vpt), parameterlength(nn_st))
```

```
(135, 180, 189)
```

We now train the various networks. For this we use `AdamOptimizerWithDecay`:

```
const n_epochs = 500000
const batch_size = 16384
const feedforward_batch = Batch(batch_size)
const transformer_batch = Batch(batch_size, seq_length, seq_length)
const opt_method = AdamOptimizerWithDecay(n_epochs, T; η₁ = 1e-2, η₂ = 1e-6)

o_vpff = Optimizer(opt_method, nn_vpff)
o_vpt = Optimizer(opt_method, nn_vpt)
o_st = Optimizer(opt_method, nn_st)
```

```
o_vpff(nn_vpff, dl, feedforward_batch, n_epochs)
o_vpt(nn_vpt, dl, transformer_batch, n_epochs)
o_st(nn_st, dl, transformer_batch, n_epochs)
```

After the networks have been trained we map the parameters to cpu:

```
const mtc = GeometricMachineLearning.map_to_cpu
nn_vpff = mtc(nn_vpff)
nn_vpt = mtc(nn_vpt)
nn_st = mtc(nn_st)
```

After this we use `iterate` to obtain predicted orbits:

```
ics_val₁ = [0., sin(0.9), cos(0.9 + π)]
ics_val₂ = [0., sin(1.1), cos(1.1)]
const t_validation = 120

function produce_trajectory(ics_val)
    problem = odeproblem(ics_val;   timespan = (0, t_validation),
                                    timestep = timestep,
```

```
                                    parameters = default_parameters)
    solution = integrate(problem, ImplicitMidpoint())
    trajectory = Float32.(DataLoader(solution; suppress_info = true).input)
    nn_vpff_solution = iterate(nn_vpff, trajectory[:, 1];
                                            n_points = Int(floor(t_validation / timestep)) + 1)
    nn_vpt_solution = iterate(nn_vpt, trajectory[:, 1:seq_length];
                                            n_points = Int(floor(t_validation / timestep)) + 1)
    nn_st_solution = iterate(nn_st, trajectory[:, 1:seq_length];
                                            n_points = Int(floor(t_validation / timestep)) + 1)
    trajectory, nn_vpff_solution, nn_vpt_solution, nn_st_solution
end

trajectory₁, nn_vpff_solution₁, nn_vpt_solution₁, nn_st_solution₁ = produce_trajectory(ics_val₁)
trajectory₂, nn_vpff_solution₂, nn_vpt_solution₂, nn_st_solution₂ = produce_trajectory(ics_val₂)
```

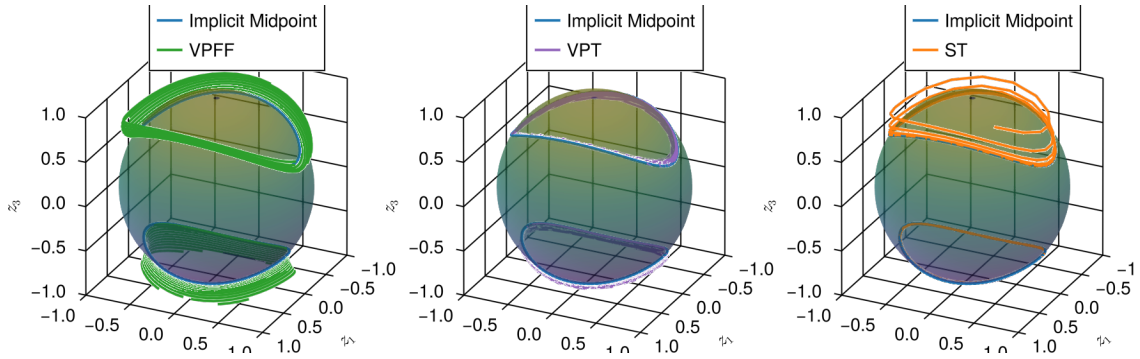


Figure 11.4: Evaluation of the three different networks for an interval (0, 120).

We can see that the volume-preserving transformer performs much better than the volume-preserving feedforward neural network and the standard transformer. It is especially noteworthy that its curves stick to the sphere at all times, which is not the case for the standard transformer. We also see that the standard transformer seems to perform better for one of the curves shown, but completely fails for the other. Why this is should be further investigated.

We also compare the times it takes to integrate the system with (i) implicit midpoint, (ii) the volume-preserving transformer and (iii) the standard transformer:

```
@time "Implicit Midpoint" solution = integrate(problem, ImplicitMidpoint())
trajectory = Float32.(DataLoader(solution; suppress_info = true).input)
@time "VPT" iterate(nn_vpt, trajectory[:, 1:seq_length]; n_points = Int(floor(t_validation /
↪  timestep)) + 1)
@time "ST" iterate(nn_st, trajectory[:, 1:seq_length]; n_points = Int(floor(t_validation /
↪  timestep)) + 1)
```

```
Implicit Midpoint: 0.020943 seconds (354.16 k allocations: 16.510 MiB)
VPT: 0.001891 seconds (113.61 k allocations: 4.510 MiB)
ST: 0.000496 seconds (25.41 k allocations: 1.339 MiB)
```

Here we see that standard transformer is the fastest, followed by the volume-preserving transformers. Both neural network integrators are however faster than implicit midpoint (as evaluating them is completely explicit). The ODE we treated here is a very simple one. For more complicated cases (see Chapter 9.1) the speed-up we gain by using neural networks can be up to a factor 1000.

11.3 Comparing Different `VolumePreservingAttention` Mechanisms

In the section on volume-preserving attention (see Chapter 7.3) we mentioned two ways of computing volume-preserving attention: one where we compute the correlations with a skew-symmetric matrix and one where we compute the correlations with an arbitrary matrix. Here we compare the two approaches. When calling the `VolumePreservingAttention` layer we can specify whether we want to use the skew-symmetric or the arbitrary weighting by setting the keyword `skew_sym = true` and `skew_sym = false` respectively.

In here we demonstrate the differences between the two approaches for computing correlations. For this we first generate a training set consisting of two collections of curves: (i) sine curves and (ii) cosine curve.

```
sine_cosine = zeros(1, 1000, 2)
sine_cosine[1, :, 1] .= sin.(0.:.1:99.9)
sine_cosine[1, :, 2] .= cos.(0.:.1:99.9)

const T = Float16
const dl = DataLoader(T.(sine_cosine); suppress_info = true)
```

The third axis (i.e. the parameter axis) has length two, meaning we have two different kinds of curves, i.e. the data look like this:

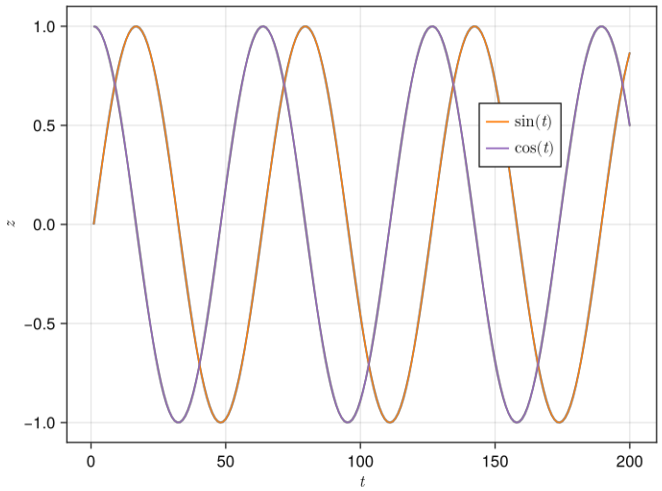


Figure 11.5: The data we treat here contains two different curves.

We want to train a single neural network on both these curves. We already noted before (see Chapter 8.9) that a simple feedforward neural network cannot do this. Here we compare three networks which are of the following form:

$$\texttt{network} = \mathcal{NN}_d \circ \Psi \circ \mathcal{NN}_u,$$

where $\mathcal{NN}_u$ refers to a neural network that scales up (see Chapter 8.7) and $\mathcal{NN}_d$ refers to a neural network that scales down. The up and down scaling is done with simple dense layers:

$$\mathcal{NN}_u(x) = \tanh(a_u x + b_u) \text{ and } \mathcal{NN}_d(x) = a_d^T x + b_d,$$

where $a_u, b_u, a_d \in \mathbb{R}^{\text{ud}}$ and b_d is a scalar. `ud` refers to *upscaling dimension*. For Ψ we consider three different choices:

1. a volume-preserving attention with skew-symmetric weighting,
2. a volume-preserving attention with arbitrary weighting,
3. an identity layer.

We further choose a sequence length 5 (i.e. the network always sees the last 5 time steps) and always predict one step into the future (i.e. the prediction window is set to 1):

```
const seq_length = 3
const prediction_window = 1
const upscale_dimension_1 = 2

function set_up_networks(upscale_dimension::Int = upscale_dimension_1)
    model_skew = Chain( Dense(1, upscale_dimension, tanh),
                        VolumePreservingAttention(upscale_dimension, seq_length; skew_sym = true),
                        Dense(upscale_dimension, 1, identity; use_bias = true)
                        )

    model_arb  = Chain( Dense(1, upscale_dimension, tanh),
                        VolumePreservingAttention(upscale_dimension, seq_length; skew_sym = false),
                        Dense(upscale_dimension, 1, identity; use_bias = true)
                        )

    model_comp = Chain( Dense(1, upscale_dimension, tanh),
                        Dense(upscale_dimension, 1, identity; use_bias = true)
                        )

    nn_skew = NeuralNetwork(model_skew, CPU(), T)
    nn_arb  = NeuralNetwork(model_arb,  CPU(), T)
    nn_comp = NeuralNetwork(model_comp, CPU(), T)

    nn_skew, nn_arb, nn_comp
end

nn_skew, nn_arb, nn_comp = set_up_networks()
```

We expect the third network to not be able to learn anything useful since it cannot resolve time series data: a regular feedforward network only ever sees one datum at a time.

Next we train the networks (here we pick a batch size of 30 and train for 1000 epochs):

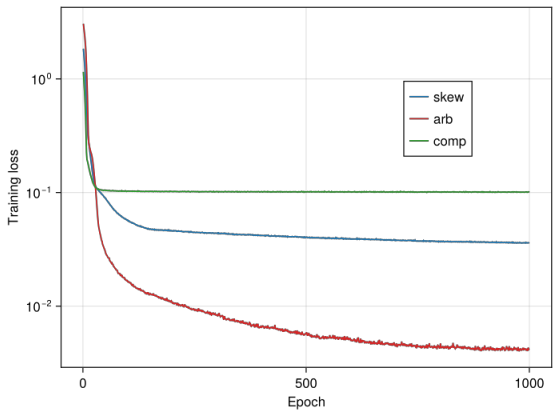


Figure 11.6: The training losses for the three networks.

Looking at the training errors, we can see that the network with the skew-symmetric weighting is stuck at a relatively high error rate, whereas the loss for the network with the arbitrary weighting is decreasing to a significantly lower level. The feedforward network without the attention mechanism is not able to learn anything useful (as was expected).

Before we can use the trained neural networks for prediction we have to make them `TransformerIntegrator`s or `NeuralNetworkIntegrator`s[1]:

```
initial_condition = dl.input[:, 1:seq_length, 2]

function make_networks_neural_network_integrators(nn_skew, nn_arb, nn_comp)
    nn_skew = NeuralNetwork(GeometricMachineLearning.DummyTransformer(seq_length),
                            nn_skew.model,
                            params(nn_skew),
                            CPU())
    nn_arb  = NeuralNetwork(GeometricMachineLearning.DummyTransformer(seq_length),
                            nn_arb.model,
                            params(nn_arb),
                            CPU())
    nn_comp = NeuralNetwork(GeometricMachineLearning.DummyNNIntegrator(),
                            nn_comp.model,
                            params(nn_comp),
                            CPU())

    nn_skew, nn_arb, nn_comp
end

nn_skew, nn_arb, nn_comp = make_networks_neural_network_integrators(nn_skew, nn_arb, nn_comp)
```

[1] Here we have to use the architectures `GeometricMachineLearning.DummyTransformer` and `GeometricMachineLearning.DummyNNIntegrator` to reformulate the three neural networks defined here as `NeuralNetworkIntegrator`s or `TransformerIntegrator`s. These *dummy architectures* can be used if the user wants to specify new neural network integrators that are not yet defined in `GeometricMachineLearning`.

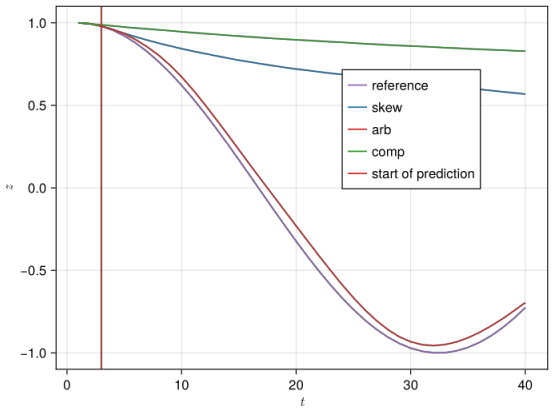


Figure 11.7: Comparing the two volume-preserving attention mechanisms for 40 points.

In the plot above we can see that the network with the arbitrary weighting performs much better; even though the red line does not fit the purple line perfectly, it manages to least qualitatively reflect the training data. We can also plot the predictions for longer time intervals:

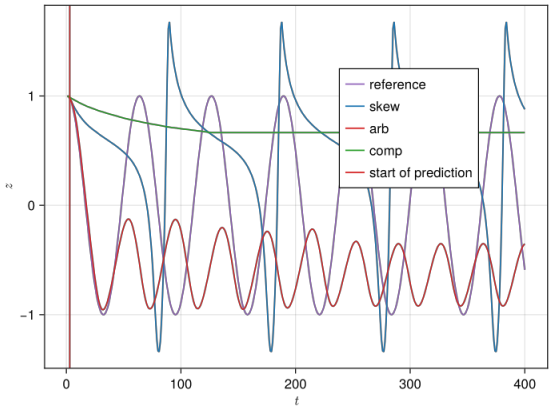


Figure 11.8: Comparing the two volume-preserving attention mechanisms for 400 points.

This advantage of the volume-preserving attention with arbitrary weighting may however be due to the fact that the skew-symmetric attention only has 3 learnable parameters, as opposed to 9 for the arbitrary weighting. We can increase the *upscaling dimension* and see how it affects the result:

```
const upscale_dimension_2 = 10

nn_skew, nn_arb, nn_comp = set_up_networks(upscale_dimension_2)

o_skew, o_arb, o_comp = set_up_optimizers(nn_skew, nn_arb, nn_comp)
```

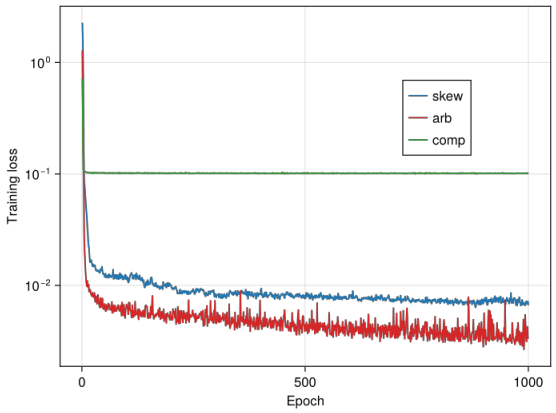


Figure 11.9: Training error for 40 points, but with an upscaling of ten.

```
initial_condition = dl.input[:, 1:seq_length, 2]

nn_skew, nn_arb, nn_comp = make_networks_neural_network_integrators(nn_skew, nn_arb, nn_comp)

fig_dark, fig_light, ax_dark, ax_light = produce_validation_plot(40, nn_skew, nn_arb, nn_comp)
```

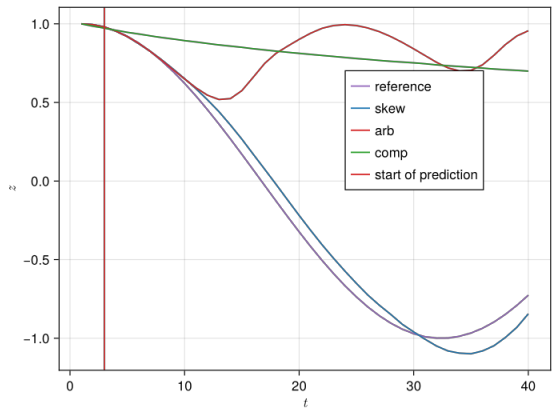


Figure 11.10: Prediction for 40 time steps.

And for a longer time interval:

```
fig_dark, fig_light, ax_dark, ax_light = produce_validation_plot(200, nn_skew, nn_arb, nn_comp)
```

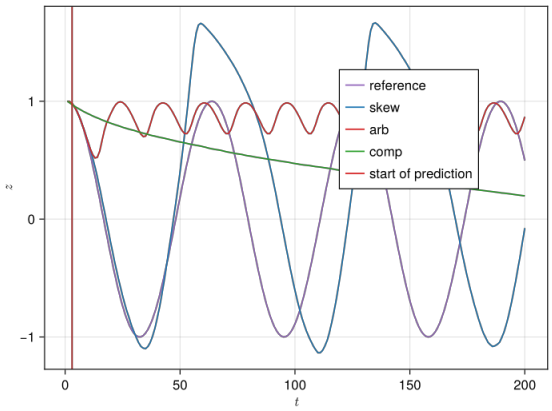


Figure 11.11: Prediction for 200 time steps.

Here we see that the arbitrary weighting quickly fails and the skew-symmetric weighting performs slightly better on longer time scales.

Library Functions

`GeometricMachineLearning.DummyNNIntegrator` – Type.

```
DummyNNIntegrator()
```

Make an instance of `DummyNNIntegrator`.

This *dummy architecture* can be used if the user wants to define a new `NeuralNetworkIntegrator`.

source

`GeometricMachineLearning.DummyTransformer` – Type.

```
DummyTransformer(seq_length)
```

Make an instance of `DummyTransformer`.

This *dummy architecture* can be used if the user wants to define a new `TransformerIntegrator`.

source

Chapter Summary

In this chapter we showed concrete examples of how to improve transformer neural networks by imbuing them with structure. The two examples we gave were (i) enforcing orthogonality constraints for some of the weights in a vision transformer (i.e. putting some of the weights on the *Stiefel manifold*) and (ii) using a volume-preserving transformer to learn the dynamics of a rigid body. In both cases we observed big improvements over the standard transformer that does not consider structure. In the first case the network was not able to learn anything if orthogonality constraints were not imposed and in the second case we obtained greatly improved long-time performance. At

the end we also compared two different approaches to designing the volume-preserving transformer: computing correlations based on a *skew-symmetric weighting* and *computing correlations based on an arbitrary weighting*. We saw that often the arbitrary weighting should be preferred over the skew-symmetric weighting, but the arbitrary weighting may also fail in other cases.

References

[26] B. Brantner. *Generalizing Adam To Manifolds For Efficiently Training Transformers*, arXiv preprint arXiv:2305.16901 (2023).

[30] B. Brantner, M. Kraus, Z. Li and G. de Romemont. *Volume-preserving transformers for learning time series data with structure*. ESAIM: Proceedings and Surveys **81**, 123–144 (2025).

Chapter 12

Learning Nonlinear Spaces

In this chapter we give another example of using the new neural network optimizer framework for manifolds, but this time for the *Grassmann manifold*. Here we suppose we are given data on a nonlinear subspace of $\mathbb{R}^N$ and want to sample additional data from this nonlinear subspace. We also utilize the Wasserstein distance and its gradient in this task.

12.1 Example of a Neural Network with a Grassmann Layer

Here we show how to implement a neural network that contains a layer whose weight is an element of the Grassmann manifold (see Chapter 2.10) and where this is useful. Recall that the Grassmann manifold $Gr(n, N)$ is the set of vector spaces of dimension n embedded in $\mathbb{R}^N$. So if we optimize on the Grassmann manifold, we optimize for an *ideal* n-dimensional vector space in the bigger space $\mathbb{R}^N$.

We visualize this:

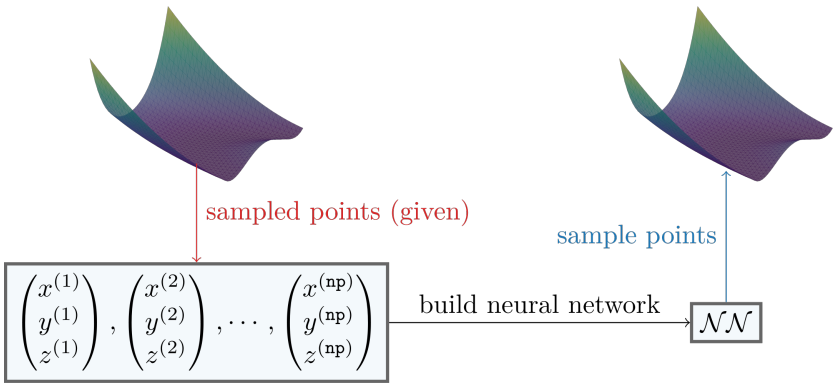


Figure 12.1: We can build a neural network that creates new samples from an unknown distribution.

So assume that we are given data on a nonlinear manifold:

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ \vdots \\ x_N^{(1)} \end{pmatrix}, \begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \\ \vdots \\ x_N^{(2)} \end{pmatrix}, \cdots, \begin{pmatrix} x_1^{(\mathsf{nop})} \\ x_2^{(\mathsf{nop})} \\ \vdots \\ x_N^{(\mathsf{nop})} \end{pmatrix} =: \mathcal{D} \subset \mathcal{M} \subset \mathbb{R}^N\text{“},$$

and $\dim(\mathcal{M}) = n$. We want to obtain additional data on this manifold by sampling from an n-dimensional distribution. Here we do not want to identify an isomorphism $\mathbb{R}^n \xrightarrow{\approx} \mathcal{M}$ (as is the case with autoencoders (see Chapter 4.1) for example), but find a mapping from a distribution on $\mathbb{R}^n$ to a distribution on $\mathcal{M}$. This is where the Grassmann manifold is useful: each element V of the Grassmann manifold is an n-dimensional subspace of $\mathbb{R}^N$ from which we can easily sample. We can then construct a mapping from this space V onto a space that contains the data points $\mathcal{D}$.

Remark 12.1.1. This example for learning weights on a Grassmann manifold also shows how `GeometricMachineLearning` can be used together with other packages. Here we use the *Wasserstein distance* from the package `BrenierTwoFluid` for example.

Graph of Rosenbrock Function as Example

Consider the following toy example: we want to sample from the graph of the (scaled) Rosenbrock function

$$f(x, y) = ((1 - x)^2 + 100(y - x^2)^2)/1000$$

without using the explicit form of the function during sampling. We show the graph of f for $(x, y) \in [-1.5, 1.5]^2$ in the following picture:

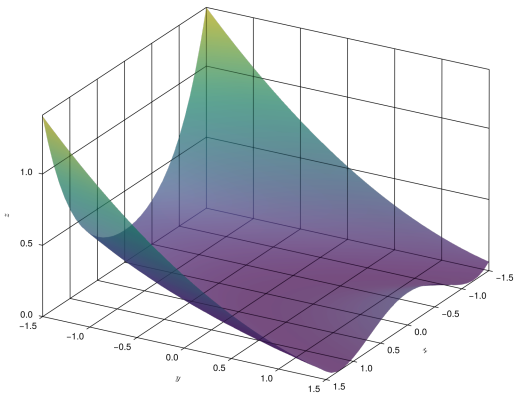


Figure 12.2: Graph of the Rosenbrock function.

We now build a neural network whose task it is to map a product of two Gaussians $\mathcal{N}(0, 1) \times \mathcal{N}(0, 1)$ onto the graph of the Rosenbrock function:

$$\Psi : \mathcal{N}(0, 1) \times \mathcal{N}(0, 1) \to \mathcal{W}(\{(x, y, z) : (x, y) \in [-1.5, 1.5] \times [-1.5, 1.5], z = f(x, y)\}) =: \tilde{\mathcal{W}},$$

where $\mathcal{W}$ is a distribution on the graph of f. For computing the loss between the two distributions, i.e. $\Psi(\mathcal{N}(0, 1) \times \mathcal{N}(0, 1))$ and $\mathcal{W}(f([-1.5, 1.5], [-1.5, 1.5]))$ we use the *Wasserstein distance*[1]. The Wasserstein distance can compute the distance between $\mathcal{N}(0, 1) \times \mathcal{N}(0, 1)$ and $\tilde{\mathcal{W}}$. For more details confer [90, 91].

[1]The implementation of the Wasserstein distance is taken from [92].

Before we can use the Wasserstein distance however to train the neural network we need to set it up. It consists of three layers:

```
using Zygote, BrenierTwoFluid

model = Chain(GrassmannLayer(2,3), Dense(3, 8, tanh), Dense(8, 3, identity))

nn = NeuralNetwork(model, CPU(), Float64)
```

We then *lift* the neural network parameters via `GlobalSection`.

```
λY = GlobalSection(params(nn))
```

As the cost function c for the Wasserstein loss[2] we simply use:

```
# this computes the cost that is associated to the Wasserstein distance
c = (x,y) -> .5 * norm(x - y)^2
∇c = (x,y) -> x - y
```

We define a function `compute_wasserstein_gradient`; this is the gradient of the Wasserstein distance with respect to one of the probability distributions it is supplied with (this is further explained below). The function is defined as:

```
function compute_wasserstein_gradient(ensemble1::AT, ensemble2::AT) where AT<:AbstractArray
    number_of_particles1 = size(ensemble1, 2)
    number_of_particles2 = size(ensemble2, 2)
    V = SinkhornVariable(copy(ensemble1'), ones(number_of_particles1) / number_of_particles1)
    W = SinkhornVariable(copy(ensemble2'), ones(number_of_particles2) / number_of_particles2)
    S = SinkhornDivergence(V, W, c, params; islog = true)
    initialize_potentials!(S)
    compute!(S)
    value(S), x_gradient!(S, ∇c)'
end
```

This function associates particles in two point clouds with each other. As an illustrative example we will compare the following two point clouds:

$$\mathcal{D}_1 = \{(x,y,z) : (x,y) \in -1.5 : 0.1 : 1.5^2, z = f(x,y,z)\}$$

and

$$\mathcal{D}_2 = \left\{ \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} + x : x \sim \texttt{rand} \right\},$$

[2]For each Wasserstein loss we need to define such a cost function. For this particular choice of c the Wasserstein distance corresponds to a W_2 loss.

where $x \sim$ `rand` means that we draw x with the function `rand`. In code the two sets are:

```
xyz_points = hcat([[x, y, rosenbrock([x,y])] for x in x for y in y]...)
```

and

```
point_cloud = rand(size(xyz_points)...) .+ [2., 2., 2.]
```

We then compute the Wasserstein gradients and plot 30 of those (picked at random):

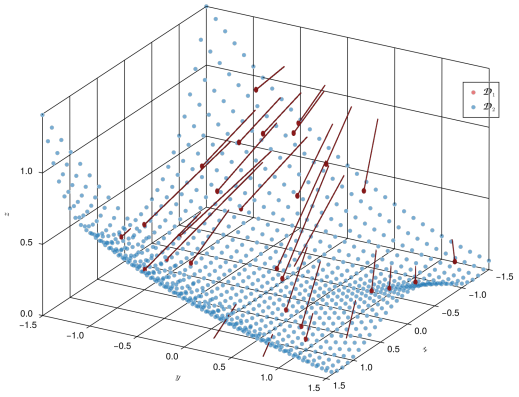


Figure 12.3: Gradient of Wasserstein distance for a particle discretization.

We now want to train a neural network based on this Wasserstein loss. The loss function is:

$$L_{\mathcal{NN}}(\theta) = W_2(\mathcal{NN}_\theta([x^{(1)}, \ldots, x^{(\texttt{np})}]), \mathcal{D}_2),$$

where `np` is the number of points in $\mathcal{D}_2$ and W_2 is the *Wasserstein distance*. We then have

$$\nabla_\theta L_{\mathcal{NN}} = (\nabla W_2)\nabla_\theta \mathcal{NN},$$

where ∇W_2 is equivalent to the function `compute_wasserstein_gradient`.

```
function compute_gradient(ps::NeuralNetworkParameters)
    samples = randn(2, size(xyz_points, 2))
    estimate, nn_pullback = Zygote.pullback(ps -> model(samples, ps), ps)

    valS, wasserstein_gradient = compute_wasserstein_gradient(estimate, xyz_points)
    valS, nn_pullback(wasserstein_gradient)[1]
end

# note the very high value for the learning rate
optimizer = Optimizer(nn, AdamOptimizer(1e-1))
```

So we use `Zygote` [93] to compute $\nabla_\theta \mathcal{NN}$ and we use `compute_wasserstein_gradient` to obtain ∇W_2. We can now train our network:

```
# note the small number of training steps
const training_steps = 80
loss_array = zeros(training_steps)
for i in 1:training_steps
    val, dp = compute_gradient(params(nn))
    loss_array[i] = val
    optimization_step!(optimizer, λY, params(nn), dp)
end
```

and plot the training loss:

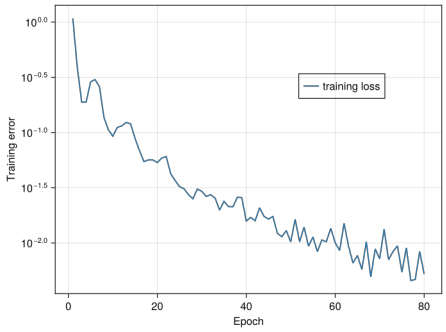


Figure 12.4: Training loss for the network with Grassmann layers.

Now we plot a few points to check how well they match the graph:

```
const number_of_points = 35
coordinates = nn(randn(2, number_of_points))
```

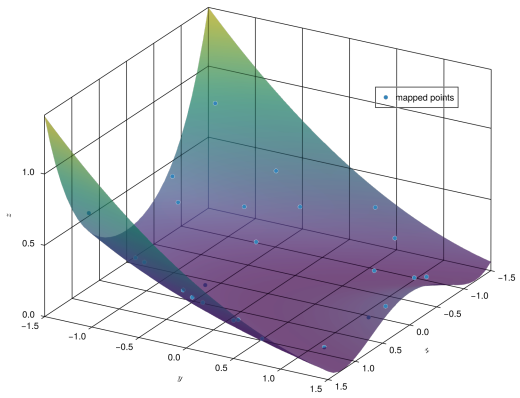


Figure 12.5: The blue points were obtained with the neural network sampler.

If points appear in darker color this means that they lie behind the graph of the Rosenbrock function.

Library Functions

`GeometricMachineLearning.GrassmannLayer` – Type.

```
GrassmannLayer(n, N)
```

Make an instance of `GrassmannLayer`.

This layer performs simple multiplication with an element of the Grassmann manifold, i.e.

$$\texttt{GrassmannLayer} : x \mapsto Ax,$$

where A is a representation of an element in the Grassmann manifold, i.e. $\mathcal{A} = \text{span}(A)$.

source

Chapter Summary

In this chapter we used a neural network, which has some of its weights lie on the *Grassmann manifold*, to sample from a distribution from which we have some data available. In order to accomplish this task we combined the gradient of a Wasserstein distance with the pullback of an automatic differentiation routine. At the end we saw that the we could sample new data which were close to the original, given ones.

Part V

Summary and Outlook

Conclusion

In this dissertation it was shown how neural networks can be imbued with structure to improve their approximation capabilities when applied to physical systems. In the following we summarize the novelties of this work and give an outlook for how it can be expanded in the future.

Reduced Order Modeling as Motivation

Most of the work presented in this dissertation is motivated by data-driven reduced order modeling (see Chapter 4.1). This is the discipline of building low-dimensional surrogate models from data that come from high-dimensional full order models. Both the low-dimensional surrogate model and the high-dimensional full order model are described by differential equations. When we talk about *structure-preserving reduced order modeling* we mean that the equation on the low-dimensional space shares features with the equation on the high-dimensional space (see Chapter 4.5). In this work these properties were mainly for the vector field to be symplectic (see Chapter 3.1) or divergence-free (see Chapter 3.2). A typical reduced order modeling framework is further divided into two phases:

1. in the *offline phase* we find the low-dimensional surrogate model (reduced representation) and
2. in the *online phase* we solve the equations in the reduced space.

For the offline phase we proposed symplectic autoencoders (see Chapter 8.2), and for the online phase we proposed volume-preserving transformers (see Chapter 8.8) and linear symplectic transformers (see Chapter 8.9). In the following we summarize the three main methods that were developed in the course of this dissertation and constitute its main results: symplectic autoencoders, structure-preserving transformers and structure-preserving optimizers.

Structure-Preserving Reduced Order Modeling of Hamiltonian Systems - The Offline Phase

A central part of this dissertation was the development of symplectic autoencoders (see Chapter 8.2) [33]. Symplectic autoencoders build on existing approaches of *symplectic neural networks* (SympNets) [10] and *proper symplectic decomposition* (PSD) [11], both of which *preserve symplecticity*. SympNets can approximate arbitrary canonical symplectic maps in $\mathbb{R}^{2n}$, i.e.

$$\text{SympNet} : \mathbb{R}^{2n} \to \mathbb{R}^{2n},$$

but the input has necessarily the same dimension as the output. PSD can change dimension, i.e.[1]

[1] Here we only show the *PSD encoder* PSD^{enc}. A complete reduced order modeling framework also need a decoder PSD^{dec} in addition to the encoder. When we use PSD both of these maps are linear, i.e. can be represented as $(\text{PSD}^{\text{enc}})^T, \text{PSD}^{\text{dec}} \in \mathbb{R}^{2N \times 2n}$.

$$\text{PSD}^{\text{enc}} : \mathbb{R}^{2N} \to \mathbb{R}^{2n},$$

but is strictly linear. Symplectic autoencoders offer a way of (i) constructing nonlinear symplectic maps that (ii) can change dimension. We used these to reduce a 400-dimensional Hamiltonian system to a two-dimensional one[2]:

$$(\mathbb{R}^{400}, H) \xrightarrow{\text{SAE}^{\text{enc}}} (\mathbb{R}^2, \bar{H}).$$

For this case we observed speed-ups of up to a factor 1000 when a symplectic autoencoder was combined with a transformer in the online phase (see Chapter 9.1). We also compared the symplectic autoencoder to a PSD, and showed that the PSD was unable to learn a useful two-dimensional representation.

Like PSD, symplectic autoencoders have the property that they induce a Hamiltonian system on the reduced space. This distinguishes them from "weakly symplectic autoencoders" [62, 63] that only approximately obtain a Hamiltonian system on a restricted domain by using a "physics-informed neural networks" [7] approach.

We also mention that the development of symplectic autoencoders required generalizing existing neural network optimizers to manifolds[3]. This is further discussed below.

Structure-Preserving Neural Network-Based Integrators - The Online Phase

For the online phase of reduced order modeling we developed new neural network architectures based on the transformer (see Chapter 8.7) [9] which is a neural network architecture that is extensively used in other fields of neural network research such as natural language processing[4]. We used transformers to build an equivalent of structure-preserving multi-step methods [3].

The transformer consists of a composition of standard neural network layers and attention layers:

$$\text{Transformer}(Z) = \mathcal{NN}_n \circ \text{AttentionLayer}_n \circ \cdots \circ \mathcal{NN}_1 \circ \text{AttentionLayer}_1(Z),$$

where $\mathcal{NN}$ indicates a standard neural network layer (e.g. a multilayer perceptron). The attention layer makes it possible for a transformer to process time series data by acting on a whole series of vectors at once:

$$\text{AttentionLayer}(Z) = \text{AttentionLayer}(z^{(1)}, \ldots, z^{(T)}) = [f^1(z^{(1)}, \ldots, z^{(T)}), \ldots, f^T(z^{(1)}, \ldots, z^{(T)})].$$

[2] $\bar{H} = H \circ \Psi_{\theta_2}^{\text{dec}} : \mathbb{R}^2 \to \mathbb{R}$ here refers to the *induced Hamiltonian on the reduced space*. *SAE* is short for *symplectic autoencoder*.

[3] We also refer to optimizers that preserve manifold structure as *structure-preserving optimizers*.

[4] The T in ChatGPT [2] stands for *transformer*.

The attention layer thus *performs a preprocessing step* after which the standard neural network layer $\mathcal{NN}$ is applied.

In this dissertation we presented two modifications of the standard transformer: the volume-preserving transformer (see Chapter 8.8) [30] and the linear symplectic transformer (see Chapter 8.9). In both cases we modified the attention mechanism so that it is either volume-preserving (in the first case) or symplectic (in the second case). The standard neural network layer $\mathcal{NN}$ was replaced by a volume-preserving feedforward neural network (see Chapter 8.6) or a symplectic neural network (see Chapter 8.5) [10] respectively.

In this dissertation we applied the volume-preserving transformer for learning the trajectory of a rigid body (see Chapter 11.2) and the linear symplectic transformer for learning the trajectory of a coupled harmonic oscillator (see Chapter 10.2). In both cases our new transformer architecture significantly outperformed the standard transformer. The trajectory modeled with the volume-preserving transformer for instance stays very close to a submanifold which is a level set of the quadratic invariant $I(z_1, z_2, z_3) = z_1^2 + z_2^2 + z_3^2$. This is not the case for the standard transformer: it moves away from this submanifold after a few time steps.

Structure-Preserving Optimizers

Training a symplectic autoencoder requires optimization on manifolds[5]. The particular manifolds we need in this case are "homogeneous spaces" [94]. In this dissertation we proposed a new optimizer framework that manages to generalize existing neural network optimizers to manifolds (see Chapter 5.1). This is done by identifying a global tangent space representation (see Chapter 2.11) and dispenses with the need for a *projection step* as is necessary in other approaches [69, 95].

As was already observed by others [69, 96, 97] putting weights on manifolds can improve training significantly in contexts other than scientific computing. Motivated by this we show an example of training a vision transformer [29] on the MNIST data set [32] to demonstrate the efficacy of the new optimizers. Contrary to other applications of the transformer we do not have to rely on layer normalization [98] or add connections to achieve convergent training for relatively big neural networks (see Chapter 11.1). We also applied the new optimizers to a neural network that contains weights on the Grassmann manifold (see Chapter 2.10) to be able to sample from a nonlinear space (see Chapter 12.1).

Outlook

We believe that the topics *structure-preserving autoencoders*, *structure-preserving transformers, structure-preserving optimizers* and *structure-preserving machine learning* in general offer great potential for future research.

Symplectic autoencoders could be used for model reduction of higher-dimensional systems [99] as well as using them for treating systems that are more general than canonical Hamiltonian ones; these include port-Hamiltonian [100] and metriplectic [101] systems. Structure-preserving model order reductions for such systems have been proposed [102–105] but without using neural networks. In the appendix we sketch how symplectic autoencoders could be used for structure-preserving model reduction of port-Hamiltonian systems (see Chapter D.1).

Structure-preserving transformers have shown great potential for learning dynamical systems, but their application should not be limited to that area. Structure-preserving machine learning techniques such as *Hamilton Monte Carlo* [106] has been used in various fields such as image classification

[5] This is necessary to preserve the symplectic structure of the neural network.

[107] and inverse problems [108] and we believe that the structure-preserving transformers introduced in this work can also find applications in these fields, by replacing the activation function in the attention layers of a vision transformer for example.

Lastly structure-preserving optimization is an exciting field, especially with regards to neural networks. The manifold optimizers introduced in this work can speed up neural network training significantly and are suitable for modern hardware (i.e. GPUs). They are however based on existing neural network optimizers such as Adam [23] and thus still lack a clear geometric interpretation. By utilizing a more geometric representation, as presented in this work, we hope to be able to find a differential equation describing Adam and other neural network optimizer, perhaps through a variational principle [109, 110]. One could also build on the existing optimization framework and use retractions other than the *geodesic retraction* and the *Cayley retraction* presented here (see Chapter 5.2); an example would be a *QR-based retraction* [111, 112]. This will be left for future work.

Part VI

References

References

[1] I. Goodfellow, Y. Bengio and A. Courville. *Deep learning* (MIT press, Cambridge, MA, 2016).

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat and others. *Gpt-4 technical report*, arXiv preprint arXiv:2303.08774 (2023).

[3] E. Hairer, C. Lubich and G. Wanner. *Geometric Numerical integration: structure-preserving algorithms for ordinary differential equations* (Springer, Heidelberg, 2006).

[4] Y. Duan, J. S. Edwards and Y. K. Dwivedi. *Artificial intelligence for decision making in the era of Big Data–evolution, challenges and research agenda.* International journal of information management **48**, 63–71 (2019).

[5] D. C. Psichogios and L. H. Ungar. *A hybrid neural network-first principles approach to process modeling.* AIChE Journal **38**, 1499–1511 (1992).

[6] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild and others. *Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence* (USDOE Office of Science (SC), Washington, DC (United States), 2019).

[7] M. Raissi, P. Perdikaris and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational physics **378**, 686–707 (2019).

[8] K. Hornik, M. Stinchcombe and H. White. *Multilayer feedforward networks are universal approximators.* Neural networks **2**, 359–366 (1989).

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. *Attention is all you need.* Advances in neural information processing systems **30** (2017).

[10] P. Jin, Z. Zhang, A. Zhu, Y. Tang and G. E. Karniadakis. *SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems.* Neural Networks **132**, 166–179 (2020).

[11] L. Peng and K. Mohseni. *Symplectic model reduction of Hamiltonian systems.* SIAM Journal on Scientific Computing **38**, A1–A27 (2016).

[12] D. N. Arnold, R. S. Falk and R. Winther. *Finite element exterior calculus, homological techniques, and applications.* Acta numerica **15**, 1–155 (2006).

[13] Y. Lishkova, P. Scherer, S. Ridderbusch, M. Jamnik, P. Liò, S. Ober-Blöbaum and C. Offen. *Discrete Lagrangian neural networks with automatic symmetry discovery.* IFAC-PapersOnLine **56**, 3203–3210 (2023).

[14] E. Dierkes, C. Offen, S. Ober-Blöbaum and K. Flaßkamp. *Hamiltonian neural networks with automatic symmetry detection.* Chaos: An Interdisciplinary Journal of Nonlinear Science **33** (2023).

[15] B. Brantner and M. Kraus. *GeometricMachineLearning.jl: Geometric Machine Learning in Julia*, https://github.com/JuliaGNI/GeometricMachineLearning.jl (2020).

[16] The Julia Company. *Documentation*, https://docs.julialang.org/en/v1/manual/documentation/ (2024). Accessed on August 19, 2024.

[17] Nvidia Corporation. *GeForce RTX 4090*, https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4090/ (2022). Accessed on August 13, 2024.

[18] GitHub, Inc. *About GitHub-hosted runners*, https://docs.github.com/en/actions/using-github-hosted-runners/using-github-hosted-runners/about-github-hosted-runners (2024). Accessed on August 13, 2024.

[19] S. Danisch and J. Krumbiegel. *Makie.jl: Flexible high-performance data visualization for Julia*. Journal of Open Source Software **6**, 3349 (2021).

[20] D. Bon, G. Pai, G. Bellaard, O. Mula and R. Duits. *Optimal Transport on the Lie Group of Roto-translations*, arXiv preprint arXiv:2402.15322 (2024).

[21] T. Bendokat, R. Zimmermann and P.-A. Absil. *A Grassmann manifold handbook: Basic geometry and computational aspects*, arXiv preprint arXiv:2011.13699 (2020).

[22] T. Bendokat and R. Zimmermann. *The real symplectic Stiefel and Grassmann manifolds: metrics, geodesics and applications*, arXiv preprint arXiv:2108.12447 (2021).

[23] D. Kingma. *Adam: a method for stochastic optimization*, arXiv preprint arXiv:1412.6980 (2014).

[24] J. N. Stephen J. Wright. *Numerical optimization* (Springer Science+Business Media, New York, NY, 2006).

[25] P.-A. Absil, R. Mahony and R. Sepulchre. *Optimization algorithms on matrix manifolds* (Princeton University Press, Princeton, New Jersey, 2008).

[26] B. Brantner. *Generalizing Adam To Manifolds For Efficiently Training Transformers*, arXiv preprint arXiv:2305.16901 (2023).

[27] J. Bajārs. *Locally-symplectic neural networks for learning volume-preserving dynamics*. Journal of Computational Physics **476**, 111911 (2023).

[28] N. Patwardhan, S. Marrone and C. Sansone. *Transformers in the real world: A survey on nlp applications*. Information **14**, 242 (2023).

[29] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly and others. *An image is worth 16x16 words: Transformers for image recognition at scale*, arXiv preprint arXiv:2010.11929 (2020).

[30] B. Brantner, M. Kraus, Z. Li and G. de Romemont. *Volume-preserving transformers for learning time series data with structure*. ESAIM: Proceedings and Surveys **81**, 123–144 (2025).

[31] M. Toda. *Vibration of a chain with nonlinear interaction*. Journal of the Physical Society of Japan **22**, 431–436 (1967).

[32] L. Deng. *The mnist database of handwritten digit images for machine learning research*. IEEE Signal Processing Magazine **29**, 141–142 (2012).

[33] B. Brantner and M. Kraus. *Symplectic autoencoders for Model Reduction of Hamiltonian Systems*, arXiv preprint arXiv:2312.10004 (2023).

[34] S. Lipschutz. *General Topology* (McGraw-Hill Book Company, New York City, New York, 1965).

[35] S. Lang. *Fundamentals of differential geometry*. Vol. 191 (Springer Science & Business Media, 2012).

[36] S. I. Richard L. Bishop. *Tensor Analysis on Manifolds* (Dover Publications, Mineola, New York, 1980).

[37] S. Lang. *Real and functional analysis*. Vol. 142 (Springer Science & Business Media, 2012).

[38] F. Mezzadri. *How to generate random matrices from the classical compact groups*, arXiv preprint math-ph/0609050 (2006).

[39] M. P. Do Carmo and J. Flaherty Francis. *Riemannian geometry*. Vol. 2 (Springer, 1992).

[40] P.-A. Absil, R. Mahony and R. Sepulchre. *Riemannian geometry of Grassmann manifolds with a view on algorithmic computation*. Acta Applicandae Mathematica **80**, 199–220 (2004).

[41] D. D. Holm, T. Schmah and C. Stoica. *Geometric mechanics and symmetry: from finite to infinite dimensions*. Vol. 12 (Oxford University Press, Oxford, UK, 2009).

[42] V. I. Arnold. *Mathematical methods of classical mechanics*. Vol. 60 of *Graduate Texts in Mathematics* (Springer Verlag, Berlin, 1978).

[43] M. Kraus, K. Kormann, P. J. Morrison and E. Sonnendrücker. *GEMPIC: geometric electromagnetic particle-in-cell methods*. Journal of Plasma Physics **83**, 905830401 (2017).

[44] M. Kraus. *GeometricIntegrators.jl: Geometric Numerical Integration in Julia*, https://github.com/JuliaGNI/GeometricIntegrators.jl (2020).

[45] Z. Ge and J. E. Marsden. *Lie-poisson hamilton-jacobi theory and lie-poisson integrators*. Physics Letters A **133**, 134–139 (1988).

[46] C. Yun, S. Bhojanapalli, A. S. Rawat, S. J. Reddi and S. Kumar. *Are transformers universal approximators of sequence-to-sequence functions?* arXiv preprint arXiv:1912.10077 (2019).

[47] D.-X. Zhou. *Universality of deep convolutional neural networks*. Applied and computational harmonic analysis **48**, 787–794 (2020).

[48] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou and M. Tegmark. *Kan: Kolmogorov-arnold networks*, arXiv preprint arXiv:2404.19756 (2024).

[49] S. Greydanus, M. Dzamba and J. Yosinski. *Hamiltonian neural networks*. Advances in neural information processing systems **32** (2019).

[50] J. W. Burby, Q. Tang and R. Maulik. *Fast neural Poincaré maps for toroidal magnetic fields*. Plasma Physics and Controlled Fusion **63**, 024001 (2020).

[51] P. Horn, V. S. Ulibarrena, B. Koren and S. P. Zwart. *A generalized framework of neural networks for Hamiltonian systems*. Journal of Computational Physics **521**, 113536 (2025).

[52] E. Celledoni, M. J. Ehrhardt, C. Etmann, R. I. McLachlan, B. Owren, C.-B. Schonlieb and F. Sherry. *Structure-preserving deep learning*. European journal of applied mathematics **32**, 888–936 (2021).

[53] T. Blickhan. *A registration method for reduced basis problems using linear optimal transport*, arXiv preprint arXiv:2304.14884 (2023).

[54] S. Fresca, L. Dede' and A. Manzoni. *A comprehensive deep learning-based approach to reduced order modeling of nonlinear time-dependent parametrized PDEs*. Journal of Scientific Computing **87**, 1–36 (2021).

[55] K. Lee and K. T. Carlberg. *Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders*. Journal of Computational Physics **404**, 108973 (2020).

[56] M. J. Gander and G. Wanner. *From Euler, Ritz, and Galerkin to modern computing*. Siam Review **54**, 627–666 (2012).

[57] F. Arbes, C. Greif and K. Urban. *The Kolmogorov N-width for linear transport: Exact representation and the influence of the data*, arXiv preprint arXiv:2305.00066 (2023).

[58] C. Greif and K. Urban. *Decay of the Kolmogorov N-width for wave problems*. Applied Mathematics Letters **96**, 216–222 (2019).

[59] A. Chatterjee. *An introduction to the proper orthogonal decomposition*. Current science, 808–817 (2000).

[60] S. Volkwein. *Proper orthogonal decomposition: Theory and reduced-order modelling*. Lecture Notes, University of Konstanz **4**, 1–29 (2013).

[61] T. M. Tyranowski and M. Kraus. *Symplectic model reduction methods for the Vlasov equation*. Contributions to Plasma Physics **63**, e202200046 (2023).

[62] P. Buchfink, S. Glas and B. Haasdonk. *Symplectic model reduction of Hamiltonian systems on nonlinear manifolds and approximation with weakly symplectic autoencoder*. SIAM Journal on Scientific Computing **45**, A289–A311 (2023).

[63] S. Yıldız, P. Goyal, T. Bendokat and P. Benner. *Data-Driven Identification of Quadratic Representations for Nonlinear Hamiltonian Systems Using Weakly Symplectic Liftings*. Journal of Machine Learning for Modeling and Computing **5** (2024).

[64] B. Gao, N. T. Son, P.-A. Absil and T. Stykel. *Riemannian optimization on the symplectic Stiefel manifold*. SIAM Journal on Optimization **31**, 1546–1575 (2021).

[65] B. O'neill. *Semi-Riemannian geometry with applications to relativity* (Academic press, New York City, New York, 1983).

[66] E. Celledoni and A. Iserles. *Approximating the exponential from a Lie algebra to a Lie group*. Mathematics of Computation **69**, 1457–1480 (2000).

[67] C. Fraikin, K. Hüper and P. V. Dooren. *Optimization over the Stiefel manifold*. In: *PAMM: Proceedings in Applied Mathematics and Mechanics*, Vol. 7 no. 1 (Wiley Online Library, 2007); pp. 1062205–1062206.

[68] M. Schlarb. *Covariant Derivatives on Homogeneous Spaces: Horizontal Lifts and Parallel Transport*. The Journal of Geometric Analysis **34**, 1–43 (2024).

[69] L. Kong, Y. Wang and M. Tao. *Momentum stiefel optimizer, with applications to suitably-orthogonal attention, and optimal transport*, arXiv preprint arXiv:2205.14173v3 (2023).

[70] A.Γ. (math.stackexchange user 253273). *Quasi-newton methods: Understanding DFP updating formula*, https://math.stackexchange.com/q/2279304 (2017). Accessed on September 19, 2024.

[71] P. Kenneweg, T. Kenneweg and B. Hammer. *Improving Line Search Methods for Large Scale Neural Network Training*. In: *2024 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA)* (IEEE, 2024); pp. 1–6.

[72] S. Vaswani, A. Mishkin, I. Laradji, M. Schmidt, G. Gidel and S. Lacoste-Julien. *Painless stochastic gradient: Interpolation, line-search, and convergence rates*. Advances in neural information processing systems **32** (2019).

[73] W. Huang, P.-A. Absil and K. A. Gallivan. *A Riemannian BFGS method for nonconvex optimization problems*. In: *Numerical Mathematics and Advanced Applications ENUMATH 2015* (Springer, 2016); pp. 627–634.

[74] B. Gao, N. T. Son and T. Stykel. *Symplectic Stiefel manifold: tractable metrics, second-order geometry and Newton's methods*, arXiv preprint arXiv:2406.14299 (2024).

[75] F. Kang and S. Zai-Jiu. *Volume-preserving algorithms for source-free dynamical systems*. Numerische Mathematik **71**, 451–463 (1995).

[76] H. Cardot. *Recurrent neural networks for temporal data processing* (BoD–Books on Demand, 2011).

[77] D. Bahdanau, K. Cho and Y. Bengio. *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473 (2014).

[78] K. Jacobs. *Discrete Stochastics* (Birkhäuser Verlag, Basel, Switzerland, 1992).

[79] B. Brantner. *Proof of Volume Preservation of $Z \mapsto Z \cdot \mathrm{Cayley}(Z^T A Z)$ for Skew-Symmetric A*. ResearchGate preprint RG.2.2.15928.81921 (2025).

[80] M.-T. Luong, H. Pham and C. D. Manning. *Effective approaches to attention-based neural machine translation*, arXiv preprint arXiv:1508.04025 (2015).

[81] K. Feng and M.-z. Qin. *The symplectic methods for the computation of Hamiltonian equations*. In: *Numerical Methods for Partial Differential Equations: Proceedings of a Conference held in Shanghai, PR China, March 25–29, 1987* (Springer, 1987); pp. 1–37.

[82] Z. Ge and K. Feng. *On the approximation of linear Hamiltonian systems*. Journal of Computational Mathematics, 88–97 (1988).

[83] *Attention Is Off By One*, https://www.evanmiller.org/attention-is-off-by-one.html. Accessed: 2025-10-13.

[84] B. Leimkuhler and S. Reich. *Simulating hamiltonian dynamics*. No. 14 (Cambridge university press, 2004).

[85] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural computation **9**, 1735–1780 (1997).

[86] A. Hemmasian and A. Barati Farimani. *Reduced-order modeling of fluid flows with transformers*. Physics of Fluids **35** (2023).

[87] A. Solera-Rico, C. S. Vila, M. Gómez, Y. Wang, A. Almashjary, S. Dawson and R. Vinuesa, *β-Variational autoencoders and transformers for reduced-order modelling of fluid flows*, arXiv preprint arXiv:2304.03571 (2023).

[88] P. Jin, Z. Lin and B. Xiao. *Optimal unit triangular factorization of symplectic matrices*. Linear Algebra and its Applications (2022).

[89] T. Besard, C. Foket and B. De Sutter. *Effective Extensible Programming: Unleashing Julia on GPUs*. IEEE Transactions on Parallel and Distributed Systems (2018), arXiv:1712.03112 [cs.PL].

[90] C. Villani and others. *Optimal transport: old and new*. Vol. 338 (Springer, 2009).

[91] C. Villani. *Topics in optimal transportation*. Vol. 58 (American Mathematical Soc., 2021).

[92] T. Blickhan. *BrenierTwoFluids.jl*, https://github.com/ToBlick/BrenierTwoFluids (2023).

[93] M. Innes. *Don't Unroll Adjoint: Differentiating SSA-Form Programs*. CoRR **abs/1810.07951** (2018), arXiv:1810.07951.

[94] T. Frankel. *The geometry of physics: an introduction* (Cambridge university press, Cambridge, UK, 2011).

[95] J. Li, L. Fuxin and S. Todorovic. *Efficient riemannian optimization on the stiefel manifold via the cayley transform*, arXiv preprint arXiv:2002.01113 (2020).

[96] A. Zhang, A. Chan, Y. Tay, J. Fu, S. Wang, S. Zhang, H. Shao, S. Yao and R. K.-W. Lee. *On orthogonality constraints for transformers*. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, Vol. 2 (Association for Computational Linguistics, 2021); pp. 375–382.

[97] L. Huang, X. Liu, B. Lang, A. Yu, Y. Wang and B. Li. *Orthogonal weight normalization: Solution to optimization over multiple dependent stiefel manifolds in deep neural networks*. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32 no. 1 (2018).

[98] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang and T. Liu. *On layer normalization in the transformer architecture*. In: *International Conference on Machine Learning* (PMLR, 2020); pp. 10524–10533.

[99] S. Fresca and A. Manzoni. *POD-DL-ROM: Enhancing deep learning-based reduced order models for nonlinear parametrized PDEs by proper orthogonal decomposition*. Computer Methods in Applied Mechanics and Engineering **388**, 114181 (2022).

[100] A. Van Der Schaft, D. Jeltsema and others. *Port-Hamiltonian systems theory: An introductory overview*. Foundations and Trends in Systems and Control **1**, 173–378 (2014).

[101] P. J. Morrison. *A paradigm for joined Hamiltonian and dissipative systems*. Physica D: Nonlinear Phenomena **18**, 410–419 (1986).

[102] A. Gruber, M. Gunzburger, L. Ju and Z. Wang. *Energetically consistent model reduction for metriplectic systems*. Computer Methods in Applied Mechanics and Engineering **404**, 115709 (2023).

[103] T. F. Moser. *Structure-Preserving Model Reduction of Port-Hamiltonian Descriptor Systems*. Ph.D. Thesis, Technische Universität München (2023).

[104] P. Schulze. *Structure-preserving model reduction for port-hamiltonian systems based on a special class of nonlinear approximation ansatzes*, arXiv preprint arXiv:2302.06479 (2023).

[105] M. Mamunuzzaman and H. Zwart. *Structure preserving model order reduction of port-Hamiltonian systems*, arXiv preprint arXiv:2203.07751 (2022).

[106] S. Duane, A. D. Kennedy, B. J. Pendleton and D. Roweth. *Hybrid monte carlo*. Physics letters B **195**, 216–222 (1987).

[107] A. D. Cobb and B. Jalaian. *Scaling Hamiltonian Monte Carlo inference for Bayesian neural networks with symmetric splitting*. In: *Uncertainty in Artificial Intelligence* (PMLR, 2021); pp. 675–685.

[108] A. Fichtner and S. Simutė. *Hamiltonian Monte Carlo inversion of seismic sources in complex media*. Journal of Geophysical Research: Solid Earth **123**, 2984–2999 (2018).

[109] A. Wibisono, A. C. Wilson and M. I. Jordan. *A variational perspective on accelerated methods in optimization*, proceedings of the National Academy of Sciences **113**, E7351–E7358 (2016).

[110] V. Duruisseaux and M. Leok. *Accelerated optimization on Riemannian manifolds via discrete constrained variational integrators*. Journal of Nonlinear Science **32**, 42 (2022).

[111] H. Sato and K. Aihara. *Cholesky QR-based retraction on the generalized Stiefel manifold.* Computational Optimization and Applications **72**, 293–308 (2019).

[112] B. Gao, N. T. Son and T. Stykel. *Optimization on the symplectic Stiefel manifold: SR decomposition-based retraction and applications.* Linear Algebra and its Applications **682**, 50–85 (2024).

[113] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. Narayanan, M. Schanen and J. Doerfert. *Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme*. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, *SC '21* (Association for Computing Machinery, New York, NY, USA, 2021).

[114] M. Betancourt. *A geometric theory of higher-order automatic differentiation*, arXiv preprint arXiv:1812.11592 (2018).

[115] J. Bolte and E. Pauwels. *A mathematical model for automatic differentiation in machine learning.* Advances in Neural Information Processing Systems **33**, 10809–10819 (2020).

[116] R. Morandin. *Modeling and numerical treatment of port-Hamiltonian descriptor systems.* Ph.D. Thesis, Technische Universität Berlin (2023).

[117] H. Yoshimura and J. E. Marsden. *Dirac structures in Lagrangian mechanics Part I: implicit Lagrangian systems.* Journal of Geometry and Physics **57**, 133–156 (2006).

[118] H. Yoshimura and J. E. Marsden. *Dirac structures in Lagrangian mechanics Part II: Variational structures.* Journal of Geometry and Physics **57**, 209–250 (2006).

[119] P. Kotyczka and L. Lefevre. *Discrete-time port-Hamiltonian systems: A definition based on symplectic integration.* Systems & Control Letters **133**, 104530 (2019).

[120] V. Mehrmann and R. Morandin. *Structure-preserving discretization for port-Hamiltonian descriptor systems.* In: *2019 IEEE 58th Conference on Decision and Control (CDC)* (IEEE, 2019); pp. 6863–6868.

[121] J. Rettberg, J. Kneifl, J. Herb, P. Buchfink, J. Fehr and B. Haasdonk. *Data-driven identification of latent port-Hamiltonian systems*, arXiv preprint arXiv:2408.08185 (2024).

[122] S. E. Otto, G. R. Macchio and C. W. Rowley. *Learning nonlinear projections for reduced-order modeling of dynamical systems using constrained autoencoders.* Chaos: An Interdisciplinary Journal of Nonlinear Science **33** (2023).

[123] V. Churavy. *KernelAbstractions.jl*, https://github.com/JuliaGPU/KernelAbstractions.jl (2024). Used on August 14, 2024.

[124] T. Besard and M. Hawkins. *Metal.jl*, https://github.com/JuliaGPU/Metal.jl (2022). Used on August 14, 2024.

[125] T. Besard, *oneAPI.jl*, https://github.com/JuliaGPU/oneAPI.jl (2022). Used on August 14, 2024.

[126] K. Feng. *The step-transition operators for multi-step methods of ODE's.* Journal of Computational Mathematics, 193–202 (1998).

[127] B. Brantner, G. de Romemont, M. Kraus and Z. Li. *Structure-Preserving Transformers for Learning Parametrized Hamiltonian Systems*, arXiv preprint arXiv:2312:11166 (2023).

[128] T. Lin and H. Zha. *Riemannian manifold learning.* IEEE transactions on pattern analysis and machine intelligence **30**, 796–809 (2008).

Part VII

Index of Docstrings

Index of Docstrings

Manifolds

Geometric Structure

Reduced Order Modeling

General Framework for Manifold Optimization

- GeometricMachineLearning.Optimizer: page 89
- GeometricMachineLearning.optimization_step!: page 91
- GeometricMachineLearning.optimize_for_one_epoch!: page 90
- GeometricMachineLearning.cayley: page 100
- GeometricMachineLearning.cayley: page 100
- GeometricMachineLearning.cayley: page 99
- GeometricMachineLearning.geodesic: page 99
- GeometricMachineLearning.geodesic: page 99
- GeometricMachineLearning.𝔄: page 101
- GeometricMachineLearning.𝔄: page 101

Optimizer Methods

- GeometricMachineLearning.AbstractCache: page 114
- GeometricMachineLearning.AdamCache: page 115
- GeometricMachineLearning.AdamOptimizer: page 113
- GeometricMachineLearning.AdamOptimizerWithDecay: page 114
- GeometricMachineLearning.GradientCache: page 114
- GeometricMachineLearning.GradientOptimizer: page 112
- GeometricMachineLearning.MomentumCache: page 114
- GeometricMachineLearning.MomentumOptimizer: page 112
- GeometricMachineLearning.OptimizerMethod: page 112
- AbstractNeuralNetworks.update!: page 115
- GeometricMachineLearning.BFGSCache: page 121
- GeometricMachineLearning.BFGSOptimizer: page 121
- AbstractNeuralNetworks.update!: page 122
- Base.vec: page 122

Layers

Architectures

- Base.iterate: page 162
- Base.iterate: page 163
- GeometricMachineLearning.GeneralizedHamiltonianArchitecture: page 167
- GeometricMachineLearning.HNNLoss: page 166
- GeometricMachineLearning.HamiltonianArchitecture: page 165
- GeometricMachineLearning.StandardHamiltonianArchitecture: page 165
- SymbolicNeuralNetworks.SymbolicPullback: page 166
- GeometricMachineLearning._processing: page 167
- GeometricMachineLearning.hamiltonian_vector_field: page 165
- GeometricMachineLearning.symbolic_hamiltonian_vector_field: page 166
- GeometricMachineLearning.GSympNet: page 173
- GeometricMachineLearning.LASympNet: page 172
- GeometricMachineLearning.SympNet: page 172
- GeometricMachineLearning.VolumePreservingFeedForward: page 175
- GeometricMachineLearning.ClassificationTransformer: page 179
- GeometricMachineLearning.StandardTransformerIntegrator: page 178
- GeometricMachineLearning.Transformer: page 179
- GeometricMachineLearning.assign_output_estimate: page 180
- GeometricMachineLearning.VolumePreservingTransformer: page 182
- GeometricMachineLearning.LinearSymplecticTransformer: page 183
- GeometricMachineLearning.SymplecticTransformer: page 185

Transformers with Structure

- GeometricMachineLearning.ClassificationLayer: page 213
- GeometricMachineLearning.accuracy: page 212
- GeometricMachineLearning.onehotbatch: page 212
- GeometricMachineLearning.split_and_flatten: page 211
- GeometricMachineLearning.DummyNNIntegrator: page 224
- GeometricMachineLearning.DummyTransformer: page 224

Learning Nonlinear Spaces

- GeometricMachineLearning.GrassmannLayer: page 232

Data Loader

Special Arrays, Tensors and Pullbacks

Part VIII

Appendix

Appendix A

Data Loader

Here we discuss the *data loader*, which forms an important part of `GeometricMachineLearning`. It is used to store data and sample from them.

A.1 Snapshot Matrix

The snapshot matrix stores solutions of the high-dimensional ODE (obtained from discretizing a PDE). In the offline phase of reduced order modeling (see Chapter 4.1) this is then used to construct reduced bases in a data-driven way. So (for a single parameter[1]) the snapshot matrix takes the following form:

$$M = \left[\begin{array}{c|c|c|c} \hat{u}_1(t_0) & \hat{u}_1(t_1) & \dots & \hat{u}_1(t_f) \\ \hat{u}_2(t_0) & \hat{u}_2(t_1) & \dots & \hat{u}_2(t_f) \\ \hat{u}_3(t_0) & \hat{u}_3(t_1) & \dots & \hat{u}_3(t_f) \\ \dots & \dots & \dots & \dots \\ \hat{u}_{2N}(t_0) & \hat{u}_{2N}(t_1) & \dots & \hat{u}_{2N}(t_f) \end{array}\right].$$

In the example above we store a matrix whose first axis is the system dimension (i.e. a column is an element of $\mathcal{M} \subset \mathbb{R}^{2N}$) and the second dimension gives the time step.

The starting point for using the snapshot matrix as data for a machine learning model is that all the columns of M live on a lower-dimensional solution manifold (see Chapter 4.1) $\mathcal{M}$ and we can use techniques such as proper orthogonal decomposition (see Chapter 4.2) and autoencoders (see Chapter 4.3) to find or approximate this solution manifold. We also note that the second axis of M does not necessarily indicate time but can also represent various parameters (including initial conditions).

A.2 Snapshot Tensor

The snapshot tensor fulfills the same role as the snapshot matrix but has a third axis that describes different initial parameters (such as different initial conditions).

[1] If we deal with a parametrized PDE then the data can be interpreted as a snapshot tensor (see Chapter A.2). For training an autoencoder (see Chapter 4.3) in the offline phase however, we interpret these data as a matrix because they come from the same solution manifold (see Chapter 4.1); each column of the snapshot matrix M represents a point on the solution manifold.

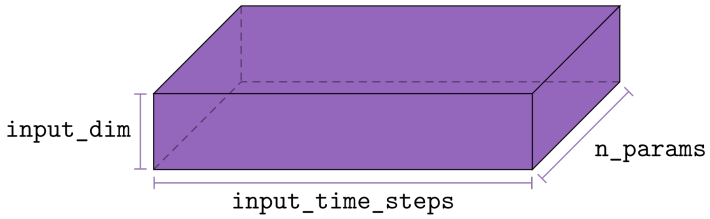


Figure A.1: The three axes of the snapshot tensor are (i) dimension, (ii) time and (iii) parameters.

When drawing samples from the snapshot tensor to train a neural networks we also need to specify a *sequence length* (as an argument to the `Batch` struct). When sampling a batch from the snapshot tensor we sample over the second axis (the *time dimension*) and the third axis of the tensor (the *parameter dimension*). The total number of batches[2] is

$$\lceil (\texttt{dl.input\_time\_steps} - \texttt{batch.seq\_length}) * \texttt{dl.n\_params} / \texttt{batch.batch\_size} \rceil,$$

where $\lceil \cdot \rceil$ is the *ceiling operation*.

A.3 The Data Loader

The `DataLoader` in `GeometricMachineLearning` is designed to make training convenient.

The data loader can be called with various types of arrays as input, for example a snapshot matrix (see Chapter A.1):

```
SnapshotMatrix = [ 1; 2;; 3; 4;; 5; 6;; 7; 8;; 9; 10 ]

dl = DataLoader(SnapshotMatrix; suppress_info = true)
```

```
DataLoader{Int64, Array{Int64, 3}, Nothing, :RegularData}([1; 2;;; 3; 4;;; 5; 6;;; 7; 8;;; 9; 10],
↪  nothing, 2, 1, 5, nothing, nothing)
```

or a snapshot tensor:

```
SnapshotTensor = [ 1;  2;; 3; 4;; 5; 6;; 7; 8;; 9; 10 ;;;]

dl = DataLoader(SnapshotTensor; suppress_info = true)
```

```
DataLoader{Int64, Array{Int64, 3}, Nothing, :TimeSeries}([1 3 ⋯ 7 9; 2 4 ⋯ 8 10;;;], nothing, 2,
↪  5, 1, nothing, nothing)
```

[2] The number of batches shown here is for the case `prediction_window` $= 0$. If `prediction_window` $\neq 0$, this number may be smaller. The corresponding function is `GeometricMachineLearning.number_of_batches`.

Here the `DataLoader` has different properties `:RegularData` and `:TimeSeries`. This indicates that in the first case we treat all columns in the input tensor independently; this is mostly used for autoencoder problems (see Chapter 4.3). In the second case we have *time series-like data*, which are mostly used for integration problems (see Chapter 8.3). As shown above the default when using a matrix is `:RegularData` and the default when using a tensor is `:TimeSeries`.

We can also treat a problem with a matrix as input as a time series-like problem by providing an additional keyword argument: `autoencoder=false`:

```
dl = DataLoader(SnapshotMatrix; autoencoder=false, suppress_info = true)
dl |> typeof
```

```
DataLoader{Int64, Array{Int64, 3}, Nothing, :TimeSeries}
```

If we deal with hamiltonian systems we typically split the coordinates into a q and a p part. Such data can also be used as input arguments for `DataLoader`:

```
SymplecticSnapshotTensor = (q = SnapshotTensor, p = SnapshotTensor)
dl = DataLoader(SymplecticSnapshotTensor)
dl |> typeof
```

```
DataLoader{Int64, @NamedTuple{q::Array{Int64, 3}, p::Array{Int64, 3}}, Nothing, :TimeSeries}
```

The dimension of the system is then the sum of the dimensions of the q and the p component:

```
dl.input_dim
```

```
4
```

Drawing Batches with `GeometricMachineLearning`

If we want to draw mini batches from a data set, we need to allocate an instance of `Batch`. If we call the corresponding functor on an instance of `DataLoader` we get the following result[1]:

```
matrix_data = [ 1 2 3 4  5;
                6 7 8 9 10]
dl = DataLoader(matrix_data; autoencoder = true)

batch = Batch(3)
batches = batch(dl)
```

[1] We first demonstrate how to sample data on the example of a matrix. The case of sampling from a tensor is slightly more complicated and is explained below.

```
([(1, 5), (1, 4), (1, 2)], [(1, 3), (1, 1)])
```

The output of applying the batch functor is always of the form:

$$([(b^t_{1,1}, b^p_{1,1}), (b^t_{1,2}, b^p_{1,2}), \ldots], [(b^t_{2,1}, b^p_{2,1}), (b^t_{2,2}, b^p_{2,2}), \ldots], [(b^t_{3,1}, b^p_{3,2}), \ldots], \ldots),$$

so it is a tuple of vectors of tuples. One vector represents one batch:

```
for (minibatch, i) in zip(batches[1], axes(batches[1], 1))
    println(stdout, minibatch[1], " = bᵗ₁" * join('₀' + d for d in digits(i)))
    println(stdout, minibatch[2], " = bᵖ₁" * join('₀' + d for d in digits(i)))
    println()
end
```

```
1 = bᵗ₁₁
5 = bᵖ₁₁

1 = bᵗ₁₂
4 = bᵖ₁₂

1 = bᵗ₁₃
2 = bᵖ₁₃
```

The tuples that make up this vector always have two entries: a *time index* b^t_{1i} and a *parameter index* b^p_{1i} indicated by the superscripts t and p respectively. Because `dl` in this example is of `autoencoder` type, the time index is always one. The parameter index differs. Because the input to `DataLoader` was a 2×5 matrix and we specified a batch size of three, there are two batches in total. The second batch is:

```
for (minibatch, i) in zip(batches[2], axes(batches[2], 1))
    println(stdout, minibatch[1], " = bᵗ₁" * join('₀' + d for d in digits(i)))
    println(stdout, minibatch[2], " = bᵖ₁" * join('₀' + d for d in digits(i)))
    println()
end
```

```
1 = bᵗ₁₁
3 = bᵖ₁₁

1 = bᵗ₁₂
1 = bᵖ₁₂
```

Looking at the first and the second batch together, we see that we sample with replacement, i.e. all indices $b^p_{1i} = 1, 2, 3, 4, 5$ appear. This also works if the data are in (q, p) form:

```
qp_data = (q = rand(Float32, 2, 5), p = rand(Float32, 2, 5))
dl = DataLoader(qp_data; autoencoder = true)

batch = Batch(3)
batch(dl)
```

```
([(1, 3), (1, 4), (1, 1)], [(1, 2), (1, 5)])
```

In those two examples the `autoencoder` keyword was set to `true` (the default). This is why the first index was always `1`. This changes if we set `autoencoder = false`:

```
qp_data = (q = rand(Float32, 2, 5), p = rand(Float32, 2, 5))
dl = DataLoader(qp_data; autoencoder = false) # false is default

batch = Batch(3)
batch(dl)
```

```
([(4, 1), (3, 1), (2, 1)], [(1, 1)])
```

Specifically the sampling routines do the following:

1. $\texttt{n\_indices} \leftarrow \texttt{n\_params} \vee \texttt{input\_time\_steps}$,
2. $\texttt{indices} \leftarrow \texttt{shuffle}(\texttt{1} : \texttt{n\_indices})$,
3. $\mathcal{I}_i \leftarrow \texttt{indices}[(\texttt{i} - \texttt{1}) \cdot \texttt{batch\_size} + \texttt{1}{:}i \cdot \texttt{batch\_size}]$ for $i = 1, \ldots, (\text{last} - 1)$,
4. $\mathcal{I}_{\texttt{last}} \leftarrow \texttt{indices}[(\texttt{n\_batches} - \texttt{1}) \cdot \texttt{batch\_size} + \texttt{1}{:}\ \texttt{end}]$.

Note that the routines are implemented in such a way that no two indices appear double, i.e. we *sample without replacement*.

Sampling from a tensor

We can also sample from a tensor:

```
qp_data = (q = rand(Float32, 2, 5, 3), p = rand(Float32, 2, 5, 3))
dl = DataLoader(qp_data)

# also specify sequence length and a prediction window here
batch = Batch(4, 2, 0)
batch(dl)
```

```
([(1, 3), (3, 3), (2, 3), (4, 3)], [(1, 1), (3, 1), (2, 1), (4, 1)], [(1, 2), (3, 2), (2, 2), (4,
↪  2)])
```

Sampling from a tensor is done the following way ($\mathcal{I}_i$ again denotes the batch indices for the i-th batch):

1. `time_indices` $\leftarrow$ `shuffle`$(1 : (\texttt{input\_time\_steps} - \texttt{seq\_length} - \texttt{prediction\_window})$,
2. `parameter_indices` $\leftarrow$ `shuffle`$(1 : \texttt{n\_params})$,
3. `complete_indices` $\leftarrow$ `product`$(\texttt{time\_indices}, \texttt{parameter\_indices})$,
4. $\mathcal{I}_i \leftarrow$ `complete_indices`$[(i-1) \cdot \texttt{batch\_size} + 1 : i \cdot \texttt{batch\_size}]$ for $i = 1, \ldots, (\text{last} - 1)$,
5. $\mathcal{I}_{\text{last}} \leftarrow$ `complete_indices`$[(\text{last} - 1) \cdot \texttt{batch\_size} + 1 : \texttt{end}]$.

Note that we supplied two additional arguments to the `Batch` constructor here: `seq_length` and `prediction_window`. These two arguments specify how many time are considered in one mini batch and how long the prediction runs into the future respectively. These two numbers are explained when we talk about structure on product spaces (see Chapter 7.3).

This algorithm can be visualized the following way (here `batch_size = 4`):

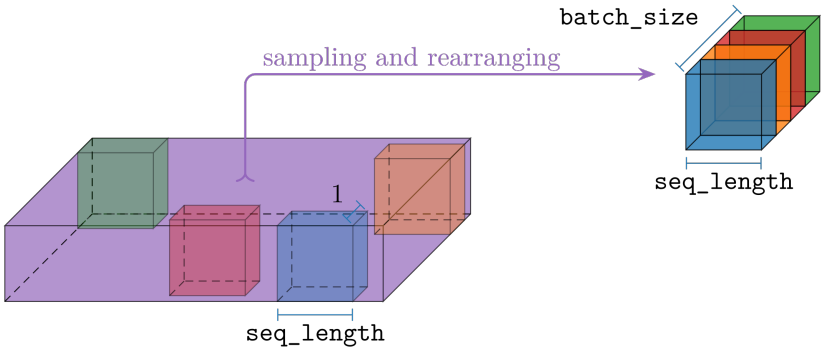


Figure A.2: Visualization of sampling from a tensor. Here the batch size was specified as four, i.e. we sample four blocks.

Here the sampling is performed over the second axis (the *time step dimension*) and the third axis (the *parameter dimension*). Whereas each block has thickness 1 in the x direction (i.e. pertains to a single parameter), its length in the y direction is `seq_length`. In total we sample as many such blocks as the batch size is big. By construction those blocks are never the same throughout a training epoch but may intersect each other!

Library Functions

`GeometricMachineLearning.DataLoader` – Type.

```
DataLoader(data)
```

Make an instance based on a data set.

This is designed to make training convenient.

Fields of `DataLoader`

The fields of the `DataLoader` struct are the following:

- `input`: The input data with axes (i) system dimension, (ii) number of time steps and (iii) number of parameters.
- `output`: The tensor that contains the output (supervised learning) - this may be of type `Nothing` if the constructor is only called with one tensor (unsupervised learning).
- `input_dim`: The *dimension* of the system, i.e. what is taken as input by a regular neural network.
- `input_time_steps`: The length of the entire time series (length of the second axis).
- `n_params`: The number of parameters that are present in the data set (length of third axis)
- `output_dim`: The dimension of the output tensor (first axis). If `output` is of type `Nothing`, then this is also of type `Nothing`.
- `output_time_steps`: The size of the second axis of the output tensor. If `output` is of type `Nothing`, then this is also of type `Nothing`.

Implementation

Even though `DataLoader` can be called with inputs of various forms, internally it always stores tensors with three axes.

```
using GeometricMachineLearning

data = [1 2 3; 4 5 6]
dl = DataLoader(data)
dl.input

# output

[ Info: You have provided a matrix as input. The axes will be interpreted as (i) system
↪  dimension and (ii) number of parameters.
2×1×3 Array{Int64, 3}:
[:, :, 1] =
 1
 4

[:, :, 2] =
 2
 5

[:, :, 3] =
 3
 6
```

source

`GeometricMachineLearning.Batch` – Type.

```
Batch
```

`Batch` is a struct whose functor acts on an instance of `DataLoader` to produce a sequence of training samples for training for one epoch.

See `Batch(::Int)` and `Batch(::Int, ::Int, ::Int)` for the different constructors.

The functor

An instance of `Batch` can be called on an instance of `DataLoader` to produce a sequence of samples that contain all the input data, i.e. for training for one epoch.

The output of applying `batch:Batch` to `dl::DataLoader` is a tuple of vectors of integers. Each of these vectors contains two integers: the first is the *time index* and the second one is the *parameter index*.

Examples

Consider the following example for drawing batches of size 2 for an instance of `DataLoader` constructed with a vector:

```
using GeometricMachineLearning
import Random

rng = Random.TaskLocalRNG()
Random.seed!(rng, 123)

dl = DataLoader(rand(rng, 5))
batch = Batch(2)

batch(dl)

# output

[ Info: You have provided a matrix as input. The axes will be interpreted as (i) system
↪  dimension and (ii) number of parameters.
([(1, 5), (1, 3)], [(1, 4), (1, 1)], [(1, 2)])
```

Here the first index is always 1 (the time dimension). We get a total number of 3 batches. The last batch is only of size 1 because we *sample without replacement*. Also see the docstring for `DataLoader(::AbstractVector)`.

source

`GeometricMachineLearning.Batch` – Method.

```
Batch(batch_size)
```

Make an instance of `Batch` for a specific batch size.

This is, among others, used to train neural networks of `NeuralNetworkIntegrator` type (as opposed to `TransformerIntegrator`).

source

`GeometricMachineLearning.Batch` – Method.

```
Batch(batch_size, seq_length)
```

Make an instance of `Batch` for a specific batch size and a sequence length.

This is used to train neural networks of `TransformerIntegrator` type.

Optionally the prediction window can also be specified by calling:

```
using GeometricMachineLearning

batch_size = 2
seq_length = 3
prediction_window = 2

Batch(batch_size, seq_length, prediction_window)

# output

Batch{:Transformer}(2, 3, 2)
```

Note that here the batch is of type `:Transformer`.

source

`GeometricMachineLearning.number_of_batches` – Function.

```
number_of_batches(dl, batch)
```

Compute the number of batches.

Here the distinction is between data that are *time-series like* and data that are *autoencoder like*.

Examples

```
using GeometricMachineLearning
using GeometricMachineLearning: number_of_batches
import Random

Random.seed!(123)

dat = [1, 2, 3, 4, 5]
dl₁ = DataLoader(dat; autoencoder = false, suppress_info = true) # time series-like
dl₂ = DataLoader(dat; autoencoder = true, suppress_info = true) # autoencoder-like
batch = Batch(3)

nob₁ = number_of_batches(dl₁, batch)
nob₂ = number_of_batches(dl₂, batch)
println(stdout, "Number of batches of dl₁: ", nob₁)
println(stdout, "Number of batches of dl₂: ", nob₂)
println(stdout, batch(dl₁), "\n", batch(dl₂))
```

```
# output

Number of batches of dl₁: 2
Number of batches of dl₂: 2
([(1, 1), (4, 1), (2, 1)], [(3, 1)])
([(1, 3), (1, 2), (1, 4)], [(1, 1), (1, 5)])
```

Here we see that in the *autoencoder case* that last minibatch has an additional element.

source

`GeometricMachineLearning.convert_input_and_batch_indices_to_array` – Function.

```
convert_input_and_batch_indices_to_array(dl, batch, batch_indices)
```

Assign batch data based on (i) input and (ii) batch indices.

Examples

```
using GeometricMachineLearning

dl = DataLoader([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]; suppress_info = true)
batch = Batch(3)
batch_indices = [(1, 1), (1, 3), (1, 5)]

GeometricMachineLearning.convert_input_and_batch_indices_to_array(dl, batch, batch_indices)

# output

1×1×3 Array{Float64, 3}:
[:, :, 1] =
 0.1

[:, :, 2] =
 0.3

[:, :, 3] =
 0.5
```

source

Appendix B

Special Arrays, Tensors and Pullbacks

`GeometricMachineLearning` has custom versions of matrices such as the symmetric and the skew-symmetric matrix implemented. These are important ingredients in e.g. SympNets and volume-preserving transformers and it is therefore important that those implementations also run efficiently on GPU. We also show how to build custom pullbacks for specific functions in `Julia`.

B.1 Symmetric, Skew-Symmetric and Triangular Matrices.

Among the special arrays implemented in `GeometricMachineLearning` `SymmetricMatrix`, `SkewSymMatrix`, `UpperTriangular` and `LowerTriangular` are the most common ones and similar implementations can also be found in other libraries; `LinearAlgebra.jl` has an implementation of a symmetric matrix called `Symmetric` for example. The versions of these matrices in `GeometricMachineLearning` are however more memory efficient as they only store as many parameters as are necessary, i.e. $n(n+1)/2$ for the symmetric matrix and $n(n-1)/2$ for the other three. In addition operations such as matrix and tensor multiplication are implemented for these matrices to work in parallel on GPU via `GeometricMachineLearning.tensor_mat_mul` for example. We here give an overview of *elementary* custom matrices that are implemented in `GeometricMachineLearning`. More *involved* matrices are the so-called global tangent spaces (see Chapter 2.11).

Custom Matrices

`GeometricMachineLearning` has two types of *triangular matrices*. The first one is `UpperTriangular`:

$$U = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & \ddots & & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 \end{pmatrix}.$$

And the second one is `LowerTriangular`:

$$L = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(n-1)} & 0 \end{pmatrix}.$$

An instance of `SkewSymMatrix` can be written as $A = L - L^T$ or $A = U^T - U$:

$$A = \begin{pmatrix} 0 & -a_{21} & \cdots & -a_{n1} \\ a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(n-1)} & 0 \end{pmatrix}.$$

And lastly a `SymmetricMatrix`:

$$B = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n(n-1)} & a_{nn} \end{pmatrix}.$$

Note that any matrix $M \in \mathbb{R}^{n\times n}$ can be written

$$M = \frac{1}{2}(M - M^T) + \frac{1}{2}(M + M^T),$$

where the first part of this matrix is skew-symmetric and the second part is symmetric. This is also how the constructors for `SkewSymMatrix` and `SymmetricMatrix` are designed. Consider an arbitrary matrix:

```
M = [1; 2; 3;; 4; 5; 6;; 7; 8; 9]
```

```
3×3 Matrix{Int64}:
 1  4  7
 2  5  8
 3  6  9
```

Calling `SkewSymMatrix` on M is equivalent to doing $M \to \frac{1}{2}(M - M^T)$:

```
A = SkewSymMatrix(M)
```

```
3×3 SkewSymMatrix{Float64, Vector{Float64}}:
  0.0   1.0  2.0
 -1.0   0.0  1.0
 -2.0  -1.0  0.0
```

And calling `SymmetricMatrix` on M is equivalent to doing $M \to \frac{1}{2}(M + M^T)$:

```
B = SymmetricMatrix(M)
```

```
3×3 SymmetricMatrix{Float64, Vector{Float64}}:
 1.0  3.0  5.0
 3.0  5.0  7.0
 5.0  7.0  9.0
```

We can further confirm the identity above:

```
M  ≈ A + B
```

```
true
```

Note that for `LowerTriangular` and `UpperTriangular` no projection step is involved, which means that if we start with a matrix of type `AbstractMatrix{Int64}` we will end up with a matrix that is also of type `AbstractMatrix{Int64}`. The type changes however when we call `SkewSymMatrix` and `SymmetricMatrix`:

```
(typeof(A) <: AbstractMatrix{Int64}, typeof(B) <: AbstractMatrix{Int64})
```

```
(false, false)
```

For the triangular matrices:

```
U = UpperTriangular(M)
L = LowerTriangular(M)
(typeof(U) <: AbstractMatrix{Int64}, typeof(L) <: AbstractMatrix{Int64})
```

```
(true, true)
```

How are Special Matrices Stored?

The following image demonstrates how a skew-symmetric matrix is stored in `GeometricMachineLearning`:

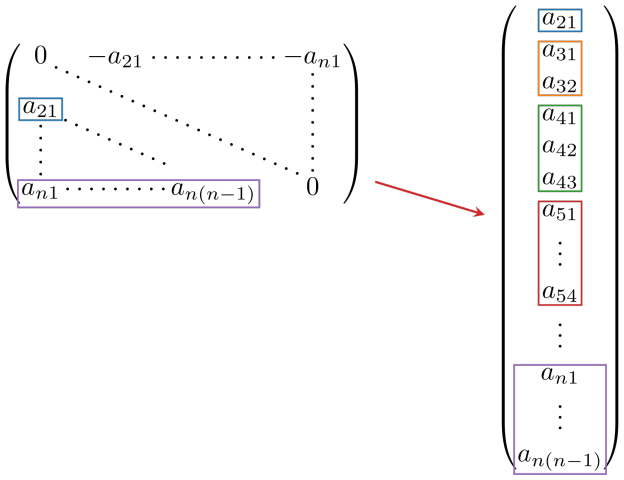


Figure B.1: The elements of a skew-symmetric matrix (and other special matrices) are stored as a vector. The elements of the big vector are the entries on the lower left of the matrix, stored row-wise.

So what is stored internally is a vector of size $n(n-1)/2$ for the skew-symmetric matrix and the triangular matrices, and a vector of size $n(n+1)/2$ for the symmetric matrix.

Sample Random Matrices

We can sample a random skew-symmetric matrix:

```
A = rand(SkewSymMatrix, 3)
```

```
3×3 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0       -0.521214  -0.586807
 0.521214   0.0       -0.890879
 0.586807   0.890879   0.0
```

and then access the vector:

```
A.S
```

```
3-element Vector{Float64}:
 0.521213795535383
 0.5868067574533484
 0.8908786980927811
```

This is equivalent to sampling a vector and then assigning a matrix[1]:

[1] We fixed the seed to the same value in both these examples.

```
S = rand(3 * (3 - 1) ÷ 2)
SkewSymMatrix(S, 3)
```

```
3×3 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0       -0.521214  -0.586807
 0.521214   0.0       -0.890879
 0.586807   0.890879   0.0
```

These special matrices are important for SympNets (see Chapter 8.5), volume-preserving transformers (see Chapter 8.8) and linear symplectic transformers (see Chapter 8.9).

Parallel Computation

The functions `GeometricMachineLearning.mat_tensor_mul` and `GeometricMachineLearning.tensor_mat_mul` are also implemented for these matrices for efficient parallel computations. This is elaborated on when we take about tensors (see Chapter B.2).

Library Functions

`GeometricMachineLearning.AbstractTriangular` – Type.

```
AbstractTriangular
```

See `UpperTriangular` and `LowerTriangular`.

source

`GeometricMachineLearning.UpperTriangular` – Type.

```
UpperTriangular(S::AbstractVector, n::Int)
```

Build an upper-triangular matrix from a vector.

An upper-triangular matrix is an $n \times n$ matrix that has zeros on the diagonal and on the lower triangular.

The data are stored in a vector S similarly to other matrices. See `LowerTriangular`, `SkewSymMatrix` and `SymmetricMatrix`.

The struct two fields: `S` and `n`. The first stores all the entries of the matrix in a sparse fashion (in a vector) and the second is the dimension n for $A \in \mathbb{R}^{n\times n}$.

Examples

```
using GeometricMachineLearning
S = [1, 2, 3, 4, 5, 6]
UpperTriangular(S, 4)

# output
```

```
4×4 UpperTriangular{Int64, Vector{Int64}}:
 0  1  2  4
 0  0  3  5
 0  0  0  6
 0  0  0  0
```

source

GeometricMachineLearning.UpperTriangular – Method.

```
UpperTriangular(A::AbstractMatrix)
```

Build an upper-triangular matrix from a matrix.

This is done by taking the upper right of that matrix.

Examples

```
using GeometricMachineLearning
M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
UpperTriangular(M)

# output

4×4 UpperTriangular{Int64, Vector{Int64}}:
 0  2  3   4
 0  0  7   8
 0  0  0  12
 0  0  0   0
```

source

GeometricMachineLearning.LowerTriangular – Type.

```
LowerTriangular(S::AbstractVector, n::Int)
```

Build a lower-triangular matrix from a vector.

A lower-triangular matrix is an $n \times n$ matrix that has zeros on the diagonal and on the upper triangular.

The data are stored in a vector S similarly to other matrices. See `UpperTriangular`, `SkewSymMatrix` and `SymmetricMatrix`.

The struct two fields: `S` and `n`. The first stores all the entries of the matrix in a sparse fashion (in a vector) and the second is the dimension n for $A \in \mathbb{R}^{n\times n}$.

Examples

```
using GeometricMachineLearning
S = [1, 2, 3, 4, 5, 6]
LowerTriangular(S, 4)

# output

4×4 LowerTriangular{Int64, Vector{Int64}}:
 0  0  0  0
 1  0  0  0
 2  3  0  0
 4  5  6  0
```

source

`GeometricMachineLearning.LowerTriangular` – Method.

```
LowerTriangular(A::AbstractMatrix)
```

Build a lower-triangular matrix from a matrix.

This is done by taking the lower left of that matrix.

Examples

```
using GeometricMachineLearning
M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
LowerTriangular(M)

# output

4×4 LowerTriangular{Int64, Vector{Int64}}:
  0   0   0  0
  5   0   0  0
  9  10   0  0
 13  14  15  0
```

source

`Base.vec` – Method.

```
vec(A::AbstractTriangular)
```

Return the associated vector to A.

Examples

```
using GeometricMachineLearning

M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
LowerTriangular(M) |> vec
```

```
# output

6-element Vector{Int64}:
  5
  9
 10
 13
 14
 15
```

source

`GeometricMachineLearning.SkewSymMatrix` – Type.

```
SkewSymMatrix(S::AbstractVector, n::Integer)
```

Instantiate a skew-symmetric matrix with information stored in vector `S`.

A skew-symmetric matrix A is a matrix $A^T = -A$.

Internally the `struct` saves a vector S of size $n(n-1) \div 2$. The conversion is done the following way:

$$[A]_{ij} = \begin{cases} 0 & \text{if } i = j \\ S[((i-2)(i-1)) \div 2 + j] & \text{if } i > j \\ S[((j-2)(j-1)) \div 2 + i] & \text{else.} \end{cases}$$

So S stores a string of vectors taken from A: $S = [\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_n]$ with $\tilde{a}_i = [[A]_{i1}, [A]_{i2}, \dots, [A]_{i(i-1)}]$.

Also see `SymmetricMatrix`, `LowerTriangular` and `UpperTriangular`.

Examples

```
using GeometricMachineLearning
S = [1, 2, 3, 4, 5, 6]
SkewSymMatrix(S, 4)

# output

4×4 SkewSymMatrix{Int64, Vector{Int64}}:
 0  -1  -2  -4
 1   0  -3  -5
 2   3   0  -6
 4   5   6   0
```

source

`GeometricMachineLearning.SkewSymMatrix` – Method.

```
SkewSymMatrix(A::AbstractMatrix)
```

Perform `0.5 * (A - A')` and store the matrix in an efficient way (as a vector with $n(n-1)/2$ entries).

If the constructor is called with a matrix as input it returns a skew-symmetric matrix via the projection:

$$A \mapsto \frac{1}{2}(A - A^T).$$

Examples

```
using GeometricMachineLearning
M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
SkewSymMatrix(M)

# output

4×4 SkewSymMatrix{Float64, Vector{Float64}}:
 0.0  -1.5  -3.0  -4.5
 1.5   0.0  -1.5  -3.0
 3.0   1.5   0.0  -1.5
 4.5   3.0   1.5   0.0
```

Extended help

Note that the constructor is designed in such a way that it always returns matrices of type `SkewSymMatrix{<:AbstractFloat}` when called with a matrix, even if this matrix is of type `AbstractMatrix{<:Integer}`.

If the user wishes to allocate a matrix `SkewSymMatrix{<:Integer}` then call:

```
SkewSymMatrix(::AbstractVector, n::Integer)
```

Note that this is different from `LowerTriangular` and `UpperTriangular` as no porjection takes place there.

source

`GeometricMachineLearning.SymmetricMatrix` – Type.

```
SymmetricMatrix(S::AbstractVector, n::Integer)
```

Instantiate a symmetric matrix with information stored in vector `S`.

A `SymmetricMatrix` A is a matrix $A^T = A$.

Internally the `struct` saves a vector S of size $n(n+1) \div 2$. The conversion is done the following way:

$$[A]_{ij} = \begin{cases} S[((i-1)i) \div 2 + j] & \text{if } i \geq j \\ S[((j-1)j) \div 2 + i] & \text{else.} \end{cases}$$

So S stores a string of vectors taken from A: $S = [\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_n]$ with $\tilde{a}_i = [[A]_{i1}, [A]_{i2}, \dots, [A]_{ii}]$.

Also see `SkewSymMatrix`, `LowerTriangular` and `UpperTriangular`.

Examples

```
using GeometricMachineLearning
S = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
SymmetricMatrix(S, 4)

# output

4×4 SymmetricMatrix{Int64, Vector{Int64}}:
 1  2  4   7
 2  3  5   8
 4  5  6   9
 7  8  9  10
```

source

`GeometricMachineLearning.SymmetricMatrix` – Method.

```
SymmetricMatrix(A::AbstractMatrix)
```

Perform a projection and store the matrix in an efficient way (as a vector with $n(n+1)/2$ entries).

If the constructor is called with a matrix as input it returns a symmetric matrix via the *projection*:

$$A \mapsto \frac{1}{2}(A + A^T).$$

Examples

```
using GeometricMachineLearning
M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
SymmetricMatrix(M)

# output

4×4 SymmetricMatrix{Float64, Vector{Float64}}:
 1.0   3.5   6.0   8.5
 3.5   6.0   8.5  11.0
 6.0   8.5  11.0  13.5
 8.5  11.0  13.5  16.0
```

Extended help

Note that the constructor is designed in such a way that it always returns matrices of type `SymmetricMatrix{<:AbstractFloat}` when called with a matrix, even if this matrix is of type `AbstractMatrix{<:Integer}`.

If the user wishes to allocate a matrix `SymmetricMatrix{<:Integer}` then call

$julia SymmetricMatrix(:: AbstractVector, n :: Integer)$‘

Note that this is different from `LowerTriangular` and `UpperTriangular` as no porjection takes place there.

source

`Base.vec` – Method.

```
vec(A)
```

Output the associated vector of `A`.

Examples

```
using GeometricMachineLearning

M = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
SkewSymMatrix(M) |> vec

# output

6-element Vector{Float64}:
 1.5
 3.0
 1.5
 4.5
 3.0
 1.5
```

source

B.2 Tensors in `GeometricMachineLearning`

We typically store training data as *tensors with three axes* in `GeometricMachineLearning`. This allows for a parallel computation of matrix products, also for the special arrays such as `LowerTriangular`, `UpperTriangular`, `SymmetricMatrix` and `SkewSymMatrix` and objects of `Manifold` type such as the `StiefelManifold`.

Library Functions

`GeometricMachineLearning.tensor_mat_mul` – Method.

```
tensor_mat_mul(A::AbstractArray{<:Number, 3}, B::AbstractMatrix)
```

Multipliy the matrix `B` onto the tensor `A` from the right.

Internally this calls the inplace version `tensor_mat_mul!`.

Examples

```
using GeometricMachineLearning: tensor_mat_mul

A = [1 1 1; 1 1 1; 1 1 1;;; 2 2 2; 2 2 2; 2 2 2]
B = [3 0 0; 0 2 0; 0 0 1]

tensor_mat_mul(A, B)

# output

3×3×2 Array{Int64, 3}:
[:, :, 1] =
 3  2  1
 3  2  1
 3  2  1

[:, :, 2] =
 6  4  2
 6  4  2
 6  4  2
```

source

`GeometricMachineLearning.tensor_mat_mul!` – Method.

```
tensor_mat_mul!(C, A, B)
```

Multiply the matrix `B` onto the tensor `A` from the right and store the result in `C`.

The function `tensor_mat_mul` calls `tensor_mat_mul!` internally.

source

`GeometricMachineLearning.tensor_mat_mul!` – Method.

```
mat_tensor_mul!(C::AbstractArray{<:Number, 3}, B::AbstractArray{<:Number, 3},
↪  A::SymmetricMatrix)
```

Multiply the symmetric matrix `A` onto the tensor `B` from the right and store the result in `C`.

This performs an efficient multiplication based on the special structure of the symmetric matrix `A`.

source

`GeometricMachineLearning.mat_tensor_mul` – Method.

```
mat_tensor_mul(A, B)
```

Multipliy the matrix `A` onto the tensor `B` from the left.

Internally this calls the inplace version `mat_tensor_mul!`.

Examples

```
using GeometricMachineLearning: mat_tensor_mul

B = [1 1 1; 1 1 1; 1 1 1;;; 2 2 2; 2 2 2; 2 2 2]
A = [3 0 0; 0 2 0; 0 0 1]

mat_tensor_mul(A, B)

# output

3×3×2 Array{Int64, 3}:
[:, :, 1] =
 3  3  3
 2  2  2
 1  1  1

[:, :, 2] =
 6  6  6
 4  4  4
 2  2  2
```

source

`GeometricMachineLearning.mat_tensor_mul!` – Method.

```
mat_tensor_mul!(C, A, B)
```

Multiply the matrix `A` onto the tensor `B` from the left and store the result in `C`.

The function `mat_tensor_mul` calls `mat_tensor_mul!` internally.

source

`GeometricMachineLearning.mat_tensor_mul!` – Method.

```
mat_tensor_mul!(C, A::LowerTriangular, B)
```

Multiply the lower-triangular matrix `A` onto the tensor `B` from the left and store the result in `C`.

This performs an efficient multiplication based on the special structure of the lower-triangular matrix `A`.

source

`GeometricMachineLearning.mat_tensor_mul!` – Method.

```
mat_tensor_mul!(C, A::UpperTriangular, B)
```

Multiply the upper-triangular matrix `A` onto the tensor `B` from the left and store the result in `C`.

This performs an efficient multiplication based on the special structure of the upper-triangular matrix `A`.

source

`GeometricMachineLearning.mat_tensor_mul!` – Method.

```
mat_tensor_mul!(C, A::SkewSymMatrix, B)
```

Multiply skew-symmetric the matrix `A` onto the tensor `B` from the left and store the result in `C`.

This performs an efficient multiplication based on the special structure of the skew-symmetric matrix `A`.

source

`GeometricMachineLearning.mat_tensor_mul!` – Method.

```
mat_tensor_mul!(C, A::SymmetricMatrix, B)
```

Multiply the symmetric matrix `A` onto the tensor `B` from the left and store the result in `C`.

This performs an efficient multiplication based on the special structure of the symmetric matrix `A`.

source

B.3 Pullbacks and Automatic Differentiation

Automatic Differentiation is an important part of modern machine learning. It is essentially a tool to compute the gradient of a loss function with respect to its input arguments, i.e. given a function $L : \Theta \to \mathbb{R}$ an AD routine computes:

$$\text{AD} : \theta \mapsto \nabla_\theta L.$$

When we train a neural network the function L is the composition of a neural network

$$\mathcal{NN}_\theta(x) = l^{(n)}_{\theta_n} \circ l^{(1)}_{\theta_1}(x)$$

and a loss function (see Chapter 4.4), so we have:

$$L(\theta) = \texttt{loss}(l^{(n)}_{\theta_n} \circ l^{(1)}_{\theta_1}(x), \texttt{minibatch})$$

where $\theta = (\theta_1, \dots, \theta_n)$ and L further depends on a specific mini batch here. So what we need to do is compute the pullback[1] of every single layer $l^{(i)}_{\theta_i}$. For this consider a function[2] $f : \mathbb{R}^m \to \mathbb{R}^n$ and $x \in \mathbb{R}^m$. The pullback of f is a function that, depending on x, provides a recipe to map an element from $\mathbb{R}^n$ to an element of $\mathbb{R}^m$:

$$\text{pullbak}(f)[x] : \mathbb{R}^m \simeq T^*_{f(x)}\mathbb{R}^m \to T^*_x\mathbb{R}^n \simeq \mathbb{R}^n,$$

where $T^*_x\mathcal{V}$ is the cotangent space of $\mathcal{V}$ at x.

How to Compute Pullbacks

`GeometricMachineLearning` has many pullbacks for custom array types and other operations implemented. The need for this essentially comes from the fact that we cannot trivially differentiate custom GPU kernels[3]. Implemented custom pullback comprise parallel multiplications with tensors (see Chapter B.2).

What is a Pullback?

Here we first explain the principle of a pullback with the example of a vector-valued function. The generalization to matrices and higher-order tensors is straight-forward.

The pullback of a vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$ can be interpreted as the *sensitivities in the input space* $\mathbb{R}^n$ with respect to variations in the output space $\mathbb{R}^m$ via the function f:

$$[\text{pullback}(f)[a \in \mathbb{R}^n, db \in \mathbb{R}^m]]_i = \sum_{j=1}^{m} \frac{\partial f_j}{\partial a_i} db_j.$$

This principle can easily be generalized to matrices. For this consider the function $g :: \mathbb{R}^{n_1 \times n_2} \to \mathbb{R}^{m_1 \times m_2}$. We then have:

$$\left[\text{pullback}(g)[A \in \mathbb{R}^{n_1 \times n_2}, dB \in \mathbb{R}^{m_1 \times m_2}]\right]_{(i_1,i_2)} = \sum_{j_1=1}^{m_1} \sum_{j_2=1}^{m_2} \frac{\partial f_{(j_1,j_2)}}{\partial a_{(i_1,i_2)}} db_{(j_1,j_2)}.$$

The generalization to higher-order tensors is again straight-forward.

Illustrative example

Consider the matrix inverse $\text{inv} : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ as an example. This fits into the above framework where *inv* is a matrix-valued function from $\mathbb{R}^{n\times n}$ to $\mathbb{R}^{n\times n}$. We here write $B := A^{-1} = \text{inv}(A)$. We thus have to compute:

[1] The term *pullback* originally comes from differential geometry [36]. This motivation is discussed below.

[2] $\mathbb{R}^m$ can be interpreted as the space of the neural network parameters Θ here.

[3] This may change in the future if the package `Enzyme` [113] reaches maturity.

$$\left[\text{pullback(inv)}[A \in \mathbb{R}^{n\times n}, dB \in \mathbb{R}^{n\times n}]\right]_{(i,j)} = \sum_{k=1}^{n}\sum_{\ell=1}^{n} \frac{\partial b_{k,\ell}}{\partial a_{i,j}} db_{k,\ell}.$$

For a matrix A that depends on a parameter ε we have:

$$\frac{\partial}{\partial \varepsilon} B = -B \left(\frac{\partial}{\partial \varepsilon} A\right) B.$$

This can easily be checked:

$$\mathbb{O} = \frac{\partial}{\partial \varepsilon}\mathbb{I} = \frac{\partial}{\partial \varepsilon}(AB) = A\frac{\partial}{\partial \varepsilon}B + \left(\frac{\partial}{\partial \varepsilon}A\right) B.$$

We can then write:

$$\begin{aligned}
\sum_{k,\ell}\left(\frac{\partial}{\partial a_{ij}} b_{k\ell}\right) db_{k\ell} &= \sum_{k\ell}\left[\frac{\partial}{\partial a_{ij}} B\right]_{k\ell} db_{k,\ell} \\
&= -\sum_{k,\ell}\left[B\left(\frac{\partial}{\partial a_{ij}} A\right) B\right]_{k\ell} db_{k\ell} \\
&= -\sum_{k,\ell,m,n} b_{km}\left(\frac{\partial a_{mn}}{\partial a_{ij}}\right) b_{n\ell} db_{k\ell} \\
&= -\sum_{k,\ell,m,n} b_{km}\delta_{im}\delta_{jn} b_{n\ell} db_{k\ell} \\
&= -\sum_{k,\ell} b_{ki} b_{j\ell} db_{k\ell} \\
&\equiv -B^T \cdot dB \cdot B^T.
\end{aligned}$$

We use this expression to differentiate the `VolumePreservingAttention` layer.

Motivation from a differential-geometric perspective

The notion of a *pullback in automatic differentiation* is motivated by the concept of *pullback in differential geometry* [114, 115]. In both cases we want to compute, based on a mapping

$$f : \mathcal{V} \to \mathcal{W}, a \mapsto f(a) =: b,$$

a *map of differentials* $db \mapsto da$. In the differential geometry case db and da are part of the associated cotangent spaces, i.e. $db \in T_b^*\mathcal{W}$ and $da \in T_a^*\mathcal{V}$; in AD we (mostly) deal with spaces of arrays, i.e. vector spaces, which means that $T_b^*\mathcal{W} \simeq \mathcal{W}$ and $T_a^*\mathcal{V} \simeq \mathcal{V}$. If we have neural network weights on manifolds however, then we have to map weights from $T_a^*\mathcal{V}$ (the result of an AD routine) to $T_a\mathcal{V}$ before we can apply a retraction (see Chapter 5.2). The mapping

$$T_a^*\mathcal{V} \to T_a\mathcal{V}$$

is equivalent to applying the Riemannian gradient (see Chapter 2.7).

Library Functions

`GeometricMachineLearning.ZygotePullback` – Type.

```
ZygotePullback <: AbstractPullback
```

The pullback based on the `Zygote` backend.

Examples

For a network that is trained on inputs only:

```
using GeometricMachineLearning
using GeometricMachineLearning: _processing

loss = AutoEncoderLoss()
_pullback = ZygotePullback(loss)
nn = NeuralNetwork(Chain(Dense(10, 2, tanh), Dense(2, 10, tanh)))
input = rand(10)
_pullback(nn.params, nn.model, input)[2](1) |> _processing |> typeof

# output

@NamedTuple{L1::@NamedTuple{W::Matrix{Float64}, b::Vector{Float64}},
↪  L2::@NamedTuple{W::Matrix{Float64}, b::Vector{Float64}}}
```

In this example `_processing` is used to get around some `Zygote` quirks.

source

References

[114] M. Betancourt. *A geometric theory of higher-order automatic differentiation*, arXiv preprint arXiv:1812.11592 (2018).

[115] J. Bolte and E. Pauwels. *A mathematical model for automatic differentiation in machine learning*. Advances in Neural Information Processing Systems **33**, 10809–10819 (2020).

Appendix C

Customizing Training

A neural network framework can be seen as a collection of (i) a neural network architecture, (ii) a loss function and (iii) an optimization procedure. In this dissertation we focused on points (i) and (iii) when designing neural networks. `GeometricMachineLearning` however also offers the possibility to change the loss function. We show how to do this here.

C.1 Adjusting the Loss Function

`GeometricMachineLearning` provides a few standard loss functions that are used as defaults for specific neural networks (see Chapter 4.4).

If these standard losses do not satisfy the user's needs, it is very easy to implement custom loss functions. Adding terms to the loss function is standard practice in machine learning to either increase stability [1] or to *inform* the network about physical properties[1] [7].

We again consider training a SympNet on the data coming from a harmonic oscillator:

```
using GeometricProblems.HarmonicOscillator: hodeproblem

sol = integrate(hodeproblem(; timespan = 100), ImplicitMidpoint())
data = DataLoader(sol; suppress_info = true)

nn = NeuralNetwork(GSympNet(2))

# train the network
o = Optimizer(AdamOptimizer(), nn)
batch = Batch(32)
n_epochs = 30
loss = FeedForwardLoss()
loss_array = o(nn, data, batch, n_epochs, loss; show_progress = false)
print(loss_array[end])
```

```
0.0026214084811743707
```

And we see that the loss goes down to a very low value. But the user might want to constrain the norm of the network parameters:

[1] Note however that we discourage using so-called physics-informed neural networks (see Chapter 4.4) as they do not preserve any physical properties but only give a potential improvement on stability in the region where we have training data.

```
# norm of parameters for single layer
network_parameter_norm(params::NamedTuple) = sum([norm(params[i]) for i in 1:length(params)])
# norm of parameters for entire network
function network_parameter_norm(params::NeuralNetworkParameters)
    sum([network_parameter_norm(params[key]) for key in keys(params)])
end

network_parameter_norm(params(nn))
```

```
4.638560763436521
```

We now implement a custom loss such that:

$$\text{loss}_{\mathcal{NN}}^{\text{custom}}(\text{input}, \text{output}) = \text{loss}_{\mathcal{NN}}^{\text{feedforward}} + \lambda \text{norm}(\mathcal{NN}.\texttt{params}).$$

```
struct CustomLoss <: GeometricMachineLearning.NetworkLoss end

const λ = .1
function (loss::CustomLoss)(model::Union{AbstractExplicitLayer, Chain},
↪   params::Union{NeuralNetworkParameters, NamedTuple}, input::QPTOAT, output::QPTOAT)
    FeedForwardLoss()(model, params, input, output) + λ * network_parameter_norm(params)
end
```

And we train the same network with this new loss:

```
loss = CustomLoss()
nn_custom = NeuralNetwork(GSympNet(2))
loss_array = o(nn_custom, data, batch, n_epochs, loss; show_progress = false)
print(loss_array[end])
```

```
0.162827448411511
```

We see that the norm of the parameters is lower:

```
network_parameter_norm(params(nn_custom))
```

```
2.096776248375609
```

We can also compare the solutions of the two networks:

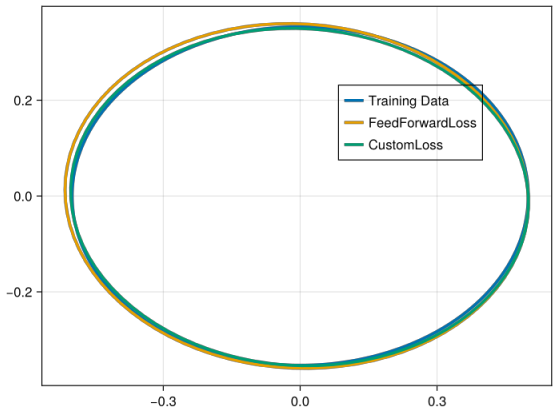


Figure C.1: Here we trained the same network with two different losses.

Wit the second loss function, for which the norm of the resulting network parameters has lower value, the network still performs well, albeit slightly worse than the network trained with the first loss.

References

[7] M. Raissi, P. Perdikaris and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational physics **378**, 686–707 (2019).

Appendix D

Other Structure-Preserving Properties

When talking about *structure* we largely limited ourselves to *symplecticity* and *volume preservation* in this dissertation[1]. A more general type of structure is that of a *port-Hamiltonian system*. In addition to a conservative Hamiltonian part it also includes external (driving) forces and a dissipative part.

D.1 Using Symplectic Autoencoders for Port-Hamiltonian Systems

Symplectic autoencoders can also be used to reduce *port-Hamiltonian systems* [100]. Here we focus on *linear port-Hamiltonian systems*[1]. These are of the form:

$$\Sigma_{\text{lpH}}(\mathbb{R}^{2N}) = \Sigma_{\text{lpH}} : \begin{cases} \dot{\hat{z}}(t) &= (\mathbb{J}_{2N} - \hat{R})\nabla H(\hat{z}(t)) + \hat{B}u(t) \\ y(t) &= \hat{B}^T \nabla H(\hat{z}(t)), \end{cases}$$

where $\mathbb{J}_{2N}$ is the Poisson tensor (see Chapter 3.1) and $\hat{R} \in \mathbb{R}^{2N\times 2N}$ is symmetric semi-positive definite (i.e. all its eigenvalues are non-negative). $\hat{z} \in \mathbb{R}^{2N}$ is called the *state of the system*, $u \in \mathbb{R}^m$ are the *system inputs*, $y \in \mathbb{R}^m$ are the *system outputs*, and $\hat{B} \in \mathbb{R}^{2N\times m}$ connects the inputs to the state. We also refer to *linear port-Hamiltonian systems* as *lpH systems*.

Similar to energy conservation of standard Hamiltonian systems, lpH systems have an associated *energy balance equation*:

Definition D.1.1. The **energy balance equation** of a lpH system is:

$$\frac{d}{dt}H(z(t)) \leq y(t)^T u(t).$$

If we further have $R = 0$, then the inequality becomes an equality.

Proof. We evaluate the derivative of $H(z(t))$ with respect to t:

[1] The orthogonality constraints imposed on the vision transformer can however also be seen as structure.

[1] For a broader class of such systems see [116]. A generalization to manifolds of such systems is also possible [117, 118].

$$\begin{aligned}\frac{d}{dt}H(\varphi^t(z_0)) &= \nabla H(\varphi^t(z_0))\frac{d}{dt}\varphi^t(z_0)\\ &= \nabla H(z(t))(\mathbb{J}-R)\nabla H(z(t)) + (\nabla H(z(t)))^T Bu(t)\\ &= (\nabla H(z(t)))^T R\nabla H(z(t)) + \underbrace{(\nabla H(z(t)))^T B}_{y(t)^T} u(t)\\ &\leq y(t)^T u(t),\end{aligned}$$

where we used that R is symmetric and positive semi-definite in the last step. □

The analogue to the Poisson tensor (see Chapter 3.1) for lpH systems are so-called *Dirac structures*:

Definition D.1.2. A Dirac structure for a vector space $\mathbb{R}^n$ is a subspace $D \subset \mathbb{R}^n \times (\mathbb{R}^n)^* \simeq \mathbb{R}^{2n}$ such that

$$D^\perp = D,$$

i.e. the orthogonal complement of D is equal to itself. Here the orthogonal complement is taken with respect to the pairing:

$$\langle\!\langle \cdot, \cdot \rangle\!\rangle : \mathbb{R}^{2n} \times \mathbb{R}^{2n} \to \mathbb{R}, (e, f) \times (\tilde{e}, \tilde{f}) \mapsto e^T\tilde{f} + \tilde{e}^T f.$$

Note that $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ is a symmetric bilinear form.

Example D.1.3. The space:

$$D = \{(e, f) : e \in \mathbb{R}^{2n}, f = \mathbb{J}_{2n}e\} \subset \mathbb{R}^{4n}$$

forms a Dirac structure. Note that we also have:

$$(\nabla_z H, X_H(z)) = (\nabla_z H, \mathbb{J}\nabla_z H) \in D.$$

So every Hamiltonian System has an associated Dirac structure.

Example D.1.4. For the lpH shown above we have the relation:

$$\begin{pmatrix} f \\ y \\ e \end{pmatrix} = \begin{pmatrix} \mathbb{J}_{2N}^T & -B & -\mathbb{I}_{2N} \\ B^T & \mathbb{O} & \mathbb{O} \\ \mathbb{I}_{2N} & \mathbb{O} & \mathbb{O} \end{pmatrix} \begin{pmatrix} \bar{e} \\ u \\ \bar{\bar{e}} \end{pmatrix},$$

where we further have the constraints and identifications $f = -\dot{z}$, $\bar{e} = \nabla_z H$ and $\bar{\bar{e}} = -Re$ to fully describe the lpH. Note that points $(f, y, e, \bar{e}, u, \bar{\bar{e}}) \in \mathbb{R}^{8N+2m}$ that satisfy the relation shown above form a Dirac structure.

In numerically solving lpH systems the Dirac structure takes a similar role to the symplectic structure of canonical Hamiltonian systems [119] and the energy-balance equation takes a similar role to energy conservation for canonical Hamiltonian systems. Structure-preserving discretization of lpH systems discretize the Dirac structure with collocation methods [120]. A one-step update can be written as

$$
\begin{aligned}
z_f &= z_0 + h \sum_{i=1}^{s} \beta_i f_i, \\
z_i &= z_0 + h \sum_{j=1}^{s} \alpha_{ij} f_j, \\
e_i &= \bar{e}_i = \nabla_{z_i} H, \\
\bar{\bar{e}}_i &= -R e_i, \\
f_i &= \mathbb{J}_{2N}^T \bar{e}_i - B u_i - \bar{\bar{e}}_i, \\
y_i &= B^T \bar{e}_i.
\end{aligned}
$$

Model order reduction of port-Hamiltonian systems can be divided into two approaches: *projection-based methods* and *interpolations of the transfer function* [103]. The first approach is equivalent to Galerkin projection (see Chapter 4.1) and we limit the discussion here to this approach. Similar to the case of canonical Hamiltonian systems (see Chapter 4.5), we reduce the system with a symplectic autoencoder (see Chapter 8.2).

When discussing symplectic model order reduciton (see Chapter 4.5) we showed that a Hamiltonian system on the reduced space $\mathbb{R}^{2n}$ is equivalent to a Hamiltonian system on $\mathcal{M} = \mathcal{R}(\mathbb{R}^{2n})$, where $\mathcal{R}$ is the *reconstruction* in a reduced order modeling framework. Similar statements are also true for lpH systems.

We will now demonstrate how to obtain a reduced-order lpH system from a full-order lpH system and vice-versa:

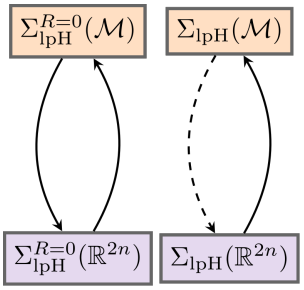


Figure D.1: We can derive full lpH systems from reduced lpH systems and vice-versa (in some cases). The solid arrows indicate that we have an explicit construction available, the dashed arrow indicates that in this specific case we do not yet know if a structure-preserving reduction is possible.

The figure above indicates that we can derive a full system $\tilde{\Sigma}_{\text{lpH}}(\mathbb{R}^{2N}) := \Sigma_{\text{lpH}}(\mathcal{M})$ from a reduced one $\Sigma_{\text{lpH}}(\mathbb{R}^{2n})$. If we have $R = 0$, i.e. if the dissipative part of the system is zero, then we can also derive a reduced system $\Sigma_{\text{lpH}}^{R=0}(\mathbb{R}^{2n})$ from a full one $\tilde{\Sigma}_{\text{lpH}}^{R=0}(\mathbb{R}^{2N}) = \Sigma_{\text{lpH}}^{R=0}(\mathcal{M})$. If and when this is true for $R \neq 0$ is an open question[2]. We now proceed with showing this equivalence, first for the special case $R = 0$.

[2]We indicate this with a dashed arrow.

The Special Case $R = 0$

We first focus on the case where $R = 0$. This case was also discussed in [119].

Theorem D.1.5. *For $R = 0$, model reduction of a lpH system with a symplectic autoencoder (Ψ^e, Ψ^d) yields a lpH system in reduced dimension of the form:*

$$\bar{\Sigma}_{\text{lpH}} : \begin{cases} \dot{z}(t) & = \mathbb{J}_{2n}\bar{H}(z(t)) + Bu(t) \\ y(t) & = B^T\nabla\bar{H}(z(t)) \end{cases},$$

where $B := (\nabla\Psi^d)^+\hat{B}$ and $\bar{H}(z(t)) := H(\Psi^d(z(t)))$.

$(\nabla\Psi^d)^+ = \mathbb{J}_{2n}^T(\nabla\Psi^d)^T\mathbb{J}_{2N}$ is the symplectic inverse (see Chapter 4.5).

Proof. We have to proof that the dynamics of $\Psi^d(z)$, that approximate $\hat{z}$, are described by a lpH system. We first insert $\bar{z}(t) \approx \Psi^d(z(t))$ into the first equation of Σ_{lpH}:

$$(\nabla\Psi^d)\dot{z}(t) = \mathbb{J}_{2N}\nabla H(\Psi^d(z(t))) + Bu(t).$$

We then multiply the equation above with the *symplectic inverse* $(\nabla\Psi^d)^+$:

$$\begin{aligned} \dot{z}(t) &= (\nabla\Psi^d)^+\mathbb{J}_{2N}\nabla H(\Psi^d(z(t))) + (\nabla\Psi^d)^+\hat{B}u(t) \\ &= \mathbb{J}_{2n}(\nabla\Psi^d)^T\nabla H(\Psi^d(z(t))) + (\nabla\Psi^d)^+\hat{B}u(t) \\ &= \mathbb{J}_{2n}\nabla(H \circ \Psi^d(z(t))) + Bu(t), \end{aligned}$$

thus proving our assertion. □

From the Reduced Space to the Full Space for $R \neq 0$

Here we show that we can construct a lpH system on the full space from a lpH system on the reduced space; this holds for both $R = 0$ and $R \neq 0$. The corresponding proof was already introduced in similar form by [121].

Theorem D.1.6. *A lpH system on the reduced space induces a lpH system on the full space.*

Proof. Consider a reduced lpH system:

$$\Sigma_{\text{lpH}} : \begin{cases} \dot{z}(t) & = (\mathbb{J}_{2n} - R)\nabla H(z(t)) + Bu(t) \\ y(t) & = B^T\nabla H(z(t)), \end{cases}$$

where $R \in \mathbb{R}^{2n\times 2n}$ and $B \in \mathbb{R}^{2n\times m}$. After multiplying the first equation with $\nabla_z\mathcal{R}$ from the left we get:

$$\frac{d}{dt}\mathcal{R}(z(t)) = \nabla_z\mathcal{R}(\mathbb{J}_{2n} - R)\nabla_z H + (\nabla_z\mathcal{R})Bu(t).$$

From now on we call $\tilde{B} := (\nabla_z \mathcal{R})B$. We then look at the terms (i) $(\nabla_z \mathcal{R})\mathbb{J}_{2n}\nabla_z H$ and (ii) $(\nabla_z \mathcal{R})R\nabla_z H$. The first one (i) becomes:

$$\begin{aligned}(\nabla_z \mathcal{R})\mathbb{J}_{2n}\nabla_z H &= \mathbb{J}_{2N}\mathbb{J}_{2N}^T(\nabla_z \mathcal{R})\mathbb{J}_{2n}\nabla_z H \\ &= \mathbb{J}_{2N}((\nabla_z \mathcal{R})^+)^T\nabla_z H \\ &= \mathbb{J}_{2N}\nabla_{\mathcal{R}(z)}(H \circ \psi)\end{aligned}$$

And the second one (ii) becomes:

$$\begin{aligned}(\nabla_z \mathcal{R})R\nabla_z H &= (\nabla_z \mathcal{R})R\nabla_z(H \circ \psi \circ \mathcal{R}) \\ &= (\nabla_z \mathcal{R})R(\nabla_z \mathcal{R})^T\nabla_{\mathcal{R}(z)}(H \circ \psi) \\ &=: \tilde{R}\nabla_{\mathcal{R}(z)}\bar{H}.\end{aligned}$$

We then have in total:

$$\tilde{\Sigma}_{\text{lpH}} : \begin{cases} \frac{d}{dt}\mathcal{R}(z(t)) & = (\mathbb{J}_{2N} - \tilde{B})\nabla_{\mathcal{R}(z)}\bar{H} + \tilde{B}u(t) \\ \tilde{y}(t) & = \tilde{B}^T u(t) \end{cases}$$

where $\tilde{R}|_{\mathcal{R}(z)} = (\nabla_z \mathcal{R})^T R(\nabla_z \mathcal{R})$, $\tilde{B} := (\nabla_z \mathcal{R})B$ and $\bar{H} = H \circ \psi$. □

As was already discussed in the section on Hamiltonian model order reduction (see Chapter 4.5) the encoder Ψ^e can be constructed such that it is exactly the local inverse ψ. This was done in e.g. [122]. Enforcing this for symplectic autoencoders is also straightforward (see Chapter 8.2).

References

[100] A. Van Der Schaft, D. Jeltsema and others. *Port-Hamiltonian systems theory: An introductory overview*. Foundations and Trends in Systems and Control **1**, 173–378 (2014).

[119] P. Kotyczka and L. Lefevre. *Discrete-time port-Hamiltonian systems: A definition based on symplectic integration*. Systems & Control Letters **133**, 104530 (2019).

[120] V. Mehrmann and R. Morandin. *Structure-preserving discretization for port-Hamiltonian descriptor systems*. In: *2019 IEEE 58th Conference on Decision and Control (CDC)* (IEEE, 2019); pp. 6863–6868.

[116] R. Morandin. *Modeling and numerical treatment of port-Hamiltonian descriptor systems*. Ph.D. Thesis, Technische Universität Berlin (2023).

[121] J. Rettberg, J. Kneifl, J. Herb, P. Buchfink, J. Fehr and B. Haasdonk. *Data-driven identification of latent port-Hamiltonian systems*, arXiv preprint arXiv:2408.08185 (2024).