DEPARTMENT OF MATHEMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Mathematics in Data Science

Deep Learning for Parameter Reduction on Real-World Topography Data

Author:	Sebastian Walter
Supervisor:	Prof. Hans-Joachim Bungartz
Advisor:	Dr. Anne Reinarz, Lukas Krenz
Submission Date:	May 14, 2020

DEPARTMENT OF MATHEMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Mathematics in Data Science

Deep Learning for Parameter Reduction on Real-World Topography Data

Deep Learning für Parameterreduktion von echten topografischen Daten

Author:	Sebastian Walter
Supervisor:	Prof. Hans-Joachim Bungartz
Advisor:	Dr. Anne Reinarz, Lukas Krenz
Submission Date:	May 14, 2020

I hereby declare that this thesis is my own work and that no other sources have been used except those clearly indicated and referenced.

S. Walter

Munich, May 14, 2020 Sebastian Walter

Abstract

Simulation models for, e. g., earthquakes or tsunamis heavily depend on the size of their input parameter space. Typically, this input is some topography data which naturally comes with a huge parameter space. We explore several (generative) deep learning approaches to reduce the GEBCO dataset's dimensionality while maintaining the ability to reconstruct values back in the original parameter space. The presented approaches range from straight-forward to complex autoencoders based on convolutional neural networks. We find that the deep feature consistent variational autoencoder ([Hou+16]) can achieve a high reduction in the parameter space of roughly a 100-fold while producing appropriate reconstructions of the topography. Since the model outputs are producing reasonable results when being used for a highly complex tsunami simulation application, we conclude that our model preserves key topographic attributes during the reduction and reconstruction phases.

Keywords: *Parameter Reduction, Topography, Bathymetry, Neural Network, Convolutional Neural Network, Autoencoder, Variational Autoencoder, Generative Adversarial Network, GEBCO.*

Contents

1 Introduction

Topography data is the basis for many simulation models describing physical processes like earthquakes or tsunamis. Such simulations have great benefits to many people's everyday lives and researchers are striving to continuously make these simulations even more accurate and robust. However, the sheer amount of input parameters to their models is often challenging. Even with modern advancements in computational power, many such applications are still constrained in their computational efficiency by the size of the input parameter space. Therefore, being able to find an appropriate reduction of this space is of general interest and typically greatly increases performance. However, such reductions are difficult to achieve without losing a certain degree of information.

1.1 Scope

We aim to find some mapping of a topography dataset's parameter space into a lower dimensional space, have the ability to draw samples from there and be able to provide a reasonably good reconstruction in the original parameter space. The ultimate objective, which is beyond this thesis' scope, is to apply a sampling-based uncertainty quantification approach over a tsunami simulation. This can be done on the reduced parameter space and the reconstruction ability can be leveraged to obtain results back in the original parameter space. We explore selected deep learning approaches to find a good parameter space reduction of topography data. To do this, we use several autoencoder models of increasing complexity and ultimately evaluate the reconstruction quality based on a tsunami simulation. To our best knowledge, approaches of this kind have not yet been applied to problems of such nature, i. e. to topography or bathymetry data.

In Chapter 2, we will give a thorough introduction to neural networks and convolutional neural networks, followed by a presentation of several models. The performance of these models will be evaluated on real-world topography data throughout Chapter 3. We will then look into how the best out of them performs when passing its model output into a tsunami simulation engine (Chapter 4). Finally, we will conclude with a summary of our findings in Chapter 5.

1.2 The GEBCO Dataset

We are working with a topography dataset, the GEBCO dataset.[1] The General Bathymetric Chart of the Oceans (GEBCO) is a collection of bathymetric data. Bathymetry is the study or representation of a solid surface beneath a layer of liquid, i. e. the depths of oceans and lakes fall under this definition. The publicly available GEBCO dataset is a global terrain grid at 15 arc-second intervals. The bathymetry measurements are mainly based on acoustic methods and by deducing from gravity anomalies captured by satellites. Whenever better measurements were available from existing sources, they were included. This implies having bathymetric measurements of different quality. Figure 1.1 shows a visualization of the dataset.

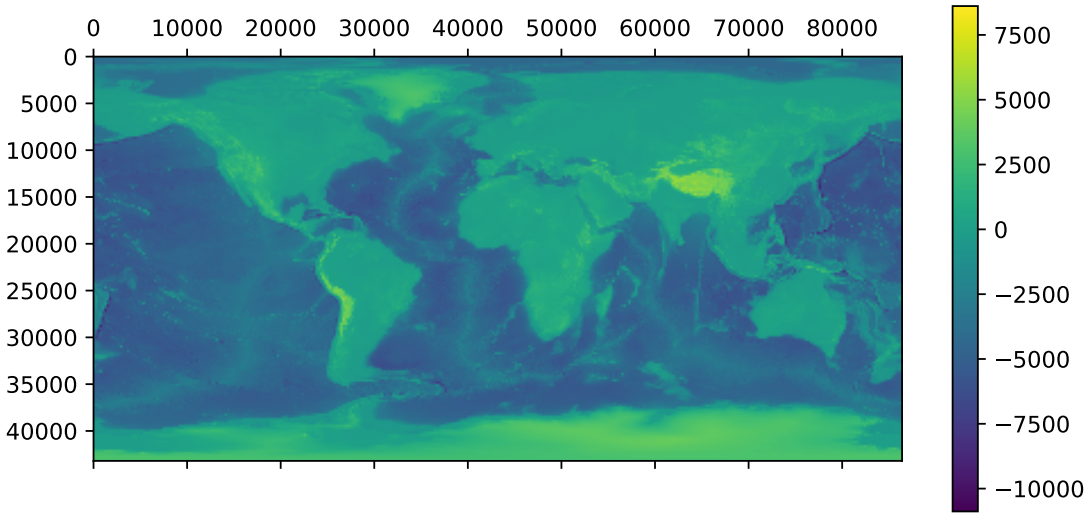

Figure 1.1 Visualizing the GEBCO dataset.

[1] GEBCO Compilation Group (2019), GEBCO_2019 Grid (doi: 10.5285/836f016a-33be-6ddc-e053-6c86abc0788e), `http://www.gebco.net`.

2 Models

The goal of this chapter is to present models of increasing complexity. First, we introduce *neural networks (NNs)* and *convolutional neural networks (CNNs)*, followed by the rather straight-forward concept of *autoencoders (AEs)* and some of its more complex extensions. We conclude this chapter with a brief introduction of the relevant theoretical aspects of *ExaHyPE* – the engine we used to derive the key results of Chapter 4.

2.1 Neural Networks

This section provides the necessary background knowledge on neural networks and most of the notations used throughout the remainder of this chapter. The content largely follows [Krö+93].

2.1.1 Perceptron

Frank Rosenblatt laid out the basis of what is nowadays known as neural networks when he introduced the perceptron in the late 1950's, [Ros58]. While it was originally intended to be an actual machine, people tend to refer to the perceptron as an architecture or algorithm.

Definition 2.1. (Perceptron)
The perceptron is defined as a simple threshold (or Heaviside) function

$$\widehat{y} = f(t) := \begin{cases} 1 & \text{if } t > 0 \\ -1 & \text{else,} \end{cases}$$

where $t = \omega^T x + b$. Here, ω is a real-valued weight vector, x the real-valued input vector and $b \in \mathbb{R}$ the bias. Often, this definition is slightly amended by augmenting x with $x_0 = 1$ and incorporating the bias term b into ω as ω_0, resulting in $x^* = (x_0, x)$, $\omega^* = (\omega_0, \omega)$ and, thus, $\widehat{y} = f\left({\omega^*}^T x^*\right)$.

The threshold function applied at the end is commonly referred to as activation function, i. e. is the output node active: yes-no. Figure 2.1 shows a nice and simple illustration of how one may think of the perceptron.

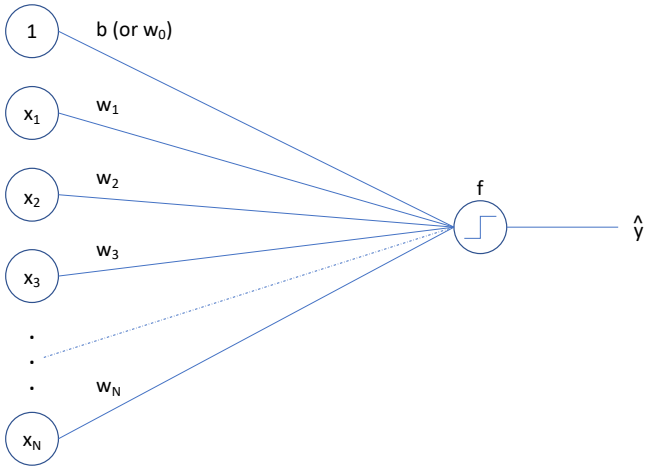

Figure 2.1 An illustration of the perceptron architecture.

While perceptron models initially yielded great results, they soon reached their own limitations at the XOR problem as shown by Minsky & Papert in [MP69]. They also showed that a model of increased complexity could overcome these limitations, however, they did not provide a solution on how to optimize the corresponding weights. The following section will introduce such a model of increased complexity, the multi-layer perceptron.

2.1.2 Multi-Layer Perceptron

The multi-layer perceptron is – as its name suggests – a collection of multiple perceptrons organized in a layered structure. First, the sigmoid activation function will be introduced, followed by a definition of the multi-layer perceptron.

Definition 2.2. (Sigmoid Activation Function)
The sigmoid activation function is defined as

$$\sigma : \mathbb{R} \to (0,1), \qquad \sigma(x) := \frac{1}{1+e^{-x}}.$$

Its derivative can be expressed in terms of the function itself:

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)\sigma(1-x).$$

Definition 2.3. (Multi-Layer Perceptron)
A multi-layer perceptron (MLP) consists of several perceptrons (also units/neurons) 'stacked' on top of each other, thus forming a layer, where we use a more general concept of activation functions (with known derivative) instead of the Heaviside function in the perceptron outputs.[1] *Every unit of a layer is connected to every unit of the layer that directly follows, but not to any other unit within its own layer. There are input, hidden and output layers. Hidden layers are located between input and output layer. This architecture is also called a* ***fully-connected feed-forward neural network****. The final output of an MLP with one hidden layer is given as*

$$\widehat{y} = g(x, \mathbf{W}) := \sigma_2 \left[W_2^T \sigma_1 \left(W_1^T x \right) \right], \quad \widehat{y} \in \mathbb{R}^{N_O}$$

where W_1, W_2 denote weight matrices, i. e. the collection of weight vectors of the input/hidden layer and the hidden/output layer, respectively. Generally speaking, while the activation functions may differ from layer to layer, they must be the same within a given layer. Otherwise units of such a layer would produce inconsistent outputs.

See e. g. [Krö+93].

This definition implies having at least one hidden layer in an MLP. Otherwise this architecture would simply collapse to having several independent perceptrons running in parallel. Again, it is helpful to graphically illustrate this architecture (Figure 2.2).

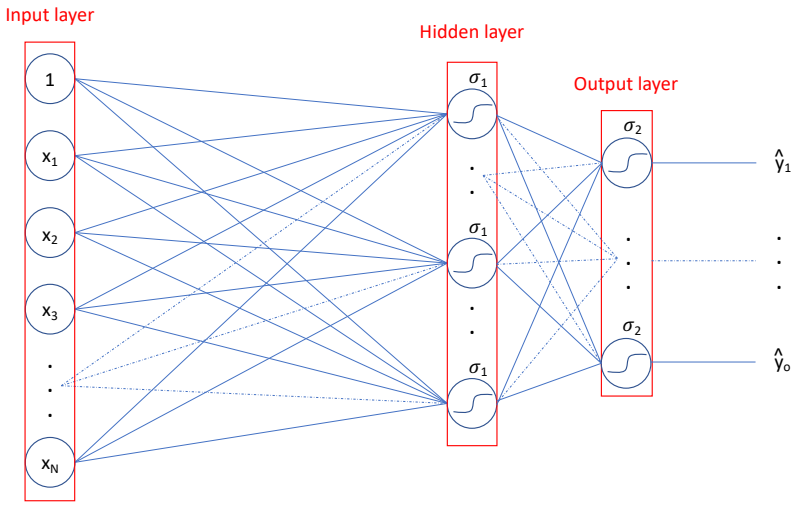

Figure 2.2 Multi-layer perceptron architecture.

[1] Traditionally, the sigmoid function was used in MLPs since it is differentiable and a good approximation of the Heaviside function. Later on, people started experimenting with other activation functions.

Clearly, we are aiming to learn the optimal weights with respect to the inputs, i. e. minimize some output error measure. However, up until the 1980's, it was not at all clear how to proceed on this matter, as a simple learning rule did not exist.

Definition 2.4. (Sum of Squared Errors)
Let the error measure at the output layer of an MLP for a given input x_n be the sum of squared errors

$$^{[n]}E := \frac{1}{2}\sum_{i=1}^{N_O}\left(^{[n]}y_i - {}^{[n]}\widehat{y}_i\right)^2,$$

where i iterates through all the units in the output layer and $^{[n]}E = E(x_n)$ denotes the error for input x_n (the same notation applies to other variables). All derivations hereafter work similarly for the cost function $J := \sum_j {}^{[j]}E$ of all training samples.

Definition 2.5. (Charbonnier Error)
The Charbonnier error is commonly defined as

$$\rho(x) = \sqrt{x^2 + \epsilon^2},$$

see e. g. [Wu+17]. ϵ controls how closely the L1 norm should be resembled while maintaining differentiability.

2.1.3 Activation Functions

There is a vast collection of activation functions, so only a few selected ones are presented here (Figure 2.3). All derivations in this paper work analogously for these functions.

Common activation functions are:

1. Sigmoid, as defined in Definition 2.2.
2. Hyperbolic tangent, $\tanh(x) = {}^{e^{2x}-1}/_{e^{2x}+1}$.
3. Rectified Linear Units (ReLU), $\mathrm{ReLU}(x) = \max\{0, x\}$.
4. Leaky ReLU, $\text{Leaky ReLU}_\alpha(x) = \max\{\alpha x, x\}$.

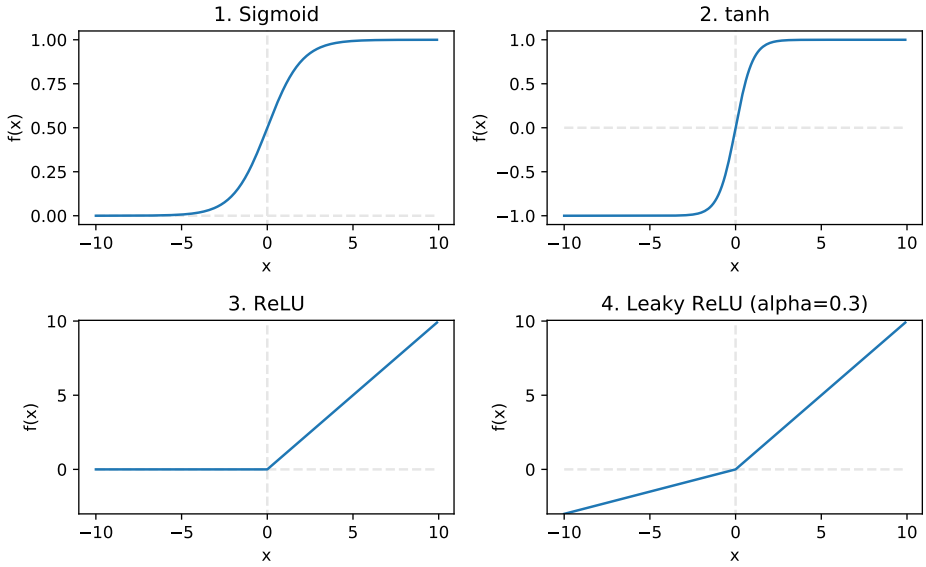

Figure 2.3 Common activation functions.

2.1.4 Backpropagation

In 1986, almost 20 years after Minsky & Papert showed that a two layer feed forward network could overcome many of the perceptron's restrictions, Rumelhart, Hinton & Williams published a simple and effective way to learn weights for an MLP, [RH86]. Similar approaches have been presented slightly earlier, e. g. [Wer74], [Par85], [LeC85].

Backpropagation – loosely speaking

Simply put, backpropagation is an automatic differentiation technique based on the chain rule from calculus, i. e. it is a method for computing gradients efficiently. Every unit in the network is visited only twice, thus the evaluation lies in $\mathcal{O}(|\mathcal{W}|)$, where $\mathcal{W}$ is the set of all the network's weights. It will become clear throughout this subsection why this is the case. We aim to propagate the error at the output layer through the hidden layer(s) back to the input layer and, by doing so, optimize the weights such that we obtain a minimal error for a given input vector.

Notations used throughout this section

An overview of notations, some of which have been introduced already:

- $N_l \in \mathbb{N},\ l \in \{1, \dots, L\}$ denotes the number of units of layer l.
- $x_n \in \mathbb{R}^{N_1}$ is one of $m \geq 1$ input vectors. For any variable, an upper left index $[n]$ indicates that this variable depends on the input x_n.

- $W^{\{l\}},\ l \in \{2, \dots, L\}$ is the weight matrix of the layer l, where $l = 1$ is the input layer and $l = L$ the output layer. Specifically, $W_{ik}^{\{l\}}$ denotes the weight connecting the previous layer's unit i with unit k of layer l. For instance, $W_{ik}^{\{L\}}$ is the weight connecting the i-th unit of the hidden layer $L-1$ with the k-th unit of the output layer L. The shape of the output layer's weight matrix is $N_{L-1} \times N_L$.
- $b_i^{\{l\}},\ l \in \{2, \dots, L\}$ is the bias at the i-th neuron of layer l.
- $W_{:,k}^{\{l\}},\ l \in \{2, \dots, L\}$ is the weight vector representing the collection of weights connecting every unit of layer $l-1$ to unit k of layer l.
- $\sigma(x)$ denotes the (element-wise) sigmoid activation function defined in Definition 2.2.
- ${}^{[n]}a_i^{\{l\}},\ l \in \{2, \dots, L\}$ is the value at node i of layer l **before** applying σ. Often this is referred to as the layer's activation.
- ${}^{[n]}z_i^{\{l\}},\ l \in \{2, \dots, L\}$ is the value at node i of layer l **after** applying σ. For $l = L$ this coincides with the model output at that particular neuron.
- ${}^{[n]}\widehat{y} \in \mathbb{R}^{N_L}$ is the vector of outputs at the output layer.
- ${}^{[n]}E$ is the error at the output layer, defined in Definition 2.4.

For the remainder of this subsection, assume we are given a neural network (NN) with $L = 3$ layers, i. e. only one hidden layer. Then, N_1 denotes the number of units in the input layer, N_2 the ones in the hidden layer and N_3 in the output layer. For simplicity, all layers use the sigmoid activation function, however, the derivation works analogously for other activation functions (the concept of sub-differentials may be required).

Using the above notations, the output at unit k of this NN can be rewritten as

$$\begin{aligned}
{}^{[n]}\widehat{y}_k &= \sigma\left[{W_{:,k}^{\{L\}}}^T\ {}^{[n]}z_i^{\{L-1\}} + b_k^{\{L\}}\right] \\
&= \sigma\left[{W_{:,k}^{\{L\}}}^T \sigma\left({W^{\{L-1\}}}^T x_n + b^{\{L-1\}}\right) + b_k^{\{L\}}\right] \\
&= \sigma\left[{W_{:,k}^{\{3\}}}^T \sigma\left({W^{\{2\}}}^T x_n + b^{\{2\}}\right) + b_k^{\{3\}}\right].
\end{aligned}$$

Remarks 2.6.

- *Actually, here, $\widehat{y} \in (0,1)^{N_L}$ as the sigmoid function only produces outputs between 0 and 1. These outputs can be interpreted as probabilities, i. e. ${}^{[n]}\widehat{y}_c$ is the probability of input sample x_n belonging to class c. To perform classification, one could then assign class c^* to x_n where $c^* = \arg\max_{c'} {}^{[n]}\widehat{y}_{c'}$. For other activation functions, the model output may not be restricted to $(0,1)^{N_O}$ and the network can be used e. g. for regression tasks.*

- *It is important that the chosen activation function is non-linear, as otherwise neural networks collapse to a single layer network, regardless of the number of hidden layers and units.*
  To prove this, choose the identity as activation function, i. e. $\sigma(x) = x$. Then, the model output simplifies to

$$ {}^{[n]}\widehat{y} = W^{\{L\}^T} \left(W^{\{L-1\}^T} \, {}^{[n]}z^{\{L-2\}} + b^{\{L-1\}} \right) + b_k^{\{L\}}. $$

  This gives a single layer network of the type $\widehat{y} = W^* x_n + b^*$, where

$$ \begin{aligned} W^* &= W^{\{L\}^T} W^{\{L-1\}^T} \dots W^{\{2\}^T}, \\ b^* &= b_k^{\{L\}} + W^{\{L\}^T} \left(b^{\{L-1\}} + W^{\{L-1\}^T} (\dots) \right). \end{aligned} $$

2.1.5 Optimization in a Neural Network

Starting from the error ${}^{[n]}E$ at the output layer $\{L\}$, first propagate the error back to the hidden layer. Meaning the quantity of interest is the derivative of ${}^{[n]}E$ with respect to the weights $W_{:,k}$ of the output layer. More specifically, we are going from each output unit k back to the hidden unit j of the hidden layer $\{L-1\}$ by

$$ \frac{\partial \, {}^{[n]}E}{\partial \, W_{jk}^{\{L\}}} = \frac{\partial \, {}^{[n]}E}{\partial \, {}^{[n]}\widehat{y}_k} \cdot \frac{\partial \, {}^{[n]}\widehat{y}_k}{\partial \, {}^{[n]}a_k^{\{L\}}} \cdot \frac{\partial \, {}^{[n]}a_k^{\{L\}}}{\partial \, W_{jk}^{\{L\}}}, \qquad j = 1, \dots, N_{L-1}, \; k = 1, \dots, N_L. \tag{2.1} $$

Simply by applying the chain rule, the non-trivial derivative of $\frac{\partial \, {}^{[n]}E}{\partial \, W_{jk}^{\{L\}}}$ breaks down into several much less complicated components. The right hand side of Equation (2.1) can be checked term by term, from right to left:

$$ \frac{\partial \, {}^{[n]}a_k^{\{L\}}}{\partial \, W_{jk}^{\{L\}}} = \frac{\partial \, \sum_{p=1}^{N_{L-1}} \left(W_{pk}^{\{L\}} \cdot {}^{[n]}z_p^{\{L-1\}} \right) + b_k^{\{L\}}}{\partial \, W_{jk}^{\{L\}}} = {}^{[n]}z_j^{\{L-1\}}, \tag{2.2} $$

$$ \frac{\partial \, {}^{[n]}\widehat{y}_k}{\partial \, {}^{[n]}a_k^{\{L\}}} = \sigma' \left({}^{[n]}a_k^{\{L\}} \right), \tag{2.3} $$

$$ \frac{\partial \, {}^{[n]}E}{\partial \, {}^{[n]}\widehat{y}_k} = - \left({}^{[n]}y_k - {}^{[n]}\widehat{y}_k \right). \tag{2.4} $$

Hence, Equation (2.1) gives

$$ \frac{\partial \, {}^{[n]}E}{\partial \, W_{jk}^{\{L\}}} = - \left({}^{[n]}y_k - {}^{[n]}\widehat{y}_k \right) \cdot \sigma' \left({}^{[n]}a_k^{\{L\}} \right) \cdot {}^{[n]}z_j^{\{L-1\}}. \tag{2.5} $$

A second item of interest is the bias at each layer. For the bias at the output layer, it follows similarly

$$\frac{\partial\ {}^{[n]}E}{\partial\ b_k^{\{L\}}} = \frac{\partial\ {}^{[n]}E}{\partial\ {}^{[n]}\widehat{y}_k} \cdot \frac{\partial\ {}^{[n]}\widehat{y}_k}{\partial\ {}^{[n]}a_k^{\{L\}}} \cdot \frac{\partial\ {}^{[n]}a_k^{\{L\}}}{\partial\ b_k^{\{L\}}} = -\left({}^{[n]}y_k - {}^{[n]}\widehat{y}_k\right) \cdot \sigma'\left({}^{[n]}a_k^{\{L\}}\right) \cdot 1,$$

since Equations (2.3)-(2.4) remain unchanged for the bias. Consequently, this means the two derivatives of interest can be expressed in terms of each other $\frac{\partial\ {}^{[n]}E}{\partial\ W_{jk}^{\{L\}}} = \frac{\partial\ {}^{[n]}E}{\partial\ b_k^{\{L\}}} \cdot {}^{[n]}z_j^{\{L-1\}}$.

Propagating the error back one more layer, i. e. to the hidden layer's weights and biases, works in the same fashion, i. e.

$$\begin{aligned}
\frac{\partial\ {}^{[n]}E}{\partial\ W_{jk}^{\{L-1\}}} &= \sum_{p=1}^{N_L} \frac{\partial\ {}^{[n]}E}{\partial\ {}^{[n]}\widehat{y}_p} \cdot \frac{\partial\ {}^{[n]}\widehat{y}_p}{\partial\ {}^{[n]}a_p^{\{L\}}} \cdot \frac{\partial\ {}^{[n]}a_p^{\{L\}}}{\partial\ {}^{[n]}z_k^{\{L-1\}}} \cdot \frac{\partial\ {}^{[n]}z_k^{\{L-1\}}}{\partial\ {}^{[n]}a_k^{\{L-1\}}} \cdot \frac{\partial\ {}^{[n]}a_k^{\{L-1\}}}{\partial\ W_{jk}^{\{L-1\}}} \\
&= -\sum_{p=1}^{N_L} \left[\left({}^{[n]}y_p - {}^{[n]}\widehat{y}_p\right) \cdot \sigma'\left({}^{[n]}a_p^{\{L\}}\right) \cdot W_{kp}^{\{L\}}\right] \cdot \sigma'\left({}^{[n]}a_k^{\{L-1\}}\right) \cdot x_j, \\
\frac{\partial\ {}^{[n]}E}{\partial\ b_k^{\{L-1\}}} &= \sum_{p=1}^{N_L} \frac{\partial\ {}^{[n]}E}{\partial\ {}^{[n]}\widehat{y}_p} \cdot \frac{\partial\ {}^{[n]}\widehat{y}_p}{\partial\ {}^{[n]}a_p^{\{L\}}} \cdot \frac{\partial\ {}^{[n]}a_p^{\{L\}}}{\partial\ {}^{[n]}z_k^{\{L-1\}}} \cdot \frac{\partial\ {}^{[n]}z_k^{\{L-1\}}}{\partial\ {}^{[n]}a_k^{\{L-1\}}} \cdot \frac{\partial\ {}^{[n]}a_k^{\{L-1\}}}{\partial\ b_k^{\{L-1\}}} \\
&= -\sum_{p=1}^{N_L} \left[\left({}^{[n]}y_p - {}^{[n]}\widehat{y}_p\right) \cdot \sigma'\left({}^{[n]}a_p^{\{L\}}\right) \cdot W_{kp}^{\{L\}}\right] \cdot \sigma'\left({}^{[n]}a_k^{\{L-1\}}\right) \cdot 1.
\end{aligned}$$

The additional sum here comes from the fact that the error with respect to the weight connecting input unit j to hidden unit k is affected by all the output units $p = 1, \ldots, N_L$. The above results in a simple recursive procedure that may be applied to NNs with an arbitrary amount of layers and neurons. It now also becomes clear why the computation of gradients with backpropagation scales linearly in the total number of network weights. First, compute a *forward pass*, i. e. pass the data from input through to the output, and, next, a *backward pass* during which the gradients are computed based on the values obtained during the previous step. Each neuron is visited twice.

Optimizing the Weights

Now that the gradients with respect to every weight in the network have been computed, we would like to find the optimal weights such that the final error at the output layer is minimized for given inputs. A natural choice for doing so is to use gradient descent. Higher-order optimization techniques, e. g. Newton's Method, generally converge faster since they exploit the curvature of the function but are computationally more expensive and, thus, usually not considered in the context of neural networks.

Definition 2.7. (Gradient Descent in Neural Networks)
Gradient descent (GD) is an iterative way to obtain (locally) optimal weights given their gradients. Let $L \in \{H, O\}$ be the current layer of the network. Let $\nabla W^{\{L\}}$ be the gradient

of the cost function J with respect to the weight matrix of layer L, $\nabla b^{\{L\}}$ the one of the bias of the layer.
Process the network for the ***entire*** *set of inputs and iteratively update weights and biases by*

$$
\begin{aligned}
{}_{t+1}W^{\{L\}} &= {}_{t}W^{\{L\}} - \alpha \nabla {}_{t}W^{\{L\}}, \\
{}_{t+1}b^{\{L\}} &= {}_{t}b^{\{L\}} - \alpha \nabla {}_{t}b^{\{L\}},
\end{aligned}
$$

until some stopping criterion is met[2] *and where $\alpha > 0$ is a suitable learning rate – typically close to zero. Repeatedly subtracting a fraction of the gradient from the current function value leads to a local minimum since the gradient always points in the steepest ascent direction. If the function is convex, this local minimum is a global minimum, however, this is typically not the case for neural networks.*

2.1.6 Deficiencies of Backpropagation

It is important to mention that, while backpropagation certainly is a very powerful method, it also has its deficiencies.

Local Minima

One of the major flaws is that functions represented by neural networks tend to be highly non-convex, i. e. we are only guaranteed to converge to a local minimum, not a global one.

Vanishing Gradients

If the given network learns very large weight values during the training phase, the input to the sigmoid activation function can possibly be very large as well (in absolute value). This means that the value returned by the sigmoid function will be close to either 0 or 1 and, therefore, the gradient will be almost 0. Hence, the respective weight updates will be close to 0, too. This would cause the training process to not achieve any further improvements. Due to its shape, the sigmoid activation function is particularly prone to running into this problem.

In 2000, Hahnloser et al. introduced the rectified linear unit (ReLU) activation function, [Hah+00], and it gained in popularity through the work of Nair and Hinton in 2010, [NH10], and also Glorot, Bordes and Bengio in 2011, [GBB11]. Using this activation function instead of the sigmoid, the network is much less prone to the problem of vanishing gradients as can be seen when comparing the shapes of these two activation functions (Figure 2.3 from before). However, there is the possibility that certain neurons "collapse", i. e. their activation becomes zero for the whole training process. Here, the leaky ReLU (Subsection 2.1.3) provides remedy.

[2] e. g. the changes in the weights or error measure at each step are almost zero.

2.1.7 Improved Network Optimization

Earlier, backpropagation in its most basic form was introduced. Many improvements and extensions to the presented procedure have been published in the past – a few of these will be briefly introduced before moving on to convolutions. Most of these enhancements are aiming particularly at improving the convergence of gradient descent (GD) with backpropagation. They can be split into two categories. Methodological adaptions of GD, on the one hand, and methods using an adaptive learning rate, on the other hand. Two of the former being:

- **Stochastic Gradient Descent (SGD)**
  Instead of evaluating the gradient based on the entire training data set – like in the regular GD – SGD only evaluates the gradient on a single training example at a time. This not only requires considerably less memory but it is also much less likely to get stuck in a certain local minimum while a better one is nearby. See e. g. [KLY18] for more details.

- **Mini-Batch GD**
  This is a mixture of GD and SGD, where the mini-batch size (i. e. number of training samples processed for one update) represents a trade-off between the computational complexity of one parameter update and the accuracy of such an update. This is typically the standard choice for training a neural network with large amounts of input data.

Using a constant learning rate α is usually not optimal. The following three are only a selected few out of many methods which can be used to gradually optimize the learning behavior over time:

- **Momentum**
  The optimization is based on the "history" of previous gradients. As long as the gradients point into the same direction, the method builds up a momentum and goes with bigger steps in that particular direction. Contrary to the other methods listed here, the momentum algorithm does not actually change the learning rate; it only adds the accumulated momentum to the gradient descent update step.

- **Adadelta**
  The learning rate is based on an exponentially decaying average of squared gradients.

- **Adaptive Moment estimation (Adam)**
  A powerful combination of Momentum and Adadelta.

The above methods (amongst others) are presented in detail in, e. g., [Rud16] and will not be further analyzed here. Some of the above adaptions aim to improve the convergence speed, others are potentially less likely to get stuck on plateaus or in bad local minima (i. e. a much better minimum would be close by).

Combining for instance mini-batch GD with Adam will generally lead to much better training behavior than using just regular mini-batch GD with a constant learning rate. Figure 2.4 shows the result of an optional Python project task of a lecture at TUM.[3] Three basic neural networks were trained for image classification on the CIFAR-10 dataset.[4] Each model is based on a fully-connected neural network with 5 hidden layers, 100 neurons each, a mini-batch size of 100 and an initial learning rate of 0.01 for the blue and orange curves, 0.001 for the green one. Despite the smaller learning rate, Adam converges faster and is more accurate (thus, training is stopped earlier).

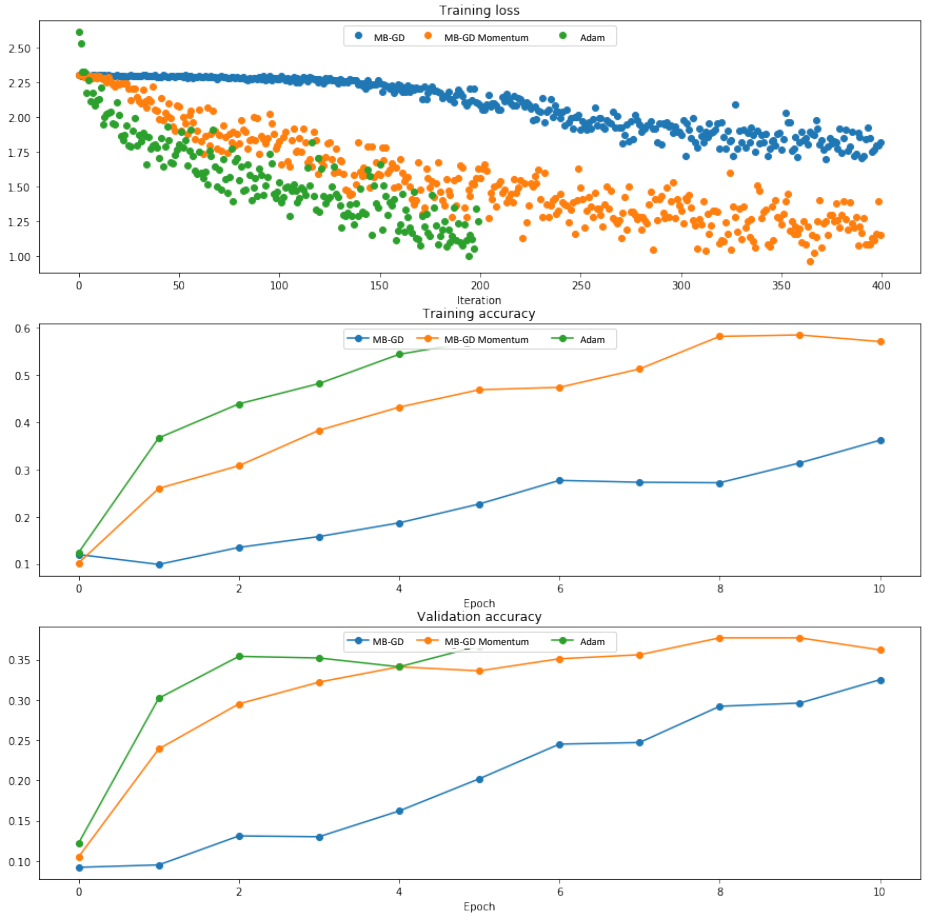

Figure 2.4 Mini-batch GD (MB-GD) with and without adaptive learning rates.

2.1.8 Regularization

To prevent a NN from *overfitting* the training data, i. e. learning to represent the training data perfectly but failing to work on unseen test data, there exist several techniques.

[3] IN2346: Introduction to Deep Learning, summer term 2018.
[4] Dataset with 10 classes; 6,000 images each. Data collected by Krizhevsky, Hinton et al., [KH+09].

Common ones are, for instance, adding an L1 or L2 penalty term to the network's cost function, just like for e. g. Ridge and Lasso regression models,[5] or using dropout. Dropout, see [Hin+12], is a regularization technique specific to deep neural networks, where – with a certain probability – neurons are randomly dropped out, i. e. set to zero, during the training phase. This has proven to lead to better generalization by keeping the network from relying heavily on certain input combinations.

Another popular regularization method is data augmentation. Say we are trying to classify whether or not there is a cat on a picture. Then our model should correctly recognize the cat regardless of its "position", i. e. we would like the classifier to be invariant to various transformations, e. g. to recognize a cat even if it is up-side-down. In this regard, many datasets could be biased as they might only contain pictures of animals where the animals are in the main focus. To nevertheless achieve accurate results, we augment the dataset by transformations of itself. For example, we can add the 180° rotation of each image to the dataset. Other augmentations are, e. g., randomly cropping images into smaller pieces (e. g. only a cat's paws remain on a certain image) or random brightness or contrast changes. In most cases, this will make a classifier more robust and increase generalization.

2.1.9 Batch Normalization

Batch normalization, introduced by [IS15], is a powerful extension to neural networks offering faster, more robust training, increased model performance and better generalization. Simply put, it is a normalization layer in a mini-batch based neural network (Subsection 2.1.7). A layer's activation a will be mean centered and standardized for each mini-batch individually. Algorithm 1 describes the procedure in more detail.

Algorithm 1: Batch Normalization applied to activation a at layer l over a mini-batch, taken from [IS15].

input : Values over mini-batch $B = \{a_1, \dots, a_m\}$; Parameters to be learned: γ, β

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} a_i \qquad \texttt{// mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu_B)^2 \qquad \texttt{// mini-batch variance}$$

$$\widehat{a}_i \leftarrow \frac{a_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \texttt{// normalize}$$

$$y_i \leftarrow \gamma \widehat{a}_i + \beta \equiv BN_{\gamma,\beta}(a_i) \qquad \texttt{// scale and shift}$$

The above transformations are differentiable and, thus, the necessary assumptions for backpropagation are not violated; there is just an additional chain rule step.

[5] For Ridge regression see e. g. [TA77], for Lasso regression [Tib96].

2.2 Convolutional Neural Networks

Fully-connected neural networks deliver great results on numerical classification tasks, but become extremely memory hungry when applied to image classification. Imagine the objective is to classify a gray-scale input image of 1,000 by 1,000 pixels with a simple NN with one hidden layer and 1,000 units. This would lead to an immense number of required weights, i. e. one weight per each pixel and unit combination, 10^9 parameters in total. Also, this approach does not capture spatial pixel correlations as every pixel is processed on its own and only combined pairwise with all other pixels. This is where convolutions come into play. Instead of applying weight matrices to the input, filters (also called kernels) are used to extract only the key features. Similarly to NNs, we can use gradient descent based techniques to find the optimal kernels which extract certain kinds of features. Finally, a fully-connected feed-forward NN could be used to perform the classification task on the reduced space of these key features. [Kha+19] gives an overview of major milestone achievements (Figure 2.5).

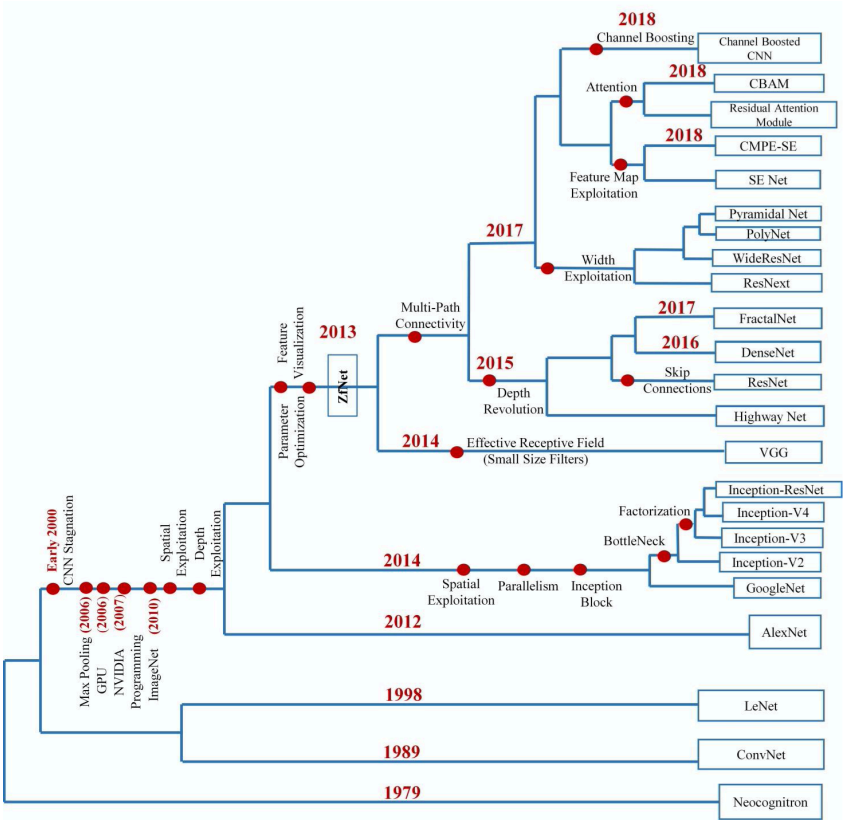

Figure 2.5 Evolutionary history of CNNs from ConvNet till to date. Taken from [Kha+19].

Coming back to the gray-scale image example, using a one-layer convolutional NN (CNN) with 64 3x3 kernels leads to just slightly above 500 parameters. Not only were convolutional

nets computationally much less expensive, but they also delivered better results on image data than regular NNs. Thus, they quickly grew popular and there have been many improvements. The subsequent architectural concepts are following [Zha+20].

2.2.1 Kernels

A convolutional neural network (CNN) is an extension of a NN where the concept of weight matrices is replaced by that of kernels. The parameters are shared over the entire network input and multiple such kernels will be applied simultaneously.

Take a single kernel of, for instance, size 3x3. Lay this kernel over the input image and slide it from left to right, top to bottom, through the input image, multiplying each input pixel by the respective kernel element. Each such operation results in a (smaller) matrix whose entries are then summed up, giving one element of the output matrix, i. e. for each step we perform a matrix product with the kernel's transpose. See e. g. Figure 2.6 for an illustration of the first of these steps of this process on a 4x4 input. This gives the first element of the final 2x2 output, $\left(\begin{smallmatrix} 30 & 1 \\ 19 & 26 \end{smallmatrix}\right)$, which is obtained by completing the remaining three steps on this input "image" by further sliding the filter.

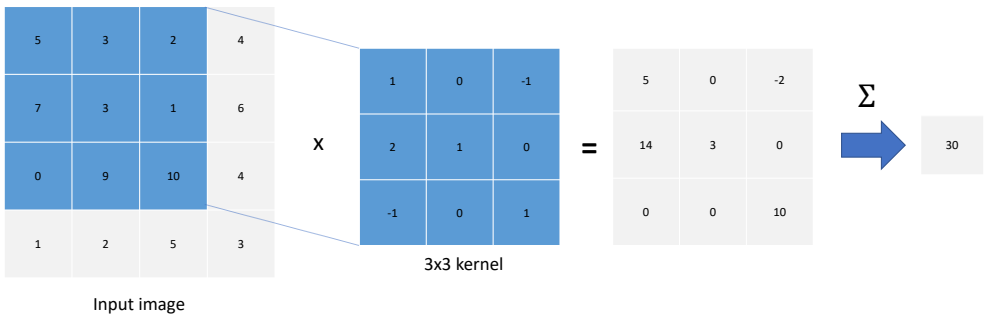

Figure 2.6 First step of applying a 3x3 kernel on a 4x4 image.

This can be done for multiple kernels simultaneously, i. e. adding a third dimension to the kernel, which yields different feature capturing filters due to randomness in the initialization phase. Applying the full set of kernels to the input forms what is called a convolutional layer. For simplicity, we will work with one kernel only going forward.

2.2.2 Strides

The process described above is using a stride of one. The stride quantifies how far the filter is moved after each operation, i. e. a stride of one means we just slide the kernel by one to the right, or downward, respectively. With a stride of two, it is moved by two, a stride of three means to skip over three rows/columns, and so on. The output dimensions then look as follows:

$$\text{Input: } N \times N, \qquad \text{Kernel: } K \times K, \qquad \text{Stride: } S$$
$$\Rightarrow \text{Output: } \left(\frac{N-K}{S}+1\right) \times \left(\frac{N-K}{S}+1\right).$$

This can obviously lead to ill-posed set-ups which is why K and S must be chosen such that $\frac{N-K}{S}$ is an integer.

2.2.3 Padding

Having several convolutional layers after another, the dimensions will shrink extremely fast and corner pixels are only ever used once. To counter these effects, we add a padding to the layer's input, increasing its size. See the figure below which extends the example from Figure 2.6 by adding a 0-padding – using a stride of one, again. One could also decide to add more than just one row/column of zeros. This amount is called the padding size. There exist approaches of padding the image with values other than zeros, e. g. reflections of the actual image, or duplicating the outer image values.

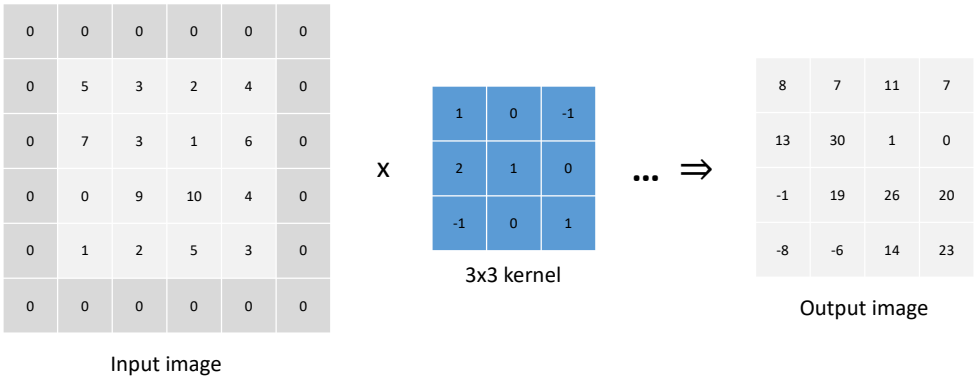

Figure 2.7 Applying a 3x3 kernel on a 0-padded 4x4 image.

An updated view on the output dimensions is then:

$$\text{Input: } N \times N, \qquad \text{Kernel: } K \times K, \qquad \text{Stride: } S, \qquad \text{Padding: } P$$
$$\Rightarrow \text{Output: } \left(\frac{N+2P-K}{S}+1\right) \times \left(\frac{N+2P-K}{S}+1\right), \qquad \frac{N+2P-K}{S} \in \mathbb{N}.$$

2.2.4 Pooling

As an alternative to regular convolutional layers with kernels, we can use what is called pooling layers for downsampling the input in a CNN. A pooling layer can be seen as a special convolutional layer with a fixed kernel which is capturing the "strongest" features

in a certain region of the picture. Typical such pooling layers are max pooling and average pooling, i. e. take the maximum – or average, respectively – of each input slice. See, for instance, [SMB10]. Adapting the previous example from Figure 2.6 by replacing the kernel with an average pooling approach, leads to a different result, see the figure hereafter.

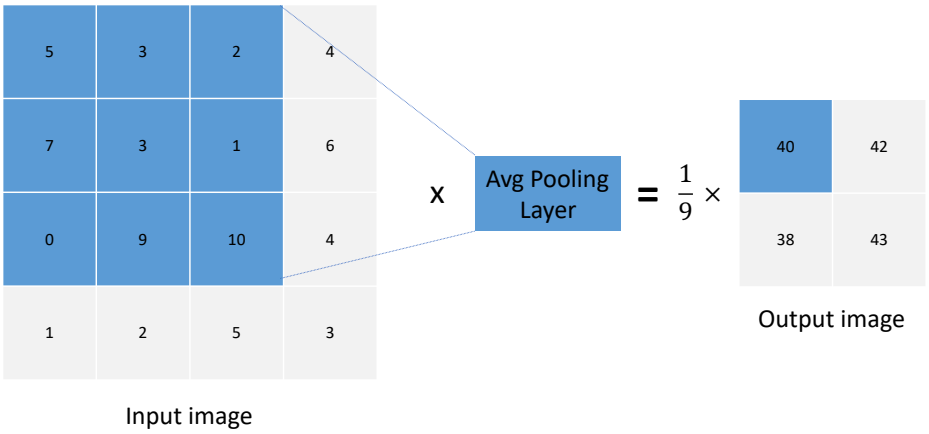

Figure 2.8 Applying an Average Pooling layer on a 4x4 image.

2.3 Autoencoders

It is not clear when or by whom an autoencoder (AE) as such has been first introduced, but it probably goes back to the 1980s where Rumelhart et al. were trying to solve an error propagation problem ([RHW85]). More recently, along with the boom in deep learning research, autoencoders (AEs) have become an increasingly popular topic. They are neural networks that perform an unsupervised task to output their own original input by compressing it into a (much) lower dimensional space and, from there, reconstructing the original input with minimal information loss. Commonly referred to as encoder and decoder steps. Due to the encoding into a lower dimensional space, only the most relevant features will be kept – whatever these may be. Typically, encoder and decoder operate in a symmetrical fashion, i. e. the encoder and decoder have the same amount of layers. Obviously, the use cases of such an architecture are highly limited, though, there exist extensions that have proven to be quite powerful and will be discussed in the following sections. One would usually use AEs for image denoising or outlier detection. [Pla18] outlines the relation between AEs and principal component analysis, where a certain autoencoder set-up allows to train weights spanning the same subspace as the one spanned by the principal component loading vectors. Typically, an AE's reconstructed output is noisy and appears blurry when working with image data.

2.3.1 Convolutional Autoencoders

Autoencoding can easily be transferred to CNNs and the corresponding architecture is commonly called a convolutional autoencoder. Convolutional layers are used to reduce the input image to a lower dimensional space. Kernels and upsampling (factor 2) layers are then used to scale up again to the original space. Many typical implementation examples of such convolutional AEs are based on the MNIST database.[6] It contains 60,000 training images and 10,000 testing images of 28×28 pixels, black and white. Each image represents one handwritten digit ranging from 0 to 9. Figure 2.9 illustrates how reconstructed outputs of a straight-forward convolutional AE implementation could look like.

Figure 2.9 Input and reconstructed output of an Autoencoder.[7]

Going forward, when speaking of an Autoencoder we are referring to a convolutional one.

2.4 Variational Autoencoders

In 2013, Kingma and Welling introduced the variational autoencoder (VAE) ([KW13]). Their method suggests adjustments to the bottleneck of regular AEs, i. e. the point where the space is most reduced. Instead of simply scaling back up from there, they show how to use variational inference to learn parameters describing a probability distribution and, thereby, represent the input data in this latent space. It is then possible to sample from this distribution and scale back up, leading to much higher robustness and making the variational autoencoder (VAE) a generative model. The latter is a desirable characteristic as it allows to generate new data simply by plugging a random sample into the decoder network. Figure 2.10 shows a simplified representation of a VAE architecture.

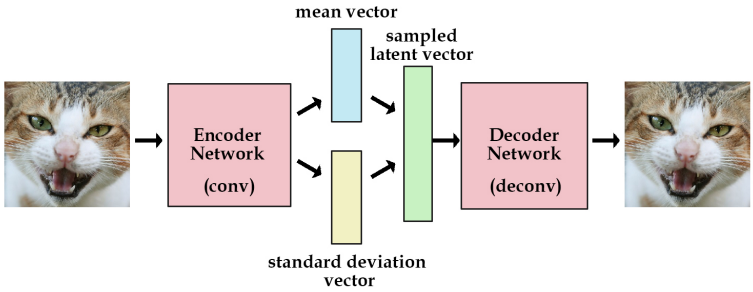

Figure 2.10 Components of a (convolutional) VAE.[8]

[6] MNIST database: Modified National Institute of Standards and Technology (NIST) database – a size-normalized and centered subset of a dataset from NIST – was created by LeCun et al., [LCB10].
[7] from a Keras Autoencoder tutorial: `https://blog.keras.io/building-autoencoders-in-keras.html`

2.4.1 Variational Inference for Autoencoders

This section's content largely follows the paper of Kingma and Welling, [KW13]. Assume some data X is generated by some latent variable z. The objective is to infer knowledge about z by computing

$$p(z|X) = \frac{p(X|z)p(z)}{p(X)}, \qquad p(X) = \int p(X|z)p(z)dz.$$

Unfortunately, this problem is intractable. We can, however, try approximating $p(z|X)$ by another distribution $q(z|X)$. Here, Kingma and Welling propose to make use of the Kullback-Leibler divergence which was originally defined in [KL51].

Definition 2.8. (Kullback-Leibler Divergence)
The Kullback-Leibler (KL) divergence, or relative entropy, measures how different one probability distribution is from another. Let p and q be probability distributions defined on the same probability space $\mathcal{X}$. We define their Kullback-Leiber divergence as:

$$\mathbb{KL}(q \mid\mid p) \coloneqq \int_{\mathcal{X}} p(z) \log \frac{p(z)}{q(z)} dz = E_{z\sim p(z)} \left[\log p(z) - \log q(z)\right]$$

The KL divergence is in general asymmetric, i. e. $\mathbb{KL}(p \mid\mid q) \neq \mathbb{KL}(q \mid\mid p)$, and non-negative.

Now, we can reformulate our objective as finding the distribution q which is the most similar to p:

$$q^*(z) = \arg\min_{q\in\mathcal{Q}} \mathbb{KL}(q(z) \mid\mid p(z|X)) = \arg\min_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q} \left[\log q(z) - \log p(z|X)\right], \tag{2.6}$$

where $\mathcal{Q}$ is a set of tractable candidate distributions. Equation (2.6) still involves the intractable $p(z|X)$, so we can maximize the (variational) lower bound instead:

$$q^*(z) = \arg\max_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q} \left[\log p(X, z) - \log q(z)\right], \tag{2.7}$$

since

$$\begin{aligned}
&\arg\min_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q} \left[\log q(z) - \log p(z|X)\right] \\
&= \arg\min_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q} \left[\log q(z) - \log p(X, z) + \underbrace{\log p(X)}_{\text{independent of } z}\right] \\
&= \arg\min_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q} \left[\log q(z) - \log p(X, z)\right] + \text{const.}
\end{aligned}$$

[8] taken from `http://kvfrans.com/variational-autoencoders-explained/`

Once more, it is possible to rewrite the objective in Equation (2.7):

$$\begin{aligned}
&\mathrm{E}_{z\sim q}\left[\log p(X,z) - \log q(z)\right] \\
=&\mathrm{E}_{z\sim q}\left[\log p(X|z)p(z) - \log q(z)\right] \\
=&\mathrm{E}_{z\sim q}\left[\log p(X|z) + \log p(z) - \log q(z)\right] \\
=&\mathrm{E}_{z\sim q}\left[\log p(X|z)\right] - \mathrm{E}_{z\sim q}\left[\log p(z) - \log q(z)\right] \\
=&\mathrm{E}_{z\sim q}\left[\log p(X|z)\right] - \mathbb{KL}(q(z) \mid\mid p(z)).
\end{aligned}$$

Part of the objective, $\mathbb{KL}(q(z) \mid\mid p(z))$, can now be calculated in closed-form for specific $\mathcal{Q}$. Putting everything together, the objective is as follows:

$$q^*(z) = \arg\max_{q\in\mathcal{Q}} \mathrm{E}_{z\sim q}\left[\log p(X|z)\right] - \mathbb{KL}(q(z) \mid\mid p(z)). \tag{2.8}$$

Before we proceed, we introduce the following notations:

- Determinant of a square matrix A: $\det(A)$,
- Trace of a matrix A: $\mathrm{tr}(A)$.

Now, let p follow a Gaussian distribution, $p(z|X) \sim \mathcal{N}(\mu, \sigma^2)$ with mean μ and variance σ^2. Accordingly, we choose $q(z|X) = \mathcal{N}\left(z \mid \mu_\theta(X), \Sigma_\theta(X)\right)$, where μ_θ and Σ_θ are arbitrary functions dependent on parameters θ which can be inferred from the data. The KL divergence of two Gaussian distributions can be calculated analytically:[9]

$$\begin{aligned}
&\mathbb{KL}(\mathcal{N}(\mu_1, \Sigma_1) \mid\mid \mathcal{N}(\mu_2, \Sigma_2)) = \\
&\qquad = \frac{1}{2}\left(\mathrm{tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1) - d + \log\frac{\det\Sigma_2}{\det\Sigma_1}\right),
\end{aligned}$$

where d is the distributions' dimensionality. However, $\mathrm{E}_{z\sim q}\left[\log p(X|z)\right]$ in Equation (2.8) is harder to solve.

The "Reparameterization Trick"

Ideally, we would still want to be able to optimize the network using gradient descent. Kingma and Welling found that, for sufficiently large batch sizes, one sample from p is sufficient to approximate $\mathrm{E}_{z\sim q}\left[\log p(X|z)\right]$, but one may also combine this with a Monte Carlo sampling approach. As we optimize our network through backpropagation, all operations need to be differentiable. However, sampling from a distribution does not meet this criteria. Therefore, Kingma and Welling propose a "reparameterization trick". Instead of sampling from $p(z|X)$, we can sample from a standard normal distribution and let the network learn the respective transformation. This workaround maintains the necessary differentiability of the network. See Figure 2.11 for an illustration; red colors indicate non-differentiable sampling operations and blue shows loss layers.

[9] Refer to, e. g. [Duc07], for a simple proof.

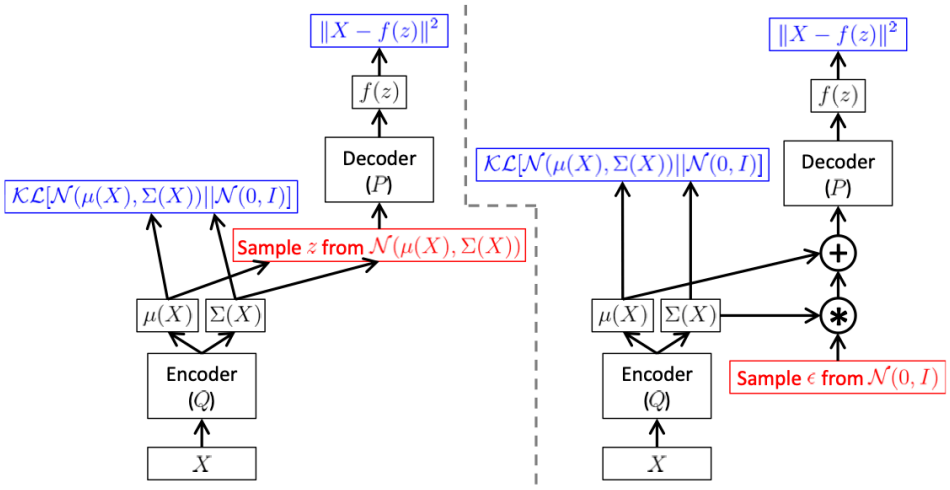

Figure 2.11 Variational autoencoder – reparameterization trick. The feedforward behavior of these networks is identical, but backpropagation can be applied only to the right network. Taken from [Doe16].

2.4.2 Deep Feature Consistency

Since its introduction in 2013, the variational autoencoder has seen many improvements, especially convolutional VAEs. One very promising of these is the combination of VAE with a pre-trained deep CNN via a perceptual loss propagation. The perceptual loss of two images is defined as their similarity of hidden features in such a pre-trained deep CNN. Hou et al. ([Hou+16]) propose how such a framework could look like – a deep feature consistent variational autoencoder (DFC VAE). They suggest to combine a VAE with a state-of-the-art pre-trained deep CNN, the VGGNet-19 (Figure 2.12).[10]

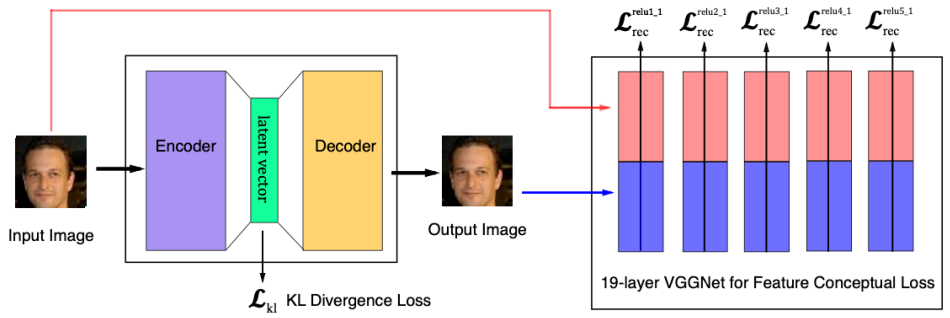

Figure 2.12 Deep feature consistent VAE. Taken from [Hou+16].

This means the VAE will be penalized by the dissimilarity of its own output compared to the original image, when passing both through the pre-trained VGGNet-19 CNN. This is expected to force the VAE to achieve a better reconstruction, ensuring a deep feature

[10] Winner of the ImageNet Challenge 2014 (Classification+Localization). For model details see [SZ15].

consistency. Figure 2.13 shows how this DFC VAE compares to a plain VAE. It shows the model outputs, i. e. image reconstructions, based on the CelebA dataset.[11] The first row shown in the figure is the input image, the second row is generated from the decoder network of a plain variational autoencoder (PVAE), and the last two rows are the results of VAE-123 and VAE-345 trained with feature perceptual loss based on VGGNet-19 layers 1-2-3 and 3-4-5, respectively. On the one hand, we can see that the PVAE seems to produce a reasonably good reconstruction, however, the images appear blurry and are lacking detail. On the other hand, the models trained with perceptual loss have a much sharper output image appearance. The figure also shows that there is quite a significant difference between VAE-123 and VAE-345. Some reconstructed images of the latter show e. g. different hair colors compared to the input but, at the same time, seem to better capture certain features like, for instance, hair strands or head coverings.

Figure 2.13 Image reconstructions from different models. Taken from [Hou+16].

2.5 Variational Autoencoder - Generative Adversarial Networks

This section will present a framework that is a powerful combination of two architectures. Both of which have proven successful over the recent years. First, we need to briefly present generative adversarial networks (GANs).

2.5.1 Generative Adversarial Networks

Ian Goodfellow introduced the Generative Adversarial Network (GAN) in 2014, [Goo+14]. This framework consists of two competing NNs: a generator and a discriminator. The

[11] containing more than 200,000 celebrity face images, see [Liu+15].

former takes random noise as an input and generates an output in the desired space; the latter tries to distinguish the generated output from input training data. The generator should learn to generate realistic outputs such that the discriminator is fooled and cannot tell true from fake anymore. For a simplified illustration, see Figure 2.14 below.

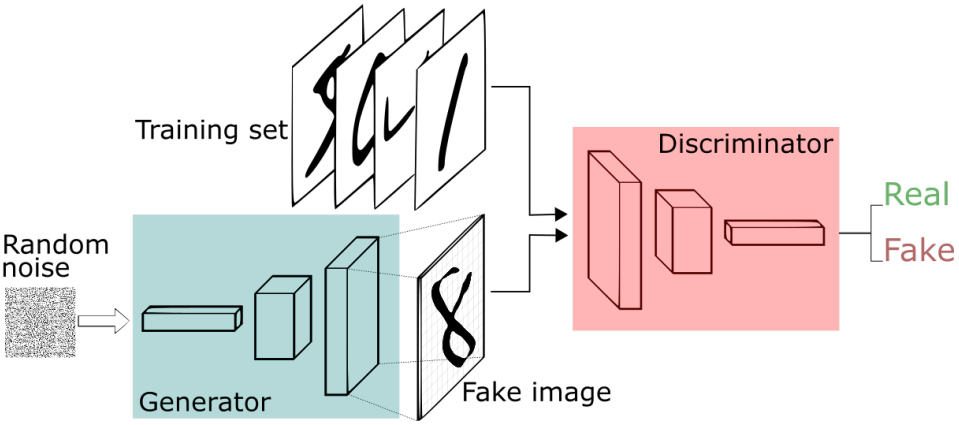

Figure 2.14 Framework of a generative adversarial network. Taken from [Bla18].

The training phase of this model is particularly complex, as one needs to maintain a certain balance between the two models. Training the generator much faster than the discriminator will make the latter unable to tell true from fake apart, even though the reconstruction is still poor and could be much better. Vice versa, if the discriminator is trained too fast, it will always be one step ahead of the generator, i. e. it will always recognize the fake picture. Simply choosing the same set of certain hyperparameters, e. g. learning rate and optimization method, is not sufficient to tackle this problem, as the two networks might behave entirely different. However, heuristics indicate that his can be overcome by training the two models in an alternating fashion, moving back and forth rather frequently.

GANs have become extremely popular for a variety of reasons. One of them being that they were the first generative model producing reasonably good outputs. Yann LeCun, known for laying out important groundwork on convolutional nets and being Facebook's Chief AI Scientist, called GANs in a talk in November 2016 at Carnegie Mellon University in Pittsburgh, Pennsylvania, the "coolest idea in deep learning in the last 20 years".[12]

2.5.2 Combining GAN & VAE

Only several months after Goodfellow's paper on GANs, Larsen et al. proposed a combination of GANs and VAEs ([Lar+15]) – building on top everything presented in this paper so far. They suggest to replace the generator in Goodfellow's GAN by a VAE, resulting in a compelling framework, variational autoencoder - generative adversarial network, for

[12] RI Seminar: Yann LeCun: The Next Frontier in AI: Unsupervised Learning. See LeCun's YouTube channel: `https://www.youtube.com/playlist?list=PL80I41oVxglK--is17UhoHVosOLFEJzKQ`

short VAE-GAN. Figure 2 from their paper illustrates how the two architectures interact, plus at which point certain training objectives are obtained (represented by gray lines). See Figure 2.15 hereafter.

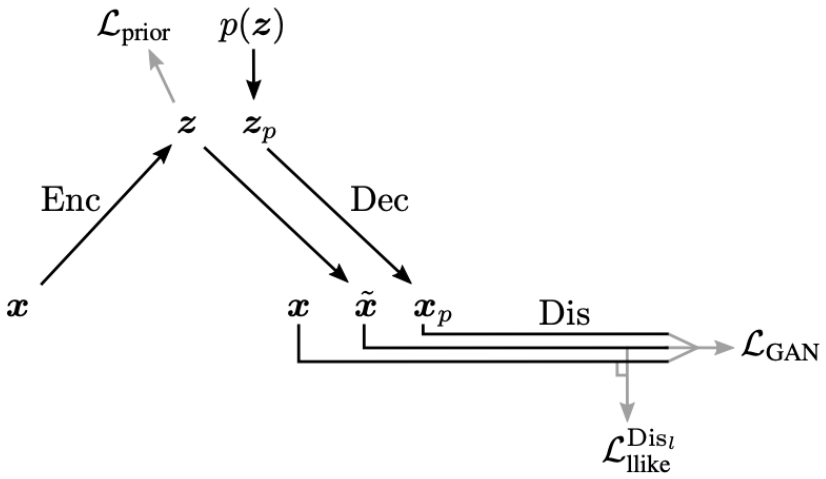

Figure 2.15 Flow through the combined VAE-GAN model during training. Taken from [Lar+15].

As the above figure suggests, training a model following such an architecture is complex. The discriminator model takes three inputs: the original image x, the VAE output $\tilde{x}$ and the decoder output x_p obtained by plugging in a random sample. $\mathcal{L}_{\text{llike}}^{\text{Dis}_l}$ represents a reconstruction error in the GAN discriminator at layer l. The straight-forward approach would be to optimize the following quantity

$$\mathcal{L} = \mathcal{L}_{\text{VAE}} + \mathcal{L}_{\text{GAN}}, \qquad \text{where } \mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{llike}}^{\text{pixel}} + \mathcal{L}_{\text{prior}}, \tag{2.9}$$

where $\mathcal{L}_{\text{llike}}^{\text{pixel}}$ is the element-wise pixel reconstruction error of the VAE[13] and $\mathcal{L}_{\text{prior}}$ is the KL divergence term. Instead, the authors suggest to replace $\mathcal{L}_{\text{llike}}^{\text{pixel}}$ in Equation (2.9) by $\mathcal{L}_{\text{llike}}^{\text{Dis}_l}$ which is the expected log-likelihood of a Gaussian observation model for the discriminator's hidden representation of layer l. The resulting training objective is then

$$\mathcal{L} = \mathcal{L}_{\text{prior}} + \mathcal{L}_{\text{llike}}^{\text{Dis}_l} + \mathcal{L}_{\text{GAN}}, \tag{2.10}$$

and the model can be trained following Algorithm 2:

[13] see the first quantity of the previously derived VAE objective in Equation (2.7).

Algorithm 2: Training the VAE-GAN model, taken from [Lar+15].

Initialize network parameters, $\theta_{Enc}, \theta_{Dec}, \theta_{Dis}$;

repeat

$X \leftarrow$ random mini-batch from dataset ;

$Z \leftarrow \text{Enc}(X)$;

$\mathcal{L}_{\text{prior}} \leftarrow \mathbb{KL}(q(Z|X) \mid\mid p(Z))$;

$\tilde{X} \leftarrow \text{Dec}(Z)$;

$\mathcal{L}_{\text{llike}}^{\text{Dis}_l} \leftarrow -\text{E}_{q(Z|X)}[p(\text{Dis}_l(X)|Z)]$;

$Z_p \leftarrow$ samples from prior $\mathcal{N}(0, I)$;

$X_p \leftarrow \text{Dec}(Z_p)$;

$\mathcal{L}_{\text{GAN}} \leftarrow \log(\text{Dis}(X)) + \log\left(1 - \text{Dis}(\tilde{X})\right) + \log(1 - \text{Dis}(X_p))$;

`// Update network parameters; gradient descent step`

$\theta_{\text{Enc}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Enc}}} \left(\mathcal{L}_{\text{prior}} + \mathcal{L}_{\text{llike}}^{\text{Dis}_l}\right)$;

$\theta_{\text{Dec}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Dec}}} \left(\gamma \mathcal{L}_{\text{llike}}^{\text{Dis}_l} - \mathcal{L}_{\text{GAN}}\right)$;

$\theta_{\text{Dis}} \overset{+}{\leftarrow} -\nabla_{\theta_{\text{Dis}}} \mathcal{L}_{\text{GAN}}$;

until *some stopping criterion*;

Here, Larsen et al. define a weight γ for the ability to reconstruct vs. fooling the discriminator. According to them, this can also be interpreted as weighting style and content. Finally, Figure 2.16 shows how VAE-GAN performs on the CelebA dataset. The authors describe the plain VAE's output as being able to draw the frontal part of the face sharply, but being blurry off-center. Also, their $\text{VAE}_{\text{Dis}_l}$ appears to produce sharper images even off-center. The VAE-GAN's reconstructed images generally look sharper with a more natural appearance.

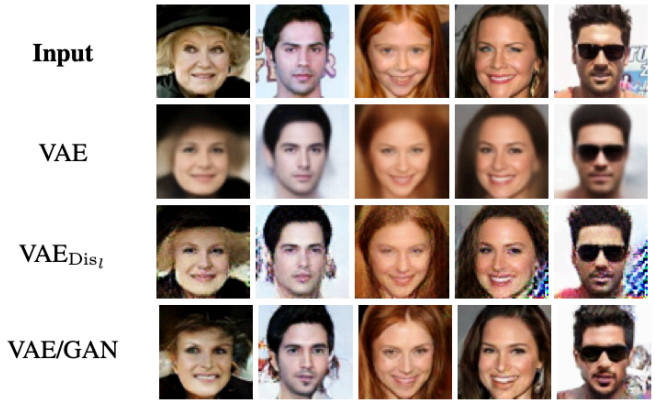

Figure 2.16 Reconstructions from different autoencoders. Taken from [Lar+15].

There has been a lot of research on VAEs, GANs and VAE-GANs since and results have been continuously pushed. In January 2019, Google's DeepMind has released the VQ-VAE-2, a second version of their Vector Quantized Variational AutoEncoder, claiming

to rival the (at the time) state-of-the-art BigGAN-deep ([ROV19]). Figure 2.17 shows generated images from VQ-VAE-2 by plugging a certain random sample into the decoder network. Thus, even though the pictures below may look very real, these persons do not exist. The authors are highlighting that the samples from the VQ-VAE-2 are especially realistic since the model successfully produces long-range dependencies. For example, matching eye colors and symmetrical attributes. At the same time, their model also covers lower density modes of the dataset like green hair color.

Figure 2.17 Representative samples from VQ-VAE-2. Taken from [ROV19].

2.6 The ExaHyPE

This section aims to briefly introduce the ExaHyPE (Exascale Hyperbolic PDE Engine), see [Rei+20]. It is an open source engine for solving first-order hyperbolic partial differential equations (PDEs) and it is built upon the PDE solver framework Peano ([Wei19]). The numerical method is given in the engine's concept and, in general, the user does not need to interact with any solver components. Our ultimate goal is to leverage ExaHyPE to simulate tsunamis based on bathymetry data. Following [Rei+20], we present only a few selected key items relevant for our specific application.

The engine needs PDEs of the following form:

$$\frac{\partial}{\partial t}Q + \nabla F(Q, \nabla Q) + B(Q) \cdot \nabla Q, \tag{2.11}$$

where $Q : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}^v$ is the state vector of the v conserved variables, Ω is the computational domain, $F(Q)$ is the flux tensor for modeling viscous effects and $B(Q)$ represents the non-conservative part. Hyperbolic systems of this form can be used for a wide range of applications involving waves. In particular, we are interested in the two dimensional shallow water equations which can model fluid flow in coastal areas. These

can be written in the form of Equation (2.11) as

$$\frac{\partial}{\partial t}\underbrace{\begin{pmatrix} h \\ hu \\ hv \\ b \end{pmatrix}}_{=Q} + \nabla \underbrace{\begin{pmatrix} hu & hv \\ hu^2 & huv \\ huv & hv^2 \\ 0 & 0 \end{pmatrix}}_{=F(Q)} + \underbrace{\begin{pmatrix} 0 \\ hg\ \partial_x(b+h) \\ hg\ \partial_y(b+h) \\ 0 \end{pmatrix}}_{=B(Q)\cdot\nabla Q} = 0,$$

where h is the water column's height, (u, v) the horizontal flow velocity, g the gravity and b the bathymetry. We will use an ADER-DG ([Zan+15]) based scheme provided in the ExaHyPE framework for solving these equations.

3 Autoencoding Bathymetry Data

Over the course of this chapter, we will explore how the previously described models perform on the GEBCO dataset. That is, we want to know how well they can approximate the original bathymetry data and capture certain kinds of features. We analyze each model's results both quantitatively and qualitatively.

3.1 Model Performances on Bathymetry Data

The code used to derive this section's results can be found on GitHub.[1] Implementations were done using Keras, a Python-based deep learning library ([Cho+15]).

General Setup

Coming back to the GEBCO dataset described in Chapter 1, we would now like to use the models from Chapter 2 to reconstruct the original data as well as possible. We will focus on quantitative results and, also, on a purely visual aspect.

To feed the data into our autoencoders, we need to break down the GEBCO dataset into many smaller pieces, which we will call samples going forward. From a modeling perspective, each sample is treated as a simple gray-scale image, i. e. each coordinate is represented by a pixel and its altitude is given through the respective pixel's value. The samples are set to a size of 96×96 pixels. Given the GEBCO dataset's dimensions of 86,400 $\times$ 43,200, this results in having a total of 405,000 samples. We split these into training (95%) and test (5%) set, where an additional 5% of the training set is reserved as a validation set for fine-tuning the models. Furthermore, the GEBCO's altitude measurements are scaled down into $[0, 1]$ to ensure higher model robustness and input/output consistency. Additionally, we augment both training and validation set by each sample's 90° rotation to the right.[2] Thus, we end up with a split of 731,025 training samples, 38,475 validation samples and 20,250 test samples – each of size 96×96 pixels.

3.1.1 Autoencoder

The very first goal is to assess the feasibility of obtaining reasonable model outputs. Therefore, we first implement a plain autoencoder, despite it not being a generative model. The AE's architecture is shown below in Figure 3.1.

[1] `https://github.com/se-wltr/topography_dl`

[2] Further augmentations were not applied as this significantly increases memory requirements.

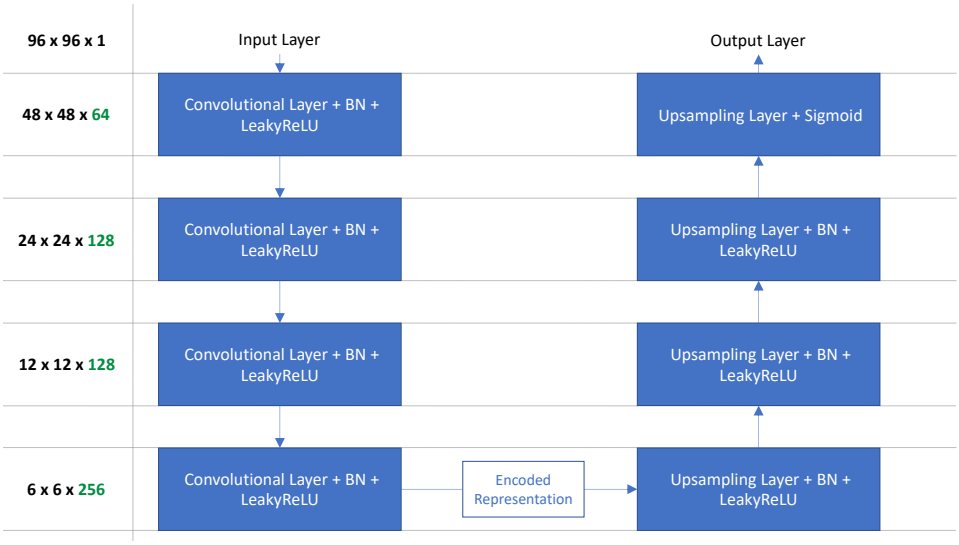

Figure 3.1 Our 8-layer autoencoder implementation architecture. The number of filters used at each layer is indicated in green.

Training this model for 50 epochs proved to be sufficient, as the loss started to stagnate from there on. We chose the Charbonnier error penalty[3] (Definition 2.5) and a rather high learning rate of 1e-2 with the Adam optimization method and a batch size of 128. As the Charbonnier loss closely resembles the L1 norm, it is known for high robustness. Figure 3.2 shows our model's loss development over time.

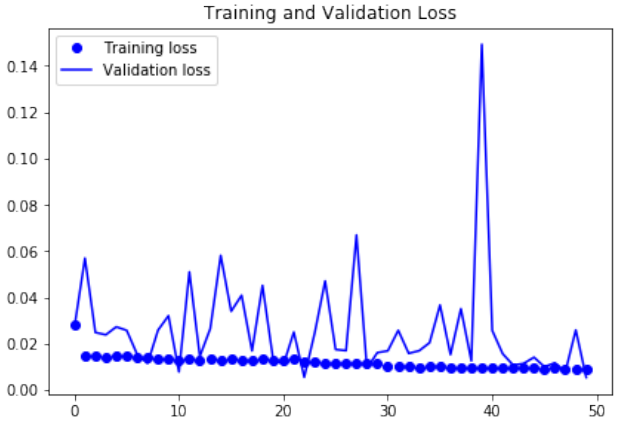

Figure 3.2 Autoencoder loss history.

Looking at how the model performs on reconstructing images from the unseen test set, Figure 3.3 shows a good visual approximation. The reconstructions appear to be not as good at capturing maximum and minimum values, i. e. the very yellow or dark spots,

[3] with ϵ =1e-10.

respectively. Very complex structures like in the 6^{th} image can apparently not be handled by our autoencoder. What we are seeing as Latent Representation of Test Images is the average over all filters of what is labeled as "Encoded Representation" in Figure 3.1.

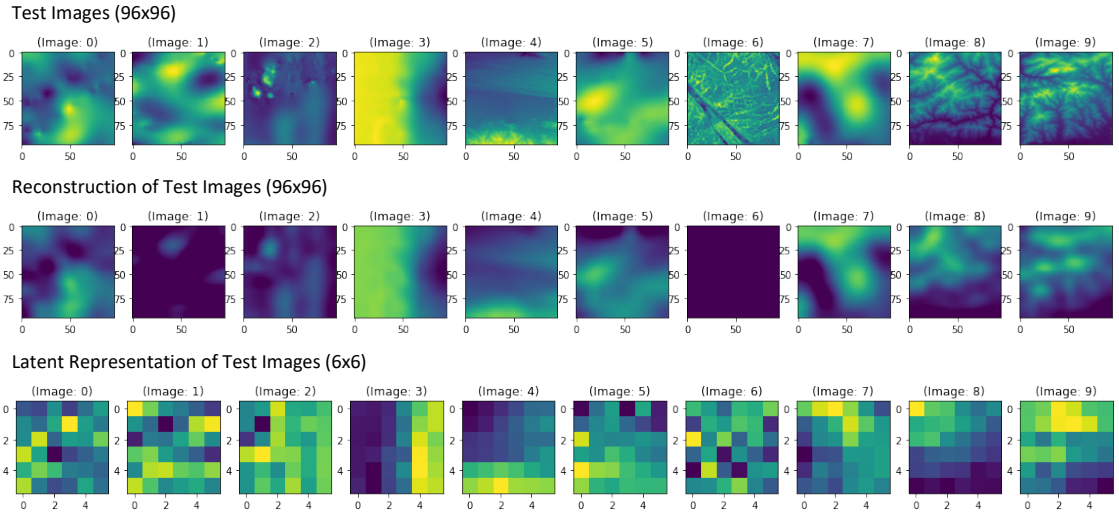

Figure 3.3 Autoencoder performance on test images. The latent representation is showing the average over the encoded space.

From a quantitative perspective, the approximation appears to be sufficiently good. Passing the whole GEBCO dataset through the trained autoencoder, we take the Euclidean distance between original and reconstruction. For the given setup, we are getting a total of 455.65, with a maximum per sample error of 5.95 (Figure 3.4). The latter, i. e. the worst reconstruction, is not coming from the test set. Visualizing this reconstruction, it appears the model cannot handle strong relative bathymetry changes within a short distance.

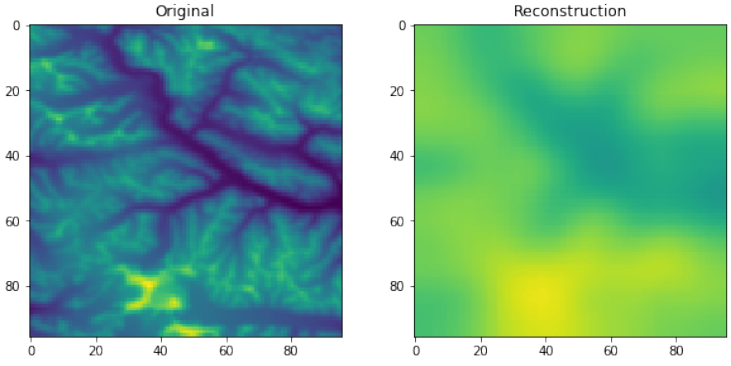

Figure 3.4 Autoencoder – maximum reconstruction error.

We conclude that this AE's reconstructions work sufficiently well and to go forward with more advanced, generative models.

3.1.2 Plain Variational Autoencoder

We build upon the previously presented autoencoder. The underlying main architecture remains unchanged, i. e. the same amount of convolutional layers, the same number of kernels, the same batch size. However, following the theory presented in the previous chapter, the VAE's architecture now looks as follows (Figure 3.5).

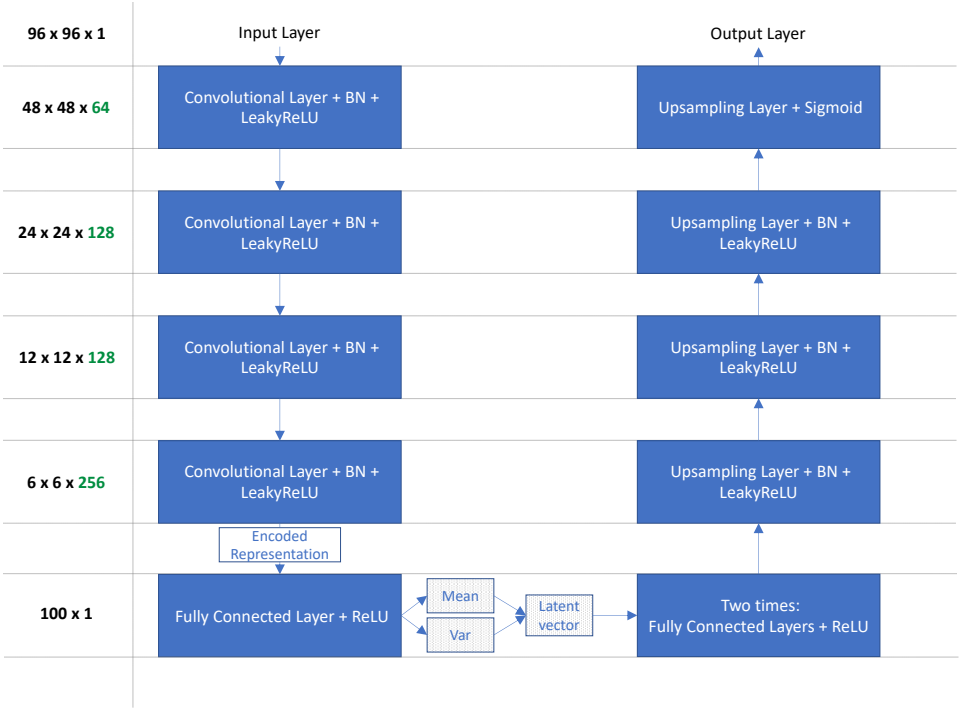

Figure 3.5 VAE implementation architecture.

This means we obtain a reduction from a 96×96 input to a single 100-dimensional latent vector. A tremendous reduction from 9,216 elements to 100.

The beginning of the decoding network has two fully connected layers to give the network more learnable parameters before switching over to convolutional upsampling layers. In addition to the VAE specific KL divergence loss, we again used the Charbonnier loss and the Adam optimization method. As our main goal is to obtain good reconstructions, we give more weight to the pure reconstruction loss than the KL loss. This makes the network focus on improving reconstruction quality rather than perfectly approximating the true latent distribution.[4] We trained for 100 epochs and, compared to before, with a much lower learning rate of 1e-6. Figure 3.6 shows the logarithmic loss development over time.

[4] Assume we wanted to use the latent distribution to generate new, realistic data, we would have to give less weight to the reconstruction loss instead and thereby trade-off some reconstruction quality. [Yan+19] provides some more in-depth reasoning on this.

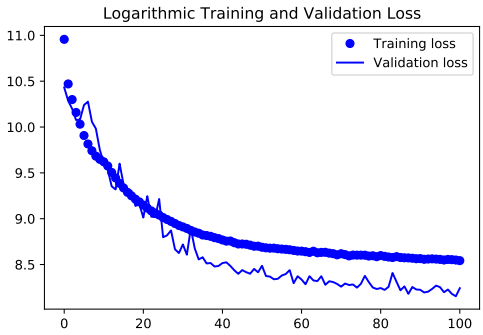

Figure 3.6 VAE loss history.

As before, we look at the reconstruction performance on the test set (Figure 3.7). The total L2 reconstruction error is at 307.88, the maximum per sample error at 10.36 (Figure 3.8). So, even though the plain VAE outperforms the AE from a quantitative perspective, the reconstructions are noisy and, thus, the plots appear blurry.

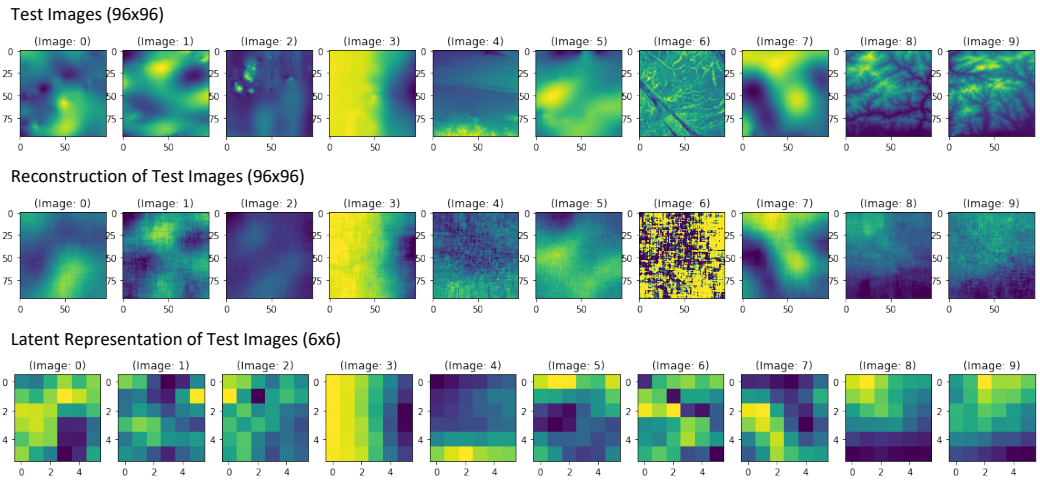

Figure 3.7 VAE performance on test images. The latent representation is the latent space's average.

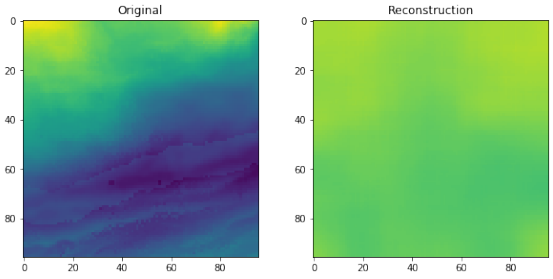

Figure 3.8 VAE – maximum reconstruction error.

The VAE's worst reconstruction seems to confirm the previous assumption of having weaker reconstructions when the input has a strong, sudden surface change. Again, the highest error sample is not part of the test set.

3.1.3 Deep Feature Consistent Variational Autoencoder

Starting from the plain VAE described in the previous subsection, we join it together with the VGGNet-19 through a perceptual loss on the first four layers. We use a weighting for the layers as we do not want to give too much credibility to this network that was trained on non topographical data.[5] We therefore give a weighting of $[1.0, 0.75, 0.5, 0.5]$ to the perceptual loss of VGGNet-19 layers one to four. We train our DFC VAE using identical hyperparameters compared to the plain VAE, but train for 115 epochs. The reconstructions seem to capture features much better than those of the previous models. However, they still appear to fail at reconstructing certain specific surfaces like in the 6^{th} image (Figure 3.9).

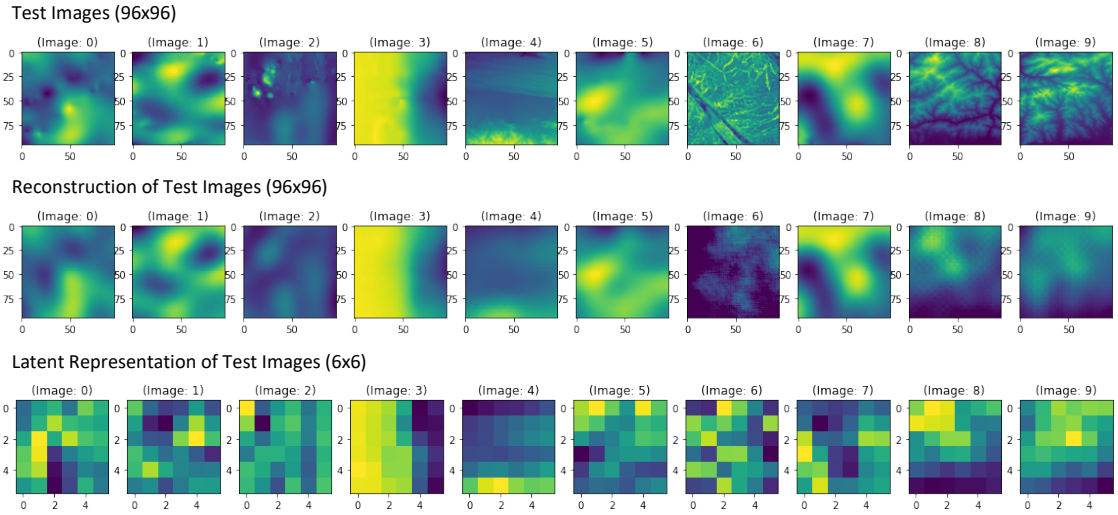

Figure 3.9 DFC VAE performance on test images. The latent representation is showing the average over the latent space.

Looking at the reconstruction L2 errors again, we get a value of 213.26 for the entire dataset – much lower compared to the previous models – and a maximum reconstruction error of 7.36 on a single sample (Figure 3.10). We observe the same pattern again, the worst reconstruction is on a training set image with large topographic changes.

[5] Experiments with the perceptual loss layer weighting seem to indicate that – in our case – taking the full perceptual loss into the model, i. e. no weighting, results in poor reconstructions. The same goes for using too many layers for the perceptual loss.

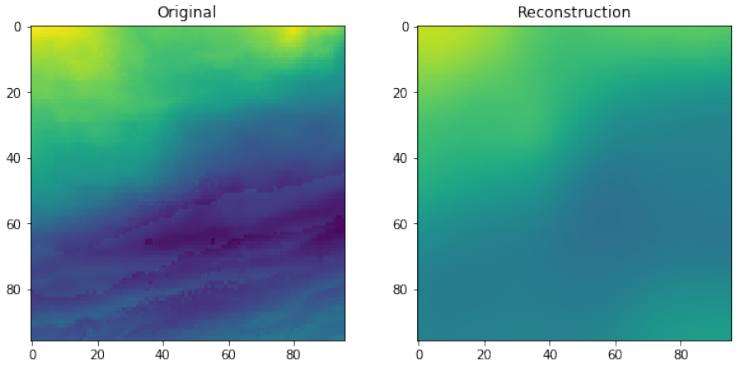

Figure 3.10 DFC VAE – maximum reconstruction error.

3.1.4 Variational Autoencoder - Generative Adversarial Network

Contrary to our expectations, our VAE implementation combined with a GAN failed to produce reasonable reconstructions, both qualitatively and quantitatively. It is still left to be explored whether or not this can be overcome when relying on different training and optimization procedures. The performance on the test set is also rather poor (Figure 3.11). Though, it appears as if the encoder network is working, as some similarity to some extent can be recognized (colors are sometimes inverted). As it currently stands, our VAE-GAN model has a total reconstruction error of above 600 with a maximum per sample error of 31.46 (Figure 3.12).

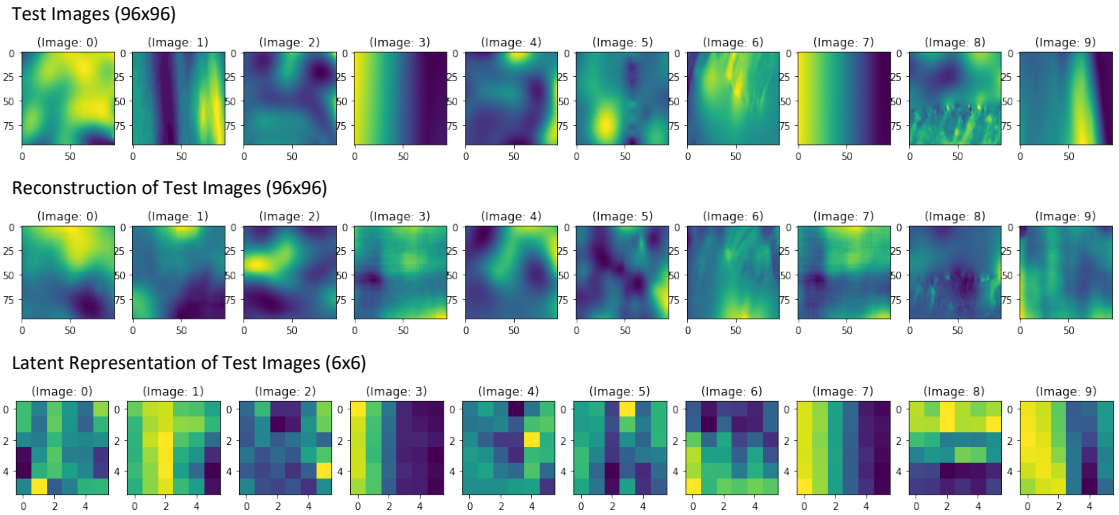

Figure 3.11 VAE-GAN performance on test images. The latent representation is showing the average over the latent space.

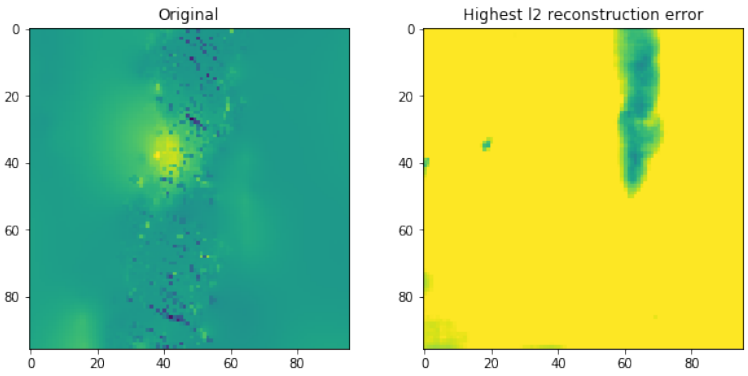

Figure 3.12 DFC VAE – maximum reconstruction error.

Looking at the model's loss development, we can see that the discriminator almost always recognizes the fake picture. Figure 3.13 shows the model accuracy with which it tells true from fake over the epochs. This would typically indicate that the discriminator is simply learning too fast. But experimenting with different hyperparameter settings has repeatedly resulted in the same pattern. One possible explanation might be that VAE-GAN models, or GANs in general, need less varying data as it is the case for, e. g., face attribute datasets. Here, we have many pictures which are all showing objects (faces) of similar shape and typically in the picture's focus. However, Bathymetry, or topography in general, is fundamentally different in this regard. While many parts of the GEBCO dataset look similar, specific details can vary a lot, as could be seen in many of the previous figures. We even have inter-sample dependencies. This means the generator has a much harder task to do, while for the discriminator it is much easier to learn what is true and what is not. Even when trying to train the VAE-GAN model based on a pre-trained VAE as generator, after a certain amount of training, the discriminator always wins.

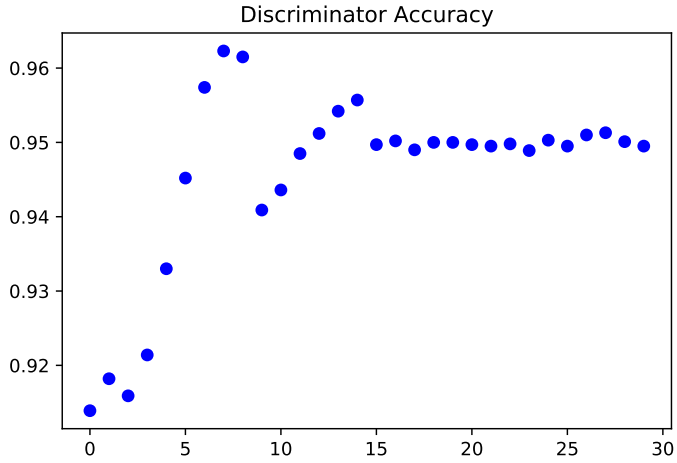

Figure 3.13 VAE-GAN – discriminator accuracy over the epochs.

3.2 Things That Did Not Work

For the sake of completeness, we would like to mention things that we tried but which did not work. Apart from the VAE-GAN model, there were a few other items:

- **Small input image size**
  While the input image size seemed to have mattered less for the plain autoencoder, it had a big impact on the other models. Initially, we trained the autoencoder based on a much smaller input image size of 60×60 with almost identical results. When moving on to variational autoencoders, this posed a rather big problem. Reconstructions were extremely poor and mostly random noise.

- **Regularization**
  Adding a penalty term to the networks cost function is usually expected to lead to higher performance on the test set. However, in our given set-up, both L1 and L2 penalties did not show such effects. Instead, they worsened results slightly.

- **Padding**
  Zero padding showed to be highly inferior to reflection padding. Analogously for no padding used at all.

- **Plain VAE**
  First, we started with a straight-forward implementation of the variational autoencoder, as it was introduced in the original paper ([KW13]). However, using only one sample to approximate the expected log-likelihood from Equation (2.8) turned out to be insufficient. Thus, we increased it to 100 – this number is solely based on heuristics.

4 Tsunami Simulation

In Chapter 3, we saw how certain models perform on reducing and reconstructing the GEBCO dataset. Over the course of the current chapter, we want to assess whether or not such reconstructions are suitable for highly complex applications like tsunami simulations. We are particularly interested in the Tohoku region, Northeast Japan (Figure 4.1). In 2011, this region has been hit by a huge tsunami, resulting from the fourth most powerful earthquake in the world since the begin of record-keeping.[1] This tragic event has cost many lives, caused nuclear accidents and even moved Honshu – Japan's main island – by some meters.[2] Having respective early warning systems in place can save many lives. Part of that is knowing where tsunami waves would potentially have the most severe impact on land. Therefore, we would not only like to simulate tsunamis but also be able to assess the uncertainty within this simulation. Quantifying this uncertainty is not part of this thesis' scope, however, having a reasonable parameter reduction of some kind will hopefully contribute to making this assessment easier. We use the ExaHyPE set-up described in Section 2.6 to run a tsunami simulation based on the best-performing model of Chapter 3 (i. e. the deep feature consistent variational autoencoder).

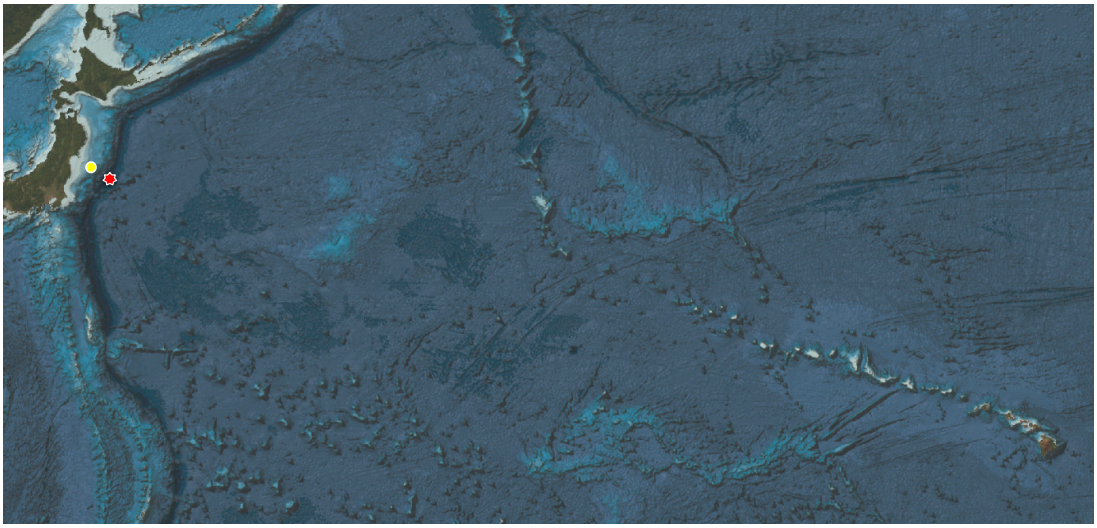

Figure 4.1 GEBCO bathymetry data east of Tohoku region, Japan. In red is the earthquake's location, in yellow the buoy's coordinates.

[1] `https://www.usgs.gov/natural-hazards/earthquake-hazards/science/20-largest-earthquakes-world`

[2] http://edition.cnn.com/2011/WORLD/asiapcf/03/12/japan.earthquake.tsunami.earth/index.html

Figure 4.2 shows the DFC VAE's reconstruction of the area near Tohoku.

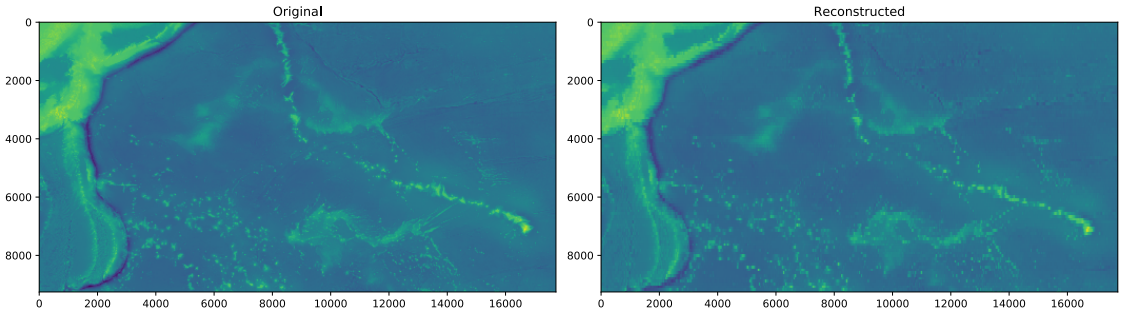

Figure 4.2 Reconstructed GEBCO bathymetry data east of Tohoku region, Japan.

We simulate a tsunami stemming from seismic activity around 150 kilometers east of the coast (the red star in Figure 4.1). We look at how a buoy in the water (yellow circle in the figure) moves when passing the original dataset through the simulation engine and when passing the DFC VAE reconstruction instead. For running the tsunami simulation, we need to specify the displacement of the bathymetry values, i. e. the earthquake. Unfortunately, we currently have these displacement values for the 2011 version of the GEBCO dataset only. This older version has a much lower resolution and less details. It is therefore quite different and, consequently, our model performs not as good on it (Figure 4.3).[3]

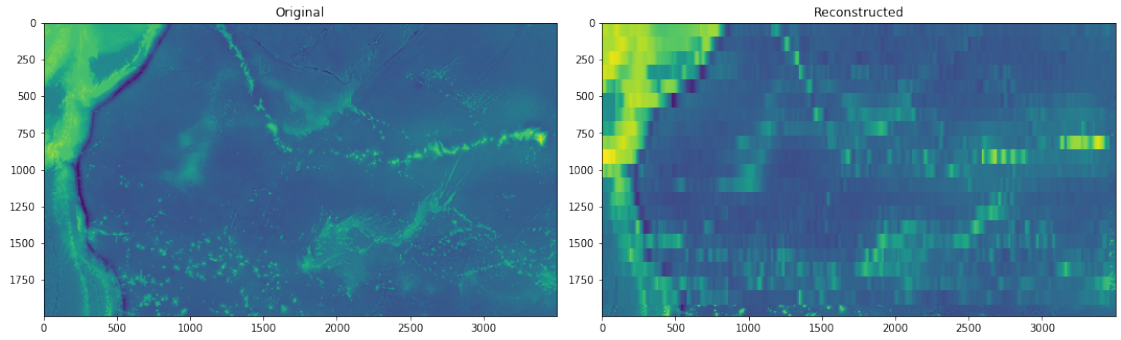

Figure 4.3 GEBCO 2011 bathymetry data east of Tohoku region, Japan.

Comparing Figures 4.2 and 4.3 shows fundamental differences between the two versions. The snippet shown in Figure 4.3, i. e. the 2011 version, has exactly 7 million data points. On roughly the same area, the 2019 version has over 160 million data points. Also, the long chain of (mostly) underwater elements, which is noticeably crossing through the area, follows an entirely different path on these versions. Nevertheless, we proceed with the current set-up – knowing that with updated displacement values, the results hereafter should improve.

[3]This should speak in favor of our model. This is a reconstruction which is entirely based on data the model has not seen during the training phase. Additionally, the data appears to be of different nature.

In addition to what was mentioned before, we also run the simulation over a flat bathymetry, i. e. a constant value. We look at a time window of $t = 7,500s$ starting from the moment the earthquake occurs and capture the current water level every 5 seconds. Figure 4.4 shows how the water height changes at the buoy's position over time. We can see a good approximation of the water heights coming from the DFC VAE's simulation basis and the water heights from the original bathymetry's simulation. This is especially true in the beginning, when the biggest wave passes the buoy (disregarding the time lag, i. e. shift on the x-axis). Later on, there are larger deviations but they are within the range of roughly a meter. The gray curve depicts a completely different outcome. As a result, it is safe to say that the DFC VAE's reconstruction works significantly better and produces results of much higher quality than a flat bathymetry.

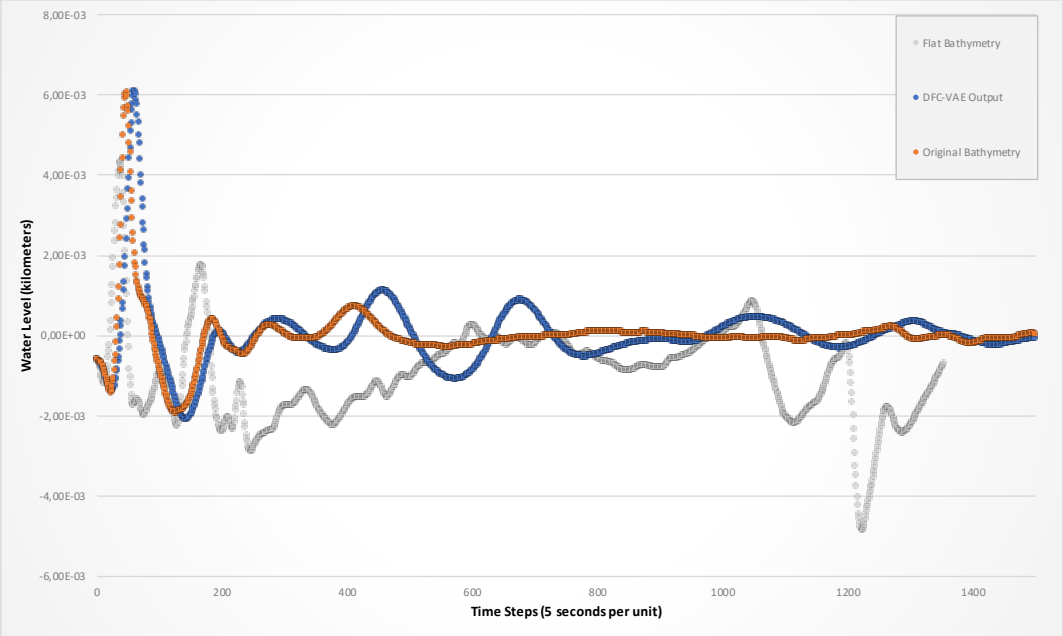

Figure 4.4 Water height at a fixed coordinate resulting from tsunami simulations.

5 Conclusion

Putting everything together, we have showed that we can find a reasonable mapping of topography data into a much lower dimensional space and a second mapping back to the original space. The best-performing model has achieved a reduction in the amount of parameters by almost 100-fold. This model's reconstructions work sufficiently well, both qualitatively and quantitatively speaking. Table 5.1 summarizes the key quantitative findings of all the models we investigated.

	Total L2 Error	**Maximum per Sample Error**
Autoencoder	455.65	5.95
Variational Autoencoder	307.88	10.36
Deep Feature Consistent Variational Autoencoder	213.26	7.36
Variational Autoencoder - Generative Adversarial Network	642.51	31.46

Table 5.1 Summary of the total and maximum per sample reconstruction errors.

The deep feature consistent variational autoencoder (DFC VAE) not only delivered the best results from a quantitative perspective, but also qualitatively. Thus, we concluded to use this model for further analysis. By passing it through a tsunami simulation in ExaHyPE, we showed that the DFC VAE's reconstruction appears to correctly capture major bathymetric attributes. The resulting wave propagations are – subject to a small time lag – close to the ones coming from the same tsunami simulation based on the original data.

Outlook

As previously indicated, one area of further improvement is, e. g., re-running the tsunami simulation exercise of Chapter 4 once the respective earthquake displacement values are available for the GEBCO 2019 version (or later). This is expected to significantly improve the model's performance on the simulation evaluation. A second item could be evaluating options to incorporate the bathymetry's gradient into the loss functions. Many of the maximum per sample error visualizations suggest that the models are the weakest when there is a high (relative) change in the bathymetry within a short distance. Another

area of future research on this topic is to explore certain 3D model approaches. These aim at modeling a 3-dimensional surface instead of working with a 2-dimensional heat map approach. Though promising, these are novel approaches as they only recently have become "tractable" due to the leap forward in generally available computational power.

List of Figures

List of Tables

Bibliography

[Bla18] B. Blagojevic. *Generating Letters using Generative Adversarial Networks (GANs)*. Nov. 2018.

[Cho+15] F. Chollet et al. *Keras*. `https://keras.io`. 2015.

[Doe16] C. Doersch. "Tutorial on variational autoencoders." In: *arXiv preprint arXiv:1606.05908* (2016).

[Duc07] J. Duchi. "Derivations for linear algebra and optimization." In: *Berkeley, California* 3 (2007), pp. 2325–5870.

[GBB11] X. Glorot, A. Bordes, and Y. Bengio. "Deep sparse rectifier neural networks." In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[Goo+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative adversarial networks." In: *arXiv preprint arXiv:1406.2661* 4.5 (2014), p. 6.

[Hah+00] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit." In: *Nature* 405.6789 (2000), p. 947.

[Hin+12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors." In: *arXiv preprint arXiv:1207.0580* (2012).

[Hou+16] X. Hou, L. Shen, K. Sun, and G. Qiu. "Deep Feature Consistent Variational Autoencoder." In: *CoRR* abs/1610.00291 (2016). arXiv: `1610.00291`.

[IS15] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *arXiv preprint arXiv:1502.03167* (2015).

[KH+09] A. Krizhevsky, G. Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[Kha+19] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. "A survey of the recent architectures of deep convolutional neural networks." In: *arXiv preprint arXiv:1901.06032* (2019).

[KL51] S. Kullback and R. A. Leibler. "On information and sufficiency." In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[KLY18] R. Kleinberg, Y. Li, and Y. Yuan. "An alternative view: When does SGD escape local minima?" In: *arXiv preprint arXiv:1802.06175* (2018).

[Krö+93] B. Kröse, B. Krose, P. van der Smagt, and P. Smagt. "An introduction to neural networks." In: (1993).

[KW13] D. P. Kingma and M. Welling. "Auto-encoding variational bayes." In: *arXiv preprint arXiv:1312.6114* (2013).

[Lar+15] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther. "Autoencoding beyond pixels using a learned similarity metric." In: *arXiv preprint arXiv:1512.09300* (2015).

[LCB10] Y. LeCun, C. Cortes, and C. Burges. "MNIST handwritten digit database." In: *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).

[LeC85] Y. LeCun. "Une procedure d'apprentissage pour réseau a seuil asymetrique." In: *Proceedings of Cognitiva 85* (1985), pp. 599–604.

[Liu+15] Z. Liu, P. Luo, X. Wang, and X. Tang. "Deep learning face attributes in the wild." In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 3730–3738.

[MP69] M. Minsky and S. Papert. "Perceptron: an introduction to computational geometry." In: *The MIT Press, Cambridge, expanded edition* 19.88 (1969), p. 2.

[NH10] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines." In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[Par85] D. Parker. "Learning-logic (TR-47)." In: *Center for Computational Research in Economics and Management Science. MIT-Press, Cambridge, Mass* 8 (1985).

[Pla18] E. Plaut. "From principal subspaces to principal components with linear autoencoders." In: *arXiv preprint arXiv:1804.10253* (2018).

[Rei+20] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, et al. "ExaHyPE: an engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* (2020), p. 107251.

[RH86] D. E. Rumelhart and G. E. Hintonf. "Learning Representations by Back-Propagating Errors." In: *NATURE* 323 (1986), p. 9.

[RHW85] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[Ros58] F. Rosenblatt. "The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological review* 65.6 (1958), p. 386.

[ROV19] A. Razavi, A. van den Oord, and O. Vinyals. “Generating diverse high-fidelity images with vq-vae-2.” In: *Advances in Neural Information Processing Systems*. 2019, pp. 14837–14847.

[Rud16] S. Ruder. “An overview of gradient descent optimization algorithms.” In: *arXiv preprint arXiv:1609.04747* (2016).

[SMB10] D. Scherer, A. Müller, and S. Behnke. “Evaluation of pooling operations in convolutional architectures for object recognition.” In: *International conference on artificial neural networks*. Springer. 2010, pp. 92–101.

[SZ15] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” In: *International Conference on Learning Representations*. 2015.

[TA77] A. N. Tikhonov and V. Y. Arsenin. “Solutions of ill-posed problems.” In: *New York* (1977), pp. 1–30.

[Tib96] R. Tibshirani. “Regression shrinkage and selection via the lasso.” In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288.

[Wei19] T. Weinzierl. “The Peano software - parallel, automaton-based, dynamically adaptive grid traversals.” In: *ACM Transactions on Mathematical Software* 45.2 (2019). (arXiv:1506.04496), 14:1–14:41.

[Wer74] P. Werbos. “Beyond Regression:" New Tools for Prediction and Analysis in the Behavioral Sciences.” In: *Ph. D. dissertation, Harvard University* (1974).

[Wu+17] B. Wu, H. Duan, Z. Liu, and G. Sun. “SRPGAN: perceptual generative adversarial network for single image super resolution.” In: *arXiv preprint arXiv:1712.05927* (2017).

[Yan+19] C. Yan, S. Wang, J. Yang, T. Xu, and J. Huang. “Re-balancing Variational Autoencoder Loss for Molecule Sequence Generation.” In: *arXiv preprint arXiv:1910.00698* (2019).

[Zan+15] O. Zanotti, F. Fambri, M. Dumbser, and A. Hidalgo. “Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting.” In: *Computers & Fluids* 118 (2015), pp. 204–224.

[Zha+20] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. `https://d2l.ai`. 2020.