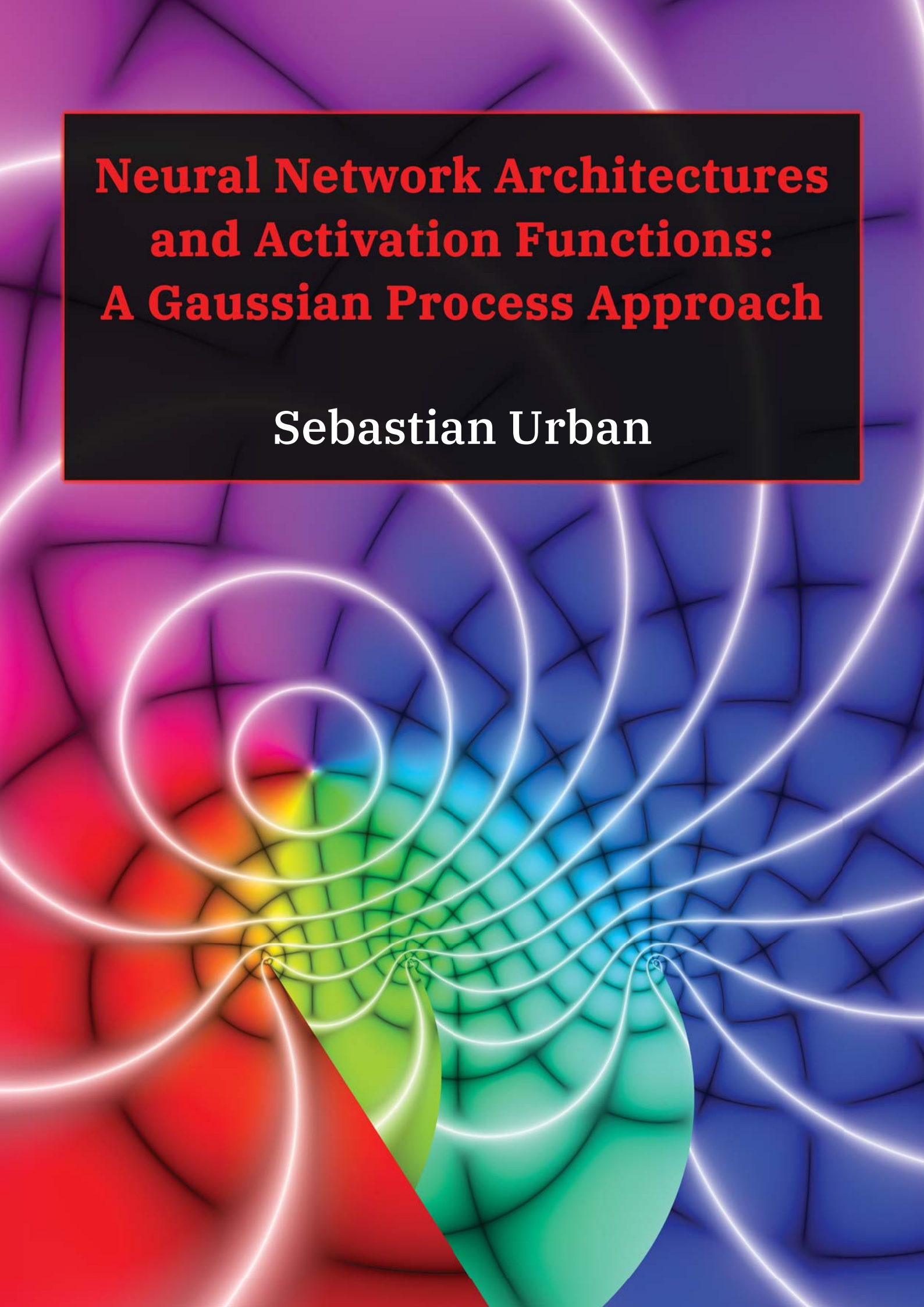
Neural Network Architectures and Activation Functions: A Gaussian Process Approach

Sebastian Urban



Fakultät für Informatik

Neural Network Architectures and Activation Functions: A Gaussian Process Approach

Sebastian Urban

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation.

Vorsitzender: Prof. Dr. rer. nat. Stephan Günnemann

Prüfende der Dissertation:

1. Prof. Dr. Patrick van der Smagt
2. Prof. Dr. rer. nat. Daniel Cremers
3. Prof. Dr. Bernd Bischl, Ludwig-Maximilians-Universität München

Die Dissertation wurde am 22.11.2017 bei der Technischen Universtät München eingereicht und durch die Fakultät für Informatik am 14.05.2018 angenommen.

Neural Network Architectures and Activation Functions: A Gaussian Process Approach

Sebastian Urban
Technical University Munich

2017

Abstract

The success of applying neural networks crucially depends on the network architecture being appropriate for the task. Determining the right architecture is a computationally intensive process, requiring many trials with different candidate architectures. We show that the neural activation function, if allowed to individually change for each neuron, can implicitly control many aspects of the network architecture, such as effective number of layers, effective number of neurons in a layer, skip connections and whether a neuron is additive or multiplicative.

Motivated by this observation we propose stochastic, non-parametric activation functions that are fully learnable and individual to each neuron. Complexity and the risk of overfitting are controlled by placing a Gaussian process prior over these functions. The result is the Gaussian process neuron, a probabilistic unit that can be used as the basic building block for probabilistic graphical models that resemble the structure of neural networks. The proposed model can intrinsically handle uncertainties in its inputs and self-estimate the confidence of its predictions. Using variational Bayesian inference and the central limit theorem, a fully deterministic loss function is derived, allowing it to be trained as efficiently as a conventional neural network using stochastic gradient descent. The posterior distribution of activation functions is inferred from the training data alongside the weights of the network. The proposed model favorably compares to deep Gaussian processes, both in model complexity and efficiency of inference. It can be directly applied to recurrent or convolutional network structures, allowing its use in audio and image processing tasks. As an empirical evaluation we present experiments on regression and classification tasks, in which our model achieves performance comparable to or better than a Dropout regularized neural network.

We further develop a novel method for automatic differentiation of elementwise-defined, tensor-valued functions that occur in the mathematical formulation of Gaussian processes. The proposed method allows efficient evaluation of the derivatives on modern GPUs and is used in our implementation of the Gaussian process neuron to achieve computational performance that is about 25% of a convential neuron with a fixed logistic activation function.

Acknowledgements

First and foremost I would like to thank my advisor *Patrick van der Smagt* for the amount of freedom and encouragement he provided. It was the foundation that allowed me to handle the ambitious projects that cumulated into this thesis.

I would also like to sincerely thank *Wiebke Köpp* for very helpful discussions and all the time, effort and energy she spent on building and testing an efficient implementation of the concepts presented in chapter 3.

Just as well I would like to extend my utmost gratitude to *Marcus Basalla* for very helpful discussions, his contributions to the concepts developed in chapter 4 and for the time and effort he spent on implementing and performing very detailed emperical evaluations of the models developed in chapter 5.

I would further like to extend many thanks to *Marvin Ludersdorfer* and *Mark Hartenstein*. Although they did not directly contribute to this work, we worked together on a predecessor project involving Gaussian processes that inspired many ideas used and developed within this thesis.

I would also like to show utmost appreciation to my parents *Jolanta* and *Janusz Urban* for the amount of support they offered during my studies and especially to my mother for taking the time to thoroughly proofread this thesis.

Finally, I would like to thank my lab colleagues *Justin Bayer*, *Christian Osendorfer*, *Nutan Chen*, *Markus Kühne*, *Philip Häusser*, *Maximilian Sölch*, *Maximilian Karl*, *Benedikt Staffler*, *Grady Jensen*, *Johannes Langer*, *Leo Gissler* and *Daniela Korhammer* for many inspiring discussions and their ability to tolerate the significant overlength of most of my presentations.

List of Publications

The following list shows the publications that originated during the course of this work.

Sebastian Urban, Patrick van der Smagt (2017). “Gaussian Process Neurons”. *International Conference on Learning Representations (ICLR) 2018 (submitted)*.

Sebastian Urban, Patrick van der Smagt (2017). “Automatic Differentiation for Tensor Algebras”. *arXiv preprint arXiv:1711.01348 [cs.SC]*.

Sebastian Urban, Patrick van der Smagt (2015). “A Neural Transfer Function for a Smooth and Differentiable Transition Between Additive and Multiplicative Interactions”. *arXiv preprint arXiv:1503.05724 [stat.ML]*.

Sebastian Urban, Marvin Ludersdorfer, Patrick van der Smagt (2015). “Sensor Calibration and Hysteresis Compensation with Heteroscedastic Gaussian Processes”. *IEEE Sensors Journal 2015*.

Sebastian Urban, Justin Bayer, Christian Osendorfer, Goran Westling, Benoni Edin, Patrick van der Smagt (2013). “Computing Grip Force and Torque from Finger Nail Images using Gaussian Processes”. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 2013*.

Wiebke Köpp, Patrick van der Smagt, Sebastian Urban (2016). “A Differentiable Transition Between Additive and Multiplicative Neurons”. *International Conference on Learning Representations (ICLR) 2016 (workshop track)*.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim

Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, Ying Zhang (2016). “Theano: A Python Framework for Fast Computation of Mathematical Expressions”. *arXiv preprint arXiv:1605.02688 [cs.SC]*.

Nutan Chen, Justin Bayer, Sebastian Urban, Patrick van der Smagt (2015). “Efficient Movement Representation by Embedding Dynamic Movement Primitives in Deep Autoencoders”. *IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids) 2015*.

Nutan Chen, Sebastian Urban, Justin Bayer, Patrick van der Smagt (2015). “Measuring Fingertip Forces from Camera Images for Random Finger Poses”. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 2015*.

Andreas Vollmayr, Stefan Sosnowski, Sebastian Urban, Sandra Hirche, J Leo van Hemmen (2014). “Snookie: An Autonomous Underwater Vehicle with Artificial Lateral-Line System”. *Flow Sensing in Air and Water 2014*.

Nutan Chen, Sebastian Urban, Christian Osendorfer, Justin Bayer, Patrick van der Smagt (2014). “Estimating Finger Grip Force from an Image of the Hand using Convolutional Neural Networks and Gaussian Processes”. *IEEE International Conference on Robotics and Automation (ICRA) 2014*.

Justin Bayer, Christian Osendorfer, Daniela Korhammer, Nutan Chen, Sebastian Urban, Patrick van der Smagt (2013). “On Fast Dropout and its Applicability to Recurrent Networks”. *arXiv*

preprint arXiv:1311.0701 [stat.ML].

Justin Bayer, Christian Osendorfer, Sebastian Urban, Patrick van der Smagt (2013). “Training Neural Networks with Implicit Variance”. *International Conference on Neural Information Processing (ICONIP) 2013*.

Christian Osendorfer, Justin Bayer, Sebastian Urban, Patrick van der Smagt (2013). “Convolutional Neural Networks Learn Compact Local Image Descriptors”. *International Conference on Neural Information Processing (ICONIP) 2013*.

Christian Osendorfer, Justin Bayer, Sebastian Urban, Patrick van der Smagt (2013). “Unsupervised Feature Learning for Low-Level Local Image Descriptors”. *arXiv preprint arXiv:1301.2840 [cs.CV]*.

Ulrich Rührmair, Christian Hilgers, Sebastian Urban, Agnes Weiershäuser, Elias Dinter, Brigitte Forster, Christian Jirauschek (2013). “Optical PUFs Reloaded”. *IACR Cryptology ePrint Archive*.

Ulrich Rührmair, Christian Hilgers, Sebastian Urban, Agnes Weiershäuser, Elias Dinter, Brigitte Forster, Christian Jirauschek (2013). “Revisiting Optical Physical Unclonable Functions”. *IACR Cryptology ePrint Archive*.

x

Contents

Chapter 1

Introduction

Artificial neurons (Haykin, 1994) are a model of biological neurons as they exist in biological brains. An artificial neuron receives inputs (real numbers) from other neurons, calculates a weighted sum of it and applies a non-linear activation function to the result, producing its output. Neurons are arranged in artificial neural networks (ANNs), which are graphs that describe how the output of each neuron connects to the inputs of other neurons. Of course, many different graphs are imaginable. One simple, yet common, type of ANNs is the feed-forward network, where neurons are arranged in layers and all neurons of one layer are connected to all neurons of the next layer. The bottom and top layers are called input and output layer respectively, while the in-between layers are called hidden layers. By iteratively calculating weighted sums and applying non-linearities any function can be represented if the network is large enough. Changing the weights changes the function and so we can "train" a network from data consisting of pairs of input and target samples by adjusting the weights so that for each input the output of the ANN resembles the target as much as possible. Since an ANN is nothing more than a function with many parameters, we can do this by calculating the gradient of some measure of how bad the network is at hitting the targets w.r.t. all weights and then use some (advanced) form of gradient descent to iteratively minimize that measure. This algorithm is known as backpropagation (Rumelhart, G. E. Hinton, et al., 1988). "Deep learning" (Y. LeCun, Bengio, et al., 2015), which currently spawns a considerable amount of research interest, refers to the training of ANNs that have many layers and are thus "deep".

Besides feed-forward networks two important ANN classes exists. A recurrent neural network (RNN) is a neural network architecture for processing a (time) sequence of inputs. It is evaluated over a series of steps, each corresponding to an item in an input sequence. The structure of an RNN matches the structure of a feed-forward network but with additional, recurrent connections on the hidden units. These connections specify how the state of the hidden layer neurons is propagated from one step to the next. Like a feed-forward network it

is trained by minimizing some loss measure with respect to its weights using the backpropagation through time algorithm (Werbos, 1990). Thus, given an input sequence of, for example, speech-recording, an RNN can be trained to output a sequence of the spoken words in text form.

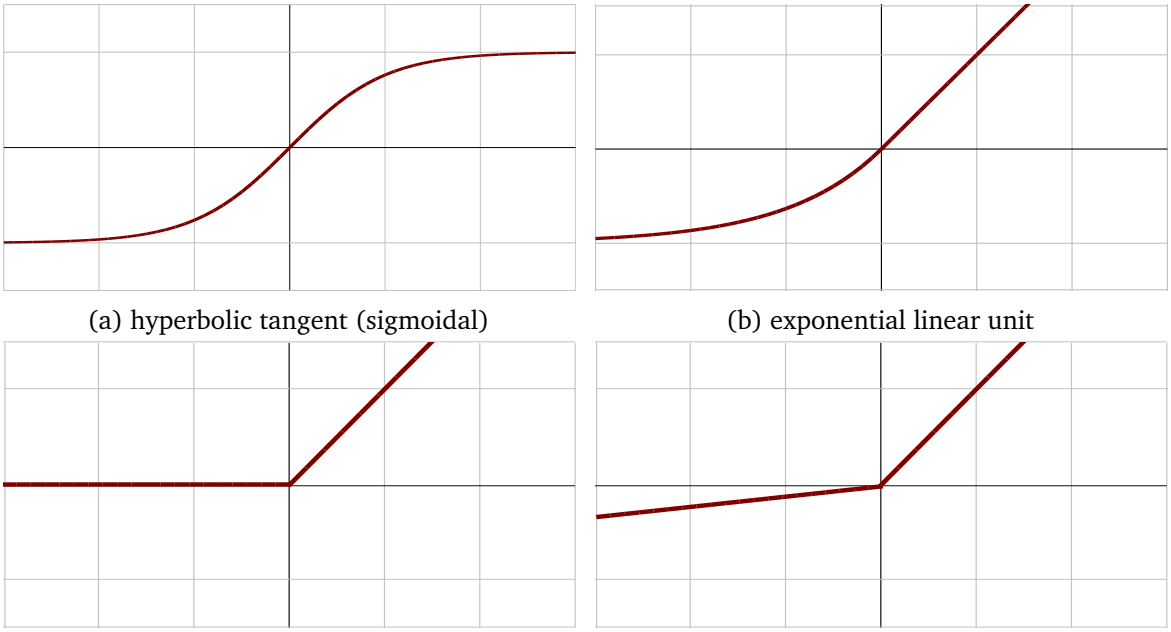
A convolutional neural network (CNN) is a network architecture developed for image processing and recognition (Y. LeCun, B. E. Boser, et al., 1990; Y. LeCun, B. Boser, et al., 1989). It is modeled after the visual cortex in biological brains and consists of a stack of alternating convolutional and pooling layers. The activations of the neurons in a convolutional layer are given by a convolution of the outputs of the previous layer with the weights, which are shared between all neurons in that layer. The motivation for this operation is that an object in an image should produce the same response without respect to its position. A pooling layer divides its inputs into blocks of a fixed size (for example 4x4) and applies a reduction operation on these blocks to compute its output. A common choice for this reduction operation is taking the maximum value, thus adding a certain amount of invariance to translations. Due to weight sharing CNNs only have a fraction of the weights a feed-forward network of the same size would have. This significantly reduces the amount of required training data, memory and training time and thus CNNs have become the network architecture of choice for image processing tasks.

1.1 State of the Art

We review the state of the art in neural activation functions, multiplicative interactions in neural networks, structure search, stochastic neural networks and their connection to regularization.

Activation Functions

Since the dawn of neural network research the most commonly used activation functions were the logistic function and other sigmoid functions. Sigmoid refers to “S”-shaped functions, for example the hyperbolic tangent (tanh). The use of the logistic function was originally inspired by the thresholding behavior of biological neurons and because its derivative is benign and can be computed very inexpensively. This choice of activation functions was not seriously challenged by researchers (except for special purpose applications), until recently when Nair et al. (2010) introduced the rectified linear unit (ReLU), a neuron with an activation function that is linear for positive inputs and zero for negative inputs. Krizhevsky, Sutskever, et al. (2012) showed that ReLUs produce significantly better results on image recognition tasks using deep networks than the common sigmoid-shaped activation functions. One possible explanation for this success is that, when a sigmoid-shaped function is used in its saturated (flat) areas, its



(a) hyperbolic tangent (sigmoidal)

(b) exponential linear unit

(c) ReLU

(d) leaky ReLU

Figure 1.1: Commonly used activation functions in neural networks.

derivative is near zero, thus leading to a weak training signal to the weights and hence training becomes slow or gives worse results. This problem is amplified in deep neural networks. There, the training signal travels through many layers and thus many potentially saturated neurons until it reaches the weights in the lower layers. Since, by the chain rule, the derivative of a composed function is the product of each of the derivatives of its composing functions, the training signal can get exponentially small. The ReLU does not have this problem, at least in the positive range, because there its derivative is simply one.

This achievement led to a wave of follow-up research in activation functions specifically tailored to deep networks. While the ReLU solved the problem of vanishing gradients for positive values, it completely cut off the gradient for negative ones; thus once a neuron enters the negative regime (either through initialization or during training) for most samples, no training signal can pass through it. To mitigate this problem Maas et al. (2013) introduced the leaky ReLU, which is also linear for negative values but with a very small, although non-zero, slope; for positive values it behaves like the ReLU. Soon after He et al. (2015) demonstrated that it is advantageous to make the slope of the negative part of the leaky ReLU an additional parameter of each neuron. This parameter was trained alongside the weights and biases of the neural network using gradient descent. A CNN using these so-called parametric ReLUs was the first to surpass human-level performance on the ImageNet classification task (Deng et al.,

2009). Commonly used activation functions are shown in fig. 1.1.

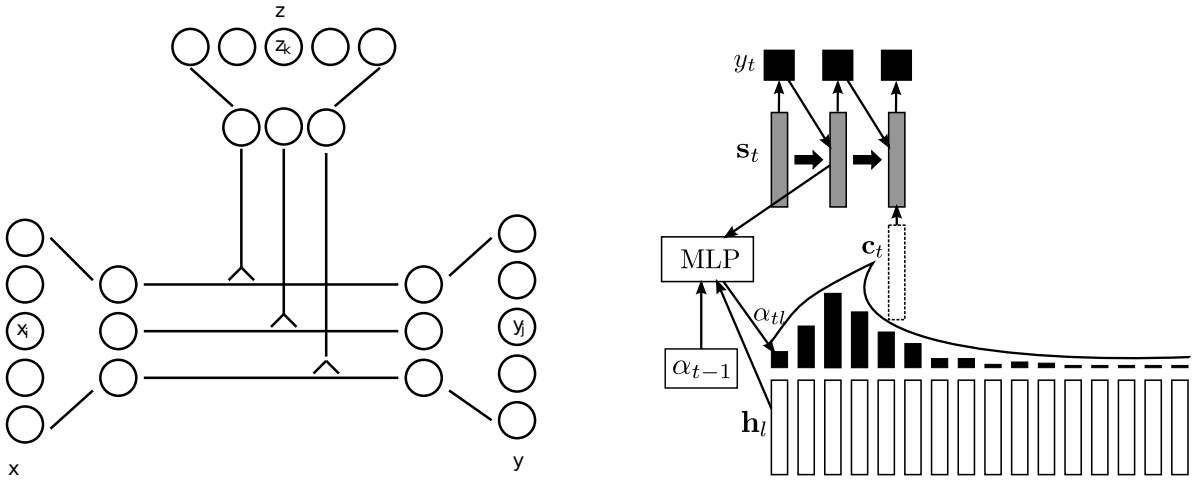
It is thus natural to ask if even more flexible activation functions are beneficial. This question was answered affirmative by Agostinelli et al. (2014) on the CIFAR-10 and CIFAR-100 benchmarks (Krizhevsky, Nair, et al., 2014). The authors introduced piecewise linear activation functions that have an arbitrary (but fixed) number of points where the function changes it slope. These points and the associated slopes are inferred from training data by stochastic gradient descent. Maxout networks (Goodfellow et al., 2013) consist of neurons that use no activation function but instead take the maximum over a set of different linear combinations of their inputs to compute their output. This, however, also results in piecewise linear segments in their outputs and thus the slopes of the different segments of piecewise linear activation functions can be encoded in the weights of maxout networks.

Instead of having a fixed parameter for the negative slope of the ReLU, Xu et al. (2015) introduced stochasticity into the activation function by sampling the value for the slope with each training iteration from a fixed uniform distribution. Clevert et al. (2015) and Klambauer et al. (2017) replaced the negative part of ReLUs with a scaled exponential function and showed that, under certain conditions, this leads to automatic renormalization of the inputs to the following layer and thereby simplifies the training of the neural networks, leading to accuracy improvements of deep feed-forward networks on tasks from the UCI Machine Learning repository (Lichman, 2013) amongst others.

Nearly fully adaptable activation functions have been proposed by Eisenach et al. (2017). The authors use a Fourier basis expansion to represent the activation function; thus with enough coefficients any (periodic) activation function can be represented. The coefficients of this expansion are trained as network parameters using stochastic gradient descent. Similarly, Scardapane et al. (2017) also use a basis expansion, but with a set of Gaussian kernels that are equally distributed over a preset input range.

Neurons that Multiply their Inputs

Let us introduce another type of artificial neuron. It has the same structure as the neuron introduced before, but instead of summing over its inputs, it calculates the product over them to compute its output. In such a multiplicative neuron the weights are not multiplied with the inputs but used as their exponents in the product. A unit that combines the calculation of a product followed by a summation (thus computing a polynomial over its inputs) is called Sigma-Pi unit, where Sigma stands for sum and Pi stands for product, and has been introduced by Rumelhart, McClelland, et al. (1987) and Shin et al. (1991). Networks containing units that perform multiplications are called higher-order neural networks. The power of Sigma-Pi units



(a) relating two images (b) attention for speech recognition

Figure 1.2: Examples for multiplicative interactions in neural networks. (a) Tripartite graph consisting of two images and their relation vector; from (Memisevic, 2011). (b) Attention mechanism over hidden units of RNN scanning over input sequence; from (Bahdanau, J. Chorowski, et al., 2016).

can be seen from Taylor series, which express every analytic function as a (possibly infinite) polynomial. Thus a Sigma-Pi unit with enough products can represent any function. Although intriguing in what they can represent, polynomials are also problematic with respect to stability, since any finite polynomial will approach infinity as the magnitudes of its variables increase. This makes higher-order neural networks significantly more challenging to train, which led to a focus of research on purely additive ANNs for many years.

Multiplicative interactions reappeared when Memisevic (2011) and Memisevic (2013) showed that they are necessary to represent the relationship between two images. The authors assume that one image can be transformed into the other by application of simple transformations such as translations and rotations to patches of the image. The proposed model is a tripartite graph (fig. 1.2a) consisting of both images and a transformation vector, the entries of which act multiplicatively on the pixels of one image to obtain the other image. Building upon this work, Alain et al. (2013) proposed an autoencoder (Bengio, 2009) containing multiplicative neurons to model the relation between two images with fewer weights and greater generalization performance.

Multiplications also proved very beneficial in the context of speech recognition and machine translation using RNNs. J. K. Chorowski et al. (2015) and Bahdanau, J. Chorowski, et al. (2016) showed that adding an attention mechanism to a speech recognition model can significantly improve its prediction accuracies. The proposed model (fig. 1.2b) consists of two RNNs, a

encoder and a decoder, and a feed-forward network controlling the attention. The encoder is used to produce a sequence of latent state vectors from the input audio. The decoder uses this sequence to to generate the output text. However, it does not process it sequentially, but its input in each step is given by a linear combination of the latent state vectors. The weights of this linear combination are computed by the attention feed-forward network that takes the previous state of the decoder as input. Thus the decoder chooses to which parts of the encoded input sequence it "attends" to for each output word. Because the linear combination is a multiplication of the outputs of two neural networks, the proposed model belongs to the class of higher-order neural networks. The same principle was applied by Bahdanau, Cho, et al. (2014) to machine translation and achieved significant improvements over models without an attention mechanism.

The Search for Network Architectures

Choosing the structure of an ANN suitable for a particular task is a non-trivial problem. The universal approximation theorem (Gybenko, 1989) guarantees that a network with one hidden layer can approximate any continuous function arbitrarily well; however, it does not bound the number of neurons required and thus such a flat network becomes too large to be practical when the function to approximate is high-dimensional (such as an image). In many cases ANNs consisting of multiple stacked layers are more efficient, i.e. they can learn the same function with less neurons and thus weights. An explanation for this effect is that in the lower layers simple function approximations are built that are then combined into more powerful functions by the subsequent layers.

However, when designing multi-layer architectures many question arise about their structure. How many layers should the network have? How many neurons should there be in each layer? Should the weights be shared according to some scheme (e.g. CNN)? What activation function should be used for the neurons in each layer? Should only additive neurons be used or will multiplicative units help?

A class of algorithms that can optimize the structure of neural network are evolution strategies (Rechenberg, 1973). These optimization algorithms mimic the progress of natural evolution on a population of artificial genomes. Each genome represents a possible solution to the problem at hand, for instance it can encode the structure of a neural network. An evolution strategy iteratively applies selection according to some fitness measure, mutation (random change) and crossover (combination) between two genomes on the population. Thus, over the progression of the algorithm the genomes will tend to produce individuals with higher fitness, although no gradient information are used. The success or failure of evolution strategies depends on the

encoding of the genome; if it is unsuitably chosen the operations of mutation and crossover will damage the population and no progress can be made.

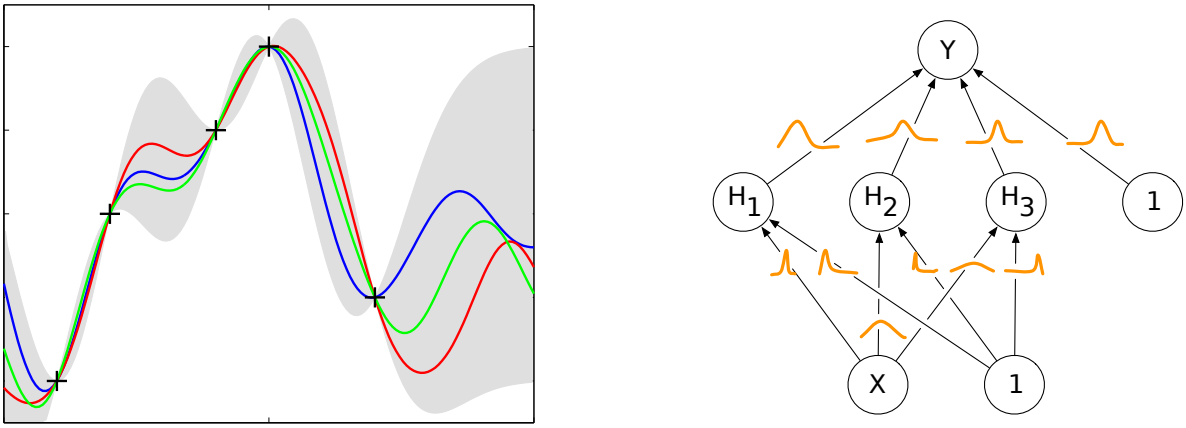
The first application of evolution strategies to feed-forward neural networks was proposed by Maniezzo (1994). The author encoded both the weights and structure in the genome, leading to a very large genome, which limited the applicability of this approach to small networks. On the other hand, NEAT (Stanley and Miikkulainen, 2002) evolves only the network structure by augmenting it from generation to generation. The fitness of each genome is evaluated by instancing the corresponding network, training it using backpropagation on the data set until convergence and calculating the loss on a separate validation set. Since the weights are not part of the genome, NEAT can scale to very large networks. However, each round of evolution strategies needs considerable time to train all networks of the population before selection can take place. HyperNEAT (Stanley, D'Ambrosio, et al., 2009) follows the same principle as NEAT but uses Compositional Pattern Producing Networks (Stanley, 2007) to decode the network structure from the genome.

Another approach is to search for ANN structures using reinforcement learning (Sutton and Barto, 1998). Zoph et al. (2016) propose a generator RNN that outputs a sequence of tokens that encodes the architecture of a CNN. As with NEAT each network architecture is decoded from the token-sequence, instantiated, trained and its loss on a validation set evaluated. The negated loss is then used as a reward for the generator RNN, which is optimized using the REINFORCE rule (R. J. Williams, 1992) with the derivative of the reward signal being approximated using the policy gradient method (Sutton, McAllester, et al., 2000). This method produced good results on the CIFAR-10 benchmark, however like evolution strategies it is computationally demanding as many candidate networks must be trained.

Uncertainty meets Regularization

ANNs as described so far are deterministic models. After being trained the network produces a deterministic output for each given test input. This output corresponds to a best guess estimate of what the network considers the appropriate target. There are mainly two motivations for extending ANNs into the probabilistic domain. First, performing Bayesian inference in a probabilistic model introduces a natural resiliency against overfitting when an appropriate prior is chosen. Second, it would be useful if the ANN could output how certain it is about its predictions, so that one knows how much trust can be put into those.

Overfitting means that the model places too much importance on getting the predictions on the training set *exactly* right and thus lets random noise and outliers heavily influence its output, which leads to a significant decrease in accuracy on the test set. Regularization is a concept for



(a) GP regression (b) neural weight uncertainty

Figure 1.3: Examples for uncertainty in machine learning models. (a) GP regression has lower confidence (shaded area) when no training point (cross) is nearby; from (Rasmussen et al., 2006). (b) Uncertain weights in a neural network parameterized using normal distributions; from (Blundell et al., 2015).

preventing statistical models from overfitting. An example is L2-regularization, which was first proposed by Tikhonov et al. (1977) in the context of logistic regression. For neural networks this method works by adding a penalty given by the magnitude of the weight vector of each neuron to the loss function. Thus it "encourages" each neuron to keep its activation in or close to the linear range of the (usually) sigmoidal activation function. In consequence, using this augmented loss as training objective will lead a multi-layer ANN to learn functions that have a more linear functional dependency on their inputs. Since this limits the classes of functions that can be learned and thus the hypothesis space, learning theory (Vapnik, 2013) predicts that the risk of overfitting is thereby reduced. The probabilistic pendant of this method is to use Bayesian inference with a Gaussian prior with zero mean on the weights. A more advanced regularization method for neural networks is Dropout (G. E. Hinton, Srivastava, et al., 2012; Srivastava et al., 2014), which randomly disables (sets their output to zero) neurons during training and thus prevents neurons in subsequent layers from co-adaptation by implicitly performing model averaging.

For many applications of predictive models it is essential to have an estimate about the uncertainty of the prediction given an input. For example, for an automated medical diagnosis system it is imperial that not only the most probable diagnosis is outputted but also how much trust can be put into the result. Gaussian process regression and classification (Rasmussen et al., 2006) provide such estimates by performing probabilistic linear regression in a high dimensional feature space computed from its inputs. Here the estimate of uncertainty depends

on the test input. The model will be more certain about predictions it makes on inputs that are similar to samples from the training set, while inputs that have hardly any resemblance to the training set will lead to predictions afflicted with high levels of uncertainty, cf. fig. 1.3a. For example, Urban, Bayer, et al. (2013) use GPs to compute grip forces and their confidence interval from video sequences of fingernails. Furthermore Urban, Ludersdorfer, et al. (2015) embed GPs within a time series filter to compensate for the hysteresis of a tactile sensor; in this work the confidence estimation is crucial for Bayesian optimal fusion of state predictions from a dynamical model and raw measurement data.

For neural networks it has been proposed (Graves, 2011; G. E. Hinton and Van Camp, 1993) to treat the weights of the ANN as probabilistic variables and apply variational inference (Bishop, 2006) to approximate their posteriors. The common choice for the variational posterior of the weights (fig. 1.3b) is a Gaussian distribution with diagonal covariance matrix. Thus, in such a probabilistic ANN the number of parameters is doubled compared to a deterministic network, since for each weight we now additionally store its variance. A further approximation is to model the output of each neuron as a normal distribution; the mean and variance being calculated from the means and variances of its inputs and the uncertainty of the weights. This method is similar to the technique of error propagation (Taylor, 1997) for scientific calculations using uncertain physical measurements (which are normally distributed in the majority of cases). Hence, the neurons in the output layer provide a prediction in form of the mean and variance parameters of a normal distribution. Blundell et al. (2015) have shown that the backpropagation algorithm can be adapted to train such probabilistic models.

In summary, treating neural networks probabilistically makes them less prone to overfitting and allows to reason about the certainty of their predictions.

1.2 Motivation for this Work

It is well known that $x\,y = \exp(\log x + \log y)$. It means that one can write a product as a sum with help of the exponential function and logarithm. This also works for negative numbers by using the complex versions of these functions. Now consider an (additive) neural network. By using the logarithm as the activation function in a layer and the exponential function in the next, we can perform multiplications without having to change the network architecture. Of course we can still apply the sigmoid after the exponential function, if we want to do some (soft) thresholding after the multiplication. Thus, solely by changing the activation function we can turn an additive neural network into a higher order neural network.

In terms of searching for the optimal architecture not much has been gained yet, since we just shifted the problem from how a neuron handles its inputs to the choice of its activation function. We still would have to instantiate many networks with different combinations of neurons using the exponential, logarithm and logistic activation functions. The reason for this is that the choice of activation function is discrete and thus it must be treated as a hyper-parameter since no derivative can be computed. However, if we could turn that discrete choice into a continuous spectrum of infinitely many activation functions, the derivative with respect to the choice of activation function would be well-defined. Thus, we could make the activation function an additional parameter (besides the bias) of each neuron and train it alongside the existing network parameters using backpropagation.

The activation function also determines the "existence" of a neuron. If it is constant zero, the network behaves as if that particular neuron and all its incoming and outgoing connections were not present at all. Furthermore it also determines the effective depth of the ANN. If a neuron uses a strictly linear activation function, the network also behaves like it was not present, but with its incoming connections directly connected to its outgoing connections and the weights adjusted accordingly (using the matrix dot product). Thus if all neurons within a layer use a linear activation function, the whole layer can be folded into the next layer, thus reducing the effective depth of the ANN. In summary, the choice of activation functions determines the effective architecture of an ANN with respect to additive or multiplicative interactions, number of neurons in a layer and number of layers.

If a (parametric) model should make predictions with confidence estimates, it needs to contain information about the certainty of its own parameters. Previous approaches to make predictions of ANNs probabilistic have modeled the uncertainty of the network weights, for example by approximating their posterior with a normal distribution, as discussed before and shown in fig. 1.3b. In this work we follow a different approach. We treat the weights as deterministic network parameters, but propose to mediate the model uncertainty through a

distribution over the activation functions of each neuron. Using activation functions which have an input-dependant variance, like the GP in fig. 1.3a, will make the certainty of the network depend on which region of the activation function is used. If areas of high confidence are used, the output of the neuron will have low variance. On the other hand, if the activation falls into a region of low confidence, the neuron output will have high variance and thus indicate that it is unsure about its output. By propagating this uncertainty through the network, the outputs of the network become predictive distributions that can be interpreted as confidence intervals. The weights influence how the uncertainty is propagated from one neuron to a subsequent neuron. If a connection from a neuron with high variance output to another neuron has a strong weight, then (by propagation of uncertainty) the receiving neuron will have an activation with high variance. However, if the weight is weak, then the high variance output does not have a high impact on the variance of the activation of the receiving neuron. Consequently, uncertainty in such a neural network can also decrease as the inputs are propagated from layer to layer.

Stochastic activation functions can also act as powerful regularizer on the ANN if Bayesian inference is performed using an appropriate prior over these functions. If the chosen prior promotes functions with small magnitude, the resulting network will try to solve the learning task with a minimum number of active neurons, because for each active neuron (non-zero output) a penalty is imposed through that prior. If the chosen prior promotes linear functions, the effective depth of the resulting network will be minimized, thus keeping it as linear as possible for solving the given learning task. Since both priors try to keep the complexity of the ANN low and thus also the size of the hypothesis space, they act as regularizers and counter overfitting.

Research on the Dropout regularization method has shown that having neuron outputs that are “noisy” also leads to better generalization performance of the network. A neuron that receives noisy inputs will tend to base its predictions on a evenly spread mixture of these inputs in order to minimize its own output variance (assuming uncorrelated noise). Large weights are disadvantageous for the neuron in that case, since they will lead to a high output variance. The net effect is that the magnitudes of the weight vectors are kept small and each neuron uses redundant inputs for its predictions. This redundancy can be interpreted as an implicit form of model averaging (Hoeting et al., 1999), which is known to counteract overfitting and thus improve the generalization ability of the whole ANN.

Motivated by the observations described in this section, this thesis proposes two new classes of activation functions for neural networks, both with the motivation to make the network architecture more dynamic and learnable while confronting overfitting in a Bayesian setting with appropriate priors.

The first class of activation functions is a continuum between the logarithm, identity and

exponential function in the real and complex domain. It is based on the mathematical theory of fractional functional iteration and fully differentiable both with respect to its input and the iteration parameter, a real number which selects a function within the continuum. The intended application of these functions is to allow the network to learn where to make use of multiplicative interactions and where to stick to purely additive neurons. This is done during standard backpropagation training with no discrete optimization steps necessary.

The second class of activation functions we propose consists of fully flexible, non-parametric functions that can be learned alongside the other parameters of the network. The only constraint on these functions is that they are smooth; thus this is a generalization of the first class. It is clear that this leads to a very expressive model, which must be regularized appropriately to avoid overfitting. We do so by treating the neural network as a probabilistic model and assuming a GP prior over the activation function of each neuron. Training is performed in a fully Bayesian setting by optimizing a variational posterior, which we derive in this work. The proposed model shares some commonalities with deep Gaussian processes but is more economical in terms of model parameters and considerably more efficient to train.

1.3 Outline

This thesis is structured as follows.

Chapter 2 is a summarization of necessary mathematical and technical prerequisites that will be used within this thesis. It contains an introduction to neural networks, Gaussian processes, numeric optimization and elements of probability theory, including sampling algorithms and variational inference.

Chapter 3 introduces a family of activation functions that span a smooth and differentiable continuum between the logarithm, identity and exponential function. Based on the concept of fractional functional iteration, real and complex versions of these families and their derivatives are derived. Experimental results of applying neural networks using these activation functions to real and synthetic datasets are presented.

Chapter 4 describes how to efficiently compute expressions for the derivatives of elementwise defined tensor-valued functions. The development of this method within this thesis was motivated as a high-performance implementation of the stochastic activation function that will be presented in chapter 5. However, since it is generally applicable to any tensor-valued function with applications inside and outside the field of machine learning, we present it independently.

Chapter 5 introduces the Gaussian process neuron, which is a stochastic neuron that uses a non-parametric activation function with a Gaussian process prior. The activation function of each neuron is fully flexible and inferred from the training data. It is discussed how to build probabilistic graphical models resembling feed-forward networks from these units and perform inference using Monte-Carlo methods. An auxiliary parameterization of this model is proposed to enable more efficient training and inference. We describe the route from the auxiliary parameterization to approximate Bayesian inference using a variational posterior. For this purpose the applicability of the central limit theorem to activations of neurons and three different variational approaches are discussed. Preliminary experimental results on regression and classification datasets as well as performance benchmarks are shown.

Chapter 6 provides a summary of the path from non-parametric model to efficient inference and establishes the relation to deep Gaussian processes. It further gives an overview of possible extensions and applications of the proposed model.

Chapter 2

Prerequisites

This chapter is a summary of technical and mathematical concepts required for the original part of this work. It also serves to establish the notations which will be used. It contains an introduction to neural networks, Gaussian Processes, numeric optimization and elements of probability theory, including sampling algorithms and variational inference. It introduces the subset of concepts necessary for this work and provides references to literature for more detailed information.

2.1 Tensor Slicing

In this work the need arises to slice tensors along one or more dimension. A star ($\star$) will be used in place of the index to select all indices of that dimension.

Let us provide a few examples. Given a matrix $X \in \mathbb{R}^{N \times M}$ the notation $X_{i\star} \in \mathbb{R}^{M}$ denotes the i-th row of X. Similarly $X_{\star j} \in \mathbb{R}^{N}$ denotes the j-th column of X.

This can also be extended to tensors and the star can be used multiple times. For example, consider the tensor $A \in \mathbb{R}^{N_1 \times N_2 \times N_3 \times N_4}$. Here $A_{i\star j\star} \in \mathbb{R}^{N_2 \times N_4}$ denotes the matrix that is obtained by fixing the first dimension of A to i and the third dimension to j.

2.2 Probability and Probability Distributions

We will use the notation $\mathrm{P}(X) = f(X)$ to denote that the random variable X is distributed according to the probability density function f. Sometimes we will abbreviate this with the notation $X \sim f$. The notation $\mathrm{P}(X \,|\, Y) = f(X, Y)$ expresses the conditional probability density of X given Y and may be abbreviated using $X \,|\, Y \sim f$.

We will use the notation $\mathrm{E}_{P(X)}[g(X)] = \int P(X)\, g(X)\, \mathrm{d}X$ to denote the expectation value

of g over $\mathrm{P}(X)$. The variance of $g(X)$ under distribution P will be written as $\mathrm{Var}_{\mathrm{P}(X)}(g(X)) = \mathrm{E}_{P(X)}[g(X)^2] - \mathrm{E}_{P(X)}[g(X)]^2$.

Consider three sets of random variables A, B and C. We say that set A is conditionally independent of B given C, if it holds that $\mathrm{P}(A \,|\, B, C) = \mathrm{P}(A \,|\, C)$ for all values of A, B and C. Conditional independence can be written using the notation

$$A \perp\!\!\!\perp B \,|\, C \,.$$

If A is conditionally independent of B given no other variables, i.e. $\mathrm{P}(A \,|\, B) = \mathrm{P}(A)$, we write

$$A \perp\!\!\!\perp B \,|\, \emptyset \,.$$

Note that this *does not* imply that A is still conditionally independent of B if another variable, for example C, becomes observed.

Checking for conditional independence can be done by writing out the joint distribution of A and B given C and checking that it factorizes,

$$\mathrm{P}(A, B \,|\, C) = \mathrm{P}(A \,|\, C)\,\mathrm{P}(B \,|\, C) \,.$$

However, this method quickly becomes cumbersome when more random variables are involved and the structure of the joint distribution is more complicated. A systematic method for checking for conditional independence using probabilistic graphical models will be discussed in section 2.2.10.

2.2.1 Average

Given a matrix $X \in \mathbb{R}^{N \times M}$ the notation

$$\langle X \rangle_{nm} \triangleq \frac{1}{N}\frac{1}{M}\sum_{n=1}^{N}\sum_{m=1}^{M} X_{nm} \tag{2.1}$$

will be used to denote the average over the indices specified in the subscript. Furthermore the notation

$$\langle X \rangle_{n \neq m} \triangleq \frac{1}{N}\frac{1}{M-1}\sum_{n=1}^{N}\sum_{\substack{m=1 \\ m \neq n}}^{M} X_{nm} \tag{2.2}$$

denotes the average over the off-diagonal elements of the matrix.

2.2.2 Jensen's Inequality

Jensen's inequality states that for every convex function $\varphi(x)$ we have

$$\varphi(\mathrm{E}[X]) \leq \mathrm{E}[\varphi(X)]\,. \tag{2.3}$$

Consequently, for a concave function $\psi(x)$, for example the logarithm, we have

$$\psi(\mathrm{E}[X]) \geq \mathrm{E}[\psi(X)]\,. \tag{2.4}$$

2.2.3 Cross Entropy

The cross entropy between two probability distributions P and Q is defined as

$$H(\mathrm{P},\mathrm{Q}) \triangleq \mathrm{E}_{\mathrm{P}(X)}[-\log \mathrm{Q}(X)] = -\int \mathrm{P}(X)\,\log \mathrm{Q}(X)\,\mathrm{d}X\,. \tag{2.5}$$

It measures the average number of nats (natural unit of information) required to encode an event from P when a coding scheme optimal for events distributed according to Q is used.

2.2.4 Kullback-Leibler Divergence

The Kullback-Leibler divergence from distribution $\mathrm{Q}(X)$ to distribution $\mathrm{P}(X)$ is defined as

$$\mathrm{KL}(P\,||\,Q) \triangleq \mathrm{E}_{\mathrm{P}(X)}\left[-\log \frac{\mathrm{Q}(X)}{\mathrm{P}(X)}\right] = -\int \mathrm{P}(X)\,\log \frac{\mathrm{Q}(X)}{\mathrm{P}(X)}\,\mathrm{d}X\,. \tag{2.6}$$

It can be interpreted as the average *additional* number of nats necessary to encode a message X from distribution $\mathrm{Q}(X)$ using a code optimal for distribution $\mathrm{P}(X)$. The Kullback-Leibler divergence is not symmetric with regard to swapping $\mathrm{P}(X)$ and $\mathrm{Q}(X)$. It can be shown that it is always non-negative, $\mathrm{KL}(P\,||\,Q) \geq 0$.

2.2.5 Central Limit Theorems

The family of central limit theorems addresses how the distribution of a sum of random variables behaves in the limit of infinitely many summands. There exist variants of the central limit theorem for sums of independent and dependent random variables.

Lindeberg-Lèvy Central Limit Theorem for Independent and Identical Random Variables

This is the simplest and most widely known version of the central limit theorem. We follow (Georgii, 2015) here. Given an infinite sequence $X_i, i \in \{1, 2, \dots\}$, of independent and identical distributed (iid.) random variables with mean $\mathrm{E}[X_i] = \mu$ and finite variance $\mathrm{Var}(X_i) = \sigma^2$. Then, for the sequence of sample averages

$$S_n \triangleq \frac{1}{n} \sum_{i=1}^{n} X_i \,, \tag{2.7}$$

the following holds

$$\sqrt{n}(S_n - \mu) \xrightarrow{\text{dist}} \mathcal{N}(0, \sigma^2) \,, \tag{2.8}$$

where $\xrightarrow{\text{dist}}$ means "converges in distribution" (Billingsley, 2013). Thus an infinite sum of iid. random variables approaches a normal distribution.

Lyapunov Central Limit Theorem for Independent Random Variables

Given an infinite sequence X_i, $i \in \{1, 2, \dots\}$, of independent random variables each with finite mean $\mathrm{E}[X_i] = \mu_i$ and finite variance $\mathrm{Var}(X_i) = \sigma_i^2$. If for some $\delta > 0$ the condition

$$\lim_{n \to \infty} \frac{1}{s_n^{2+\delta}} \sum_{i=1}^{n} \mathrm{E}\Big[|X_i - \mu_i|^{2+\delta} \Big] = 0 \tag{2.9}$$

with

$$s_n^2 \triangleq \sum_{i=1}^{n} \sigma_i^2 \tag{2.10}$$

is satisfied, then for the sequence of partial sums

$$S_n \triangleq \frac{1}{s_n} \sum_{i=1}^{n} (X_i - \mu_i) \tag{2.11}$$

it holds that $S_n \xrightarrow{\text{dist}} \mathcal{N}(0, 1)$ (Fischer, 2010). Thus it is not necessary for the random variables in a sum to be identically distributed; independence can be enough for the sum to approach a normal distribution.

Central Limit Theorem for Weakly Dependent Random Variables

We follow (Lehmann, 2004) here. If $X_1, \ldots, X_N$ are N *dependent* random variables with $\mathrm{E}[X_i] = \mu$ and $\mathrm{Var}(X_i) = \sigma^2$, then

$$S_N \triangleq \sqrt{N}\left[\left(\frac{1}{N}\sum_{n=1}^{N} X_n\right) - \mu\right] \tag{2.12}$$

in the limit $N \to \infty$ has the distribution

$$P(S_\infty) = \mathcal{N}\left(S_\infty \,\middle|\, 0, \lim_{N\to\infty} \tau_N^2\right) \tag{2.13}$$

with

$$\tau_N^2 = \sigma^2 + \frac{1}{N}\sum_{i\neq j} \mathrm{Cov}(X_i, X_j) \tag{2.14}$$

provided that $\lim_{N\to\infty} \tau_N^2$ is finite. Thus for the central limit theorem to apply the variables must be weakly correlated at most, in the sense that the sum in (2.14) is finite. Since this sum contains $N(N-1) = O(N^2)$ terms, a way to ensure this is to have $\mathrm{Cov}(X_i, X_j) \neq 0$ for at most $O(N)$ number of variable pairs (X_i, X_j).

For random variables $X'_1, \ldots, X'_N$ with inhomogeneous means $\mathrm{E}[X'_i] = \mu_i$ but still $\mathrm{Var}(X'_i) = \sigma^2$, we can set $X_i \triangleq X'_i - \mu_i$ and by applying the above theorem on X_i, we obtain that

$$S'_N \triangleq \frac{1}{\sqrt{N}}\sum_{n=1}^{N} X_n \tag{2.15}$$

in the limit $N \to \infty$ has the distribution

$$P(S'_\infty) = \mathcal{N}\left(S'_\infty \,\middle|\, \lim_{N\to\infty}\sum_{n=1}^{N}\frac{\mu_n}{\sqrt{N}}, \lim_{N\to\infty}\tau_N^2\right). \tag{2.16}$$

2.2.6 Categorical Distribution

The categorical distribution is a probability distribution that describes the possible values of a discrete random variables that can take one out of C possible values. The notation we use for a random variable $Z \in \{1, 2, \ldots, C\}$ that is categorically distributed with probabilities $\boldsymbol{\rho} \in [0, 1]^C$ is

$$Z \sim \mathrm{Cat}(\boldsymbol{\rho})\,. \tag{2.17}$$

This requires that $\sum_i \rho_i = 1$. The corresponding probability mass function is

$$\mathrm{P}(Z \mid \boldsymbol{\rho}) = \mathrm{Cat}(Z \mid \boldsymbol{\rho}) = \prod_i \rho_i^{[Z=i]} \tag{2.18}$$

where $[a = b]$ is the Iverson bracket (Iverson, 1962), defined by

$$[a = b] \triangleq \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} . \tag{2.19}$$

2.2.7 Exponential Family Distributions

An exponential family distribution is a probability distribution that has a density function of the form

$$P_\theta(\boldsymbol{X}) = h(\boldsymbol{X}) \exp(\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \boldsymbol{T}(\boldsymbol{X}) - A(\boldsymbol{\theta})) \tag{2.20}$$

or equivalently

$$P_\theta(\boldsymbol{X}) = \exp[\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \boldsymbol{T}(\boldsymbol{X}) - A(\boldsymbol{\theta}) + B(\boldsymbol{X})] \tag{2.21}$$

which is sometimes called canonical form. Here $h(\boldsymbol{X})$, $\boldsymbol{\eta}(\boldsymbol{\theta})$, $T(\boldsymbol{X})$, $A(\boldsymbol{\theta})$ and $B(\boldsymbol{X})$ are known functions and $\boldsymbol{\eta}(\boldsymbol{\theta}) \cdot \boldsymbol{T}(\boldsymbol{X}) = \sum_i \eta_i(\boldsymbol{\theta}) T_i(\boldsymbol{X})$ denotes the scalar product. The vector $\boldsymbol{\theta}$ is called the parameter vector of the distribution. The function $A(\boldsymbol{\theta})$ is called the log-partition function and is automatically determined once the other functions are specified. It is given by the logarithm of the normalization factor, which must be chosen so that the resulting density is properly normalized.

2.2.8 Univariate Normal Distribution

The normal distribution is parameterized by the mean μ and the variance $\sigma^2 > 0$. The notation we use for a random variable X that is normally distributed is

$$X \sim \mathcal{N}(\mu, \sigma^2) .$$

The corresponding probability density function (PDF) is

$$\mathrm{P}(X \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(X-\mu)^2}{2\sigma^2}\right) \tag{2.22}$$

and thus it is an exponential family distribution.

Product of Univariate Normal PDFs

As described by Bromiley (2003) the product of two Gaussian PDFs in the same random variable is a scaled Gaussian PDF. Given two normal PDFs

$$f_1(X) = \mathcal{N}(X \,|\, \mu_1, \sigma_1^2)$$
$$f_2(X) = \mathcal{N}(X \,|\, \mu_2, \sigma_2^2)$$

we can verify using basic arithmetic that

$$g(X) \triangleq f_1(X)\, f_2(X) = S_g \mathcal{N}(X \,|\, \mu_g, \sigma_g^2) \tag{2.23}$$

with the following parameters

$$\frac{1}{\sigma_g^2} = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2} \tag{2.24a}$$
$$\mu_g = \left(\frac{\mu_1}{\sigma_1^2} + \frac{\mu_2}{\sigma_2^2} \right) \sigma_g^2 \tag{2.24b}$$
$$S_g = \left(2\pi \frac{\sigma_1^2 \sigma_2^2}{\sigma_g^2} \right)^{-\frac{1}{2}} \exp\left(-\frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 \sigma_2^2} \frac{\sigma_g^2}{2} \right) . \tag{2.24c}$$

By induction these formulas can be generalized to the product of an arbitrary number of normal PDFs. For the case of N PDFs with $f_i(X) \triangleq \mathcal{N}(X \,|\, \mu_i, \sigma_i^2)$, $i \in \{1, \ldots, N\}$, the function

$$h(X) \triangleq \prod_{i=1}^{N} f_i(X) = S_h \mathcal{N}(X \,|\, \mu_h, \sigma_h^2) \tag{2.25}$$

has the following parameters

$$\frac{1}{\sigma_h^2} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \tag{2.26a}$$
$$\mu_h = \sigma_h^2 \sum_{i=1}^{N} \frac{\mu_i}{\sigma_i^2} \tag{2.26b}$$
$$S_h = (2\pi)^{-\frac{N-1}{2}} \sqrt{\frac{\sigma_h^2}{\prod_{i=1}^{N} \sigma_i^2}} \exp\left[-\frac{1}{2} \left(\sum_{i=1}^{N} \frac{\mu_i^2}{\sigma_i^2} - \frac{\mu_h^2}{\sigma_h^2} \right) \right] . \tag{2.26c}$$

2.2.9 Multivariate Normal Distribution

The d-dimensional normal distribution is parameterized by the mean vector $\boldsymbol{\mu} \in \mathbb{R}^d$ and the covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ where Σ must be positive-definite. We write

$$\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$

The PDF of the multivariate normal distribution is given by

$$\mathrm{P}(\boldsymbol{X} \mid \boldsymbol{\mu}, \Sigma) = (2\pi)^{-d/2} \left|\Sigma\right|^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{X} - \boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{X} - \boldsymbol{\mu})\right) . \tag{2.27}$$

or as

$$\mathrm{P}(\boldsymbol{X} \mid \boldsymbol{\mu}, \Sigma) = \exp\left(\zeta(\boldsymbol{\mu}, \Sigma) + \boldsymbol{\mu}^T \Sigma^{-1} \boldsymbol{X} - \frac{1}{2} \boldsymbol{X}^T \Sigma^{-1} \boldsymbol{X}\right) \tag{2.28}$$

with the log-partition function

$$\zeta(\boldsymbol{\mu}, \Sigma) = -\frac{1}{2}(d \log 2\pi + \log\left|\Sigma\right| + \boldsymbol{\mu}^T \Sigma^{-1} \boldsymbol{\mu}) .$$

The multivariate normal distribution is a member of the exponential family distributions and can be written in the form of eq. (2.21) using

$$\boldsymbol{\eta}(\boldsymbol{\theta}) = \begin{pmatrix} \Sigma^{-1}\boldsymbol{\mu} \\ -1/2 \operatorname{vec}(\Sigma^{-1}) \end{pmatrix} \tag{2.29a}$$

$$\boldsymbol{T}(\boldsymbol{X}) = \begin{pmatrix} \boldsymbol{x} \\ \operatorname{vec}(\boldsymbol{X}\boldsymbol{X}^T) \end{pmatrix} \tag{2.29b}$$

$$A(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\mu}^T \Sigma^{-1} \boldsymbol{\mu} + \frac{d}{2} \log 2\pi + \frac{1}{2} \log\left|\Sigma\right| \tag{2.29c}$$

$$B(\boldsymbol{x}) = 0 . \tag{2.29d}$$

Here we have used the vectorization operator $\operatorname{vec}(A)$ that transforms a matrix into a vector by concatenating its rows. Given $A \in \mathbb{R}^{N \times M}$ applying $\boldsymbol{a} = \operatorname{vec}(A)$ results in $\boldsymbol{a} \in \mathbb{R}^{NM}$ with $a_{nM+m} = A_{n,m}$.

Marginal Normal Distribution

Given a normally distributed random vector $\boldsymbol{X} \in \mathbb{R}^d$ with $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, the random vector $\boldsymbol{X'} \in \mathbb{R}^{d-1}$ defined by removing element r from $\boldsymbol{X}$,

$$\boldsymbol{X'} \triangleq (X_1, \ldots, X_{r-1}, X_{r+1}, \ldots, X_d)^T , \tag{2.30}$$

is normally distributed $\boldsymbol{X'} \sim \mathcal{N}(\boldsymbol{\mu'}, \Sigma')$ with

$$\boldsymbol{\mu'} = (\mu_1, \ldots, \mu_{r-1}, \mu_{r+1}, \ldots, \mu_d)^T , \tag{2.31a}$$

$$\Sigma' = \begin{pmatrix} \Sigma_{1,1} & \cdots & \Sigma_{1,r-1} & \Sigma_{1,r+1} & \cdots & \Sigma_{1,d} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \Sigma_{r-1,1} & \cdots & \Sigma_{r-1,r-1} & \Sigma_{r-1,r+1} & \cdots & \Sigma_{r-1,d} \\ \Sigma_{r+1,1} & \cdots & \Sigma_{r+1,r-1} & \Sigma_{r+1,r+1} & \cdots & \Sigma_{r+1,d} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \Sigma_{d,1} & \cdots & \Sigma_{d,r-1} & \Sigma_{d,r+1} & \cdots & \Sigma_{d,d} \end{pmatrix} . \tag{2.31b}$$

Thus the marginal distribution

$$\mathrm{P}(\boldsymbol{X'}) = \int \mathrm{P}(\boldsymbol{X}) \, \mathrm{d}X_r$$

is obtained by removing all entries from the mean vector and covariance matrix that correspond to the dimension that is marginalized out. The multivariate normal distribution is closed under marginalization and the process can be iterated to marginalize out more than one dimension.

Conditional Normal Distribution

Let the random block vector

$$\boldsymbol{X} \triangleq \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \end{bmatrix} \tag{2.32}$$

be normally distributed, $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, with mean and variance given by the block matrices

$$\boldsymbol{\mu} \triangleq \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix} , \qquad \Sigma \triangleq \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} . \tag{2.33}$$

Then the conditional is normally distributed, $\boldsymbol{X}_1 \,|\, \boldsymbol{X}_2 \sim \mathcal{N}(\overline{\boldsymbol{\mu}}_1, \overline{\Sigma}_1)$, with

$$\overline{\boldsymbol{\mu}}_1 = \boldsymbol{\mu}_1 + \Sigma_{12} \, \Sigma_{22}^{-1} (\boldsymbol{X}_2 - \boldsymbol{\mu}_2) \, , \tag{2.34a}$$

$$\overline{\Sigma}_1 = \Sigma_{11} - \Sigma_{12} \, \Sigma_{22}^{-1} \, \Sigma_{21} \, . \tag{2.34b}$$

Note that the conditional covariance $\overline{\Sigma}_1$ does not depend on the observed values. The multivariate normal distributed is closed under conditioning on a subset of its variables.

KL-divergence between Two Normal Distributions

The Kullback-Leibler divergence (2.6) can be evaluated explicitly for two multivariate normal distributions. Let $\mathrm{P}(\boldsymbol{X}) \triangleq \mathcal{N}(\boldsymbol{X} \mid \boldsymbol{\mu}_p, \Sigma_p)$ and $\mathrm{Q}(\boldsymbol{X}) \triangleq \mathcal{N}(\boldsymbol{X} \mid \boldsymbol{\mu}_q, \Sigma_q)$ be two distribution over $\boldsymbol{X} \in \mathbb{R}^d$. Then the Kullback-Leibler divergence from $\mathrm{Q}(\boldsymbol{X})$ to $\mathrm{P}(\boldsymbol{X})$ is

$$\mathrm{KL}(P \,||\, Q) = \frac{1}{2}\left(\mathrm{tr}(\Sigma_q^{-1}\,\Sigma_p) + (\boldsymbol{\mu}_q - \boldsymbol{\mu}_p)^T\,\Sigma_q^{-1}\,(\boldsymbol{\mu}_q - \boldsymbol{\mu}_p) - d + \log\frac{|\Sigma_q|}{|\Sigma_p|}\right), \tag{2.35}$$

where $|\bullet|$ denotes the determinant of a matrix and $\mathrm{tr}\,\bullet$ is its trace.

Affine Transformation

Given an affine function $\boldsymbol{Y} : \mathbb{R}^d \to \mathbb{R}^b$ with

$$\boldsymbol{Y}(\boldsymbol{X}) \triangleq \boldsymbol{c} + B\boldsymbol{X}$$

where $\boldsymbol{c} \in \mathbb{R}^b$ and $B \in \mathbb{R}^{b\times d}$, the distribution of $\boldsymbol{Y}$ is also multivariate normal with

$$\boldsymbol{Y} \sim \mathcal{N}(\boldsymbol{c} + B\boldsymbol{\mu}, B\Sigma B^T)\,. \tag{2.36}$$

Thus the multivariate normal distribution is closed under affine transformations. Furthermore, the standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbb{1})$ can be transformed into a normal distribution of any mean and covariance by means of an affine transformation of the random variable.

Product of Multivariate Normal PDFs

The product of multiple multivariate normal PDFs in the same random variable is a scaled multivariate normal PDFs. Given N multivariate normal PDF,

$$f_i(\boldsymbol{X}) = \mathcal{N}(\boldsymbol{X} \mid \boldsymbol{\mu}_i, \Sigma_i)$$

in d-dimensional space, their product

$$h(\boldsymbol{X}) \triangleq \prod_{i=1}^{N} f_i(\boldsymbol{X}) = S_h\,\mathcal{N}(\boldsymbol{X} \mid \boldsymbol{\mu}_h, \Sigma_h) \tag{2.37}$$

has the following parameters

$$\Sigma_h^{-1} = \sum_{i=1}^{N} \Sigma_i^{-1} \tag{2.38a}$$

$$\boldsymbol{\mu}_h = \Sigma_h \sum_{i=1}^{N} (\Sigma_i)^{-1} \boldsymbol{\mu}_i \tag{2.38b}$$

$$S_h = \exp\left[\frac{1}{2}\left((1-N)d\log 2\pi + \log|\Sigma_h| - \sum_{i=1}^{N}\log|\Sigma_i| + \boldsymbol{\mu}_h^T\Sigma_h^{-1}\boldsymbol{\mu}_h - \sum_{i=1}^{N}\boldsymbol{\mu}_i^T\Sigma_i^{-1}\boldsymbol{\mu}_i\right)\right] \tag{2.38c}$$

as can be seen by writing the PDFs in form (2.28) and summing the terms inside the exponential function. A detailed derivation is presented in (Bromiley, 2003).

2.2.10 Probabilistic Graphical Models

Probabilistic graphical models (Lauritzen, 1996) define a factorization of a probability distribution using a directed, acyclic graph. For example, the factorization of the joint distribution $\mathrm{P}(X_1, X_2, X_3, X_4, X_5, X_6, X_7)$ into

$$\begin{aligned}\mathrm{P}(X_1, X_2, X_3, X_4, X_5, X_6, X_7) = &\mathrm{P}(X_1)\,\mathrm{P}(X_2)\,\mathrm{P}(X_3)\,P(X_4 \,|\, X_1, X_2, X_3)\,\mathrm{P}(X_5 \,|\, X_1, X_3)\cdot \\ &\mathrm{P}(X_6 \,|\, X_4)\,\mathrm{P}(X_7 \,|\, X_4, X_5)\end{aligned} \tag{2.39}$$

corresponds to the probabilistic graphical model shown in fig. 2.1a. Distributions for the factors $\mathrm{P}(X_1)$, $\mathrm{P}(X_2)$, $\mathrm{P}(X_3)$, $P(X_4 \,|\, X_1, X_2, X_3)$, $\mathrm{P}(X_5 \,|\, X_1, X_3)$, $\mathrm{P}(X_6 \,|\, X_4)$ and $\mathrm{P}(X_7 \,|\, X_4, X_5)$ must be specified separately.

In general the joint distribution of a graphical model over random variables $\boldsymbol{X}$ is specified by

$$\mathrm{P}(\boldsymbol{X}) = \prod_k \mathrm{P}(X_k \,|\, \mathrm{parents}(X_k)) \tag{2.40}$$

where $\mathrm{parents}(X_k)$ denotes the set of parents of node X_k in the directed, acyclic graph representing the graphical model. Each random variable can be either discrete or continuous. In graphical models conditional distributions are directly available when all parents of a particular node are observed. To obtain marginal distributions for a node in the graph, all its ancestors have to be marginalized out.

Probabilistic graphical models offer a method to directly check for conditional independence in the corresponding directed, acyclic graph without having to write down the joint distribution represented by the graph. To check whether $A \perp\!\!\!\perp B \,|\, C$, where A, B and C are non-intersecting

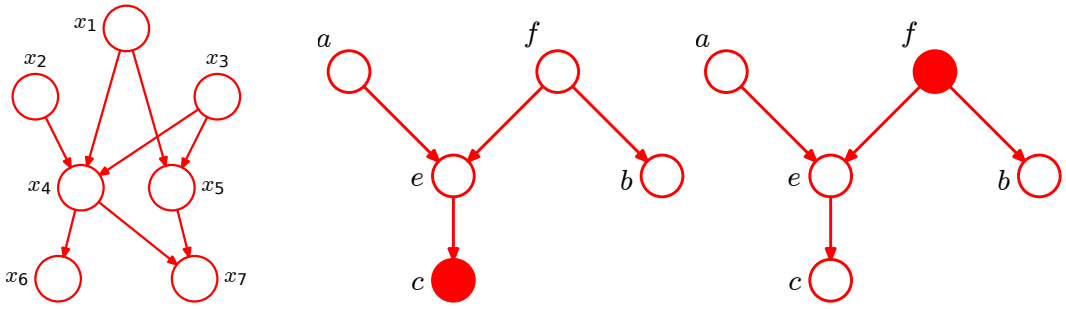


Figure 2.1: Examples for probabilistic graphical models from (Bishop, 2006). Observed variables are shown as filled nodes. (a) This corresponds to the factorization of the probability distribution in eq. (2.39). (b) Example for no d-separation between a and b by c. The path from a to b is not blocked by f because the arrows meet tail-to-tail and f is not observed. The path is neither blocked by e because the arrows meet head-to-head and its descendant c is observed. Thus $a \perp\!\!\!\perp b \,|\, c$ does not follow from this graphical model. (c) Example for d-separation between a and b by f. The path from a to b is blocked by f because the arrows meet tail-to-tail and f is observed. Thus this graphical model implies $a \perp\!\!\!\perp b \,|\, f$.

sets of random variables, holds in a graphical model perform the following routine. Consider *all* possible path from any node in set A to any node in set B. Any such path is said to be blocked if it includes a node such that

- the arrows on the path meet either head-to-tail or tail-to-tail at the node, and the meeting node is in the observed set C, *or*
- the arrows meet head-to-head at the node, and neither the meeting node, nor any of its descendants, are in the observed set C. A descendant is any node that can be reached by following the arrows in tail-to-head direction.

If *all paths* between A and B are blocked, then A is said to be d-separated from B by C and it holds that $A \perp\!\!\!\perp B \,|\, C$. A proof is given in (Lauritzen, 1996). Two examples are shown in fig. 2.1b and fig. 2.1c.

2.2.11 The Unscented Transform

The unscented transform (Julier et al., 1996, 1997) is a method for calculating the statistics of a random variable that undergoes a transformation by a non-linear function. It works by approximating a multivariate normal distribution by a finite and *deterministic* set of points that are called sigma points and have the same mean and covariance as the distribution. The sigma points are then propagated through the non-linear function and used to estimate the moments of the transformed distribution. No Taylor expansion of the non-linear function is necessary.

Let $\boldsymbol{X} \in \mathbb{R}^d$ be a d-dimensional random variable with distribution

$$\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}^{\boldsymbol{X}}, \Sigma^{\boldsymbol{X}})$$

and let $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^c$ be an arbitrary function. The distribution of the random variable $\boldsymbol{Y} \in \mathbb{R}^c$ is determined by $\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{X})$. Consider the $2d+1$ sigma points $\boldsymbol{X}_\star$ and the associated weights $W_\star$, which are given by

$$\boldsymbol{X}_0 \triangleq \boldsymbol{\mu}^{\boldsymbol{X}}\,, \qquad W_0 \triangleq \frac{\kappa}{d+\kappa}\,, \tag{2.41a}$$

$$\boldsymbol{X}_i \triangleq \boldsymbol{\mu}^{\boldsymbol{X}} + \mathcal{S}_{i\star}\,, \qquad W_i \triangleq \frac{1}{2\,(d+\kappa)}\,, \quad i \in \{1,\dots,d\}\,, \tag{2.41b}$$

$$\boldsymbol{X}_{i+d} \triangleq \boldsymbol{\mu}^{\boldsymbol{X}} - \mathcal{S}_{i\star}\,, \qquad W_{i+d} \triangleq \frac{1}{2\,(d+\kappa)}\,, \quad i \in \{1,\dots,d\}\,, \tag{2.41c}$$

where

$$\mathcal{S} \triangleq \sqrt{(d+\kappa)\Sigma^{\boldsymbol{X}}} \tag{2.42}$$

denotes the matrix square root of the scaled covariance matrix; it can be calculated using an eigendecomposition of $\Sigma^{\boldsymbol{X}}$. The parameter $\kappa \in \mathbb{R}$ controls the spread of the sigma points and can be chosen freely; a standard heuristic is to set $\kappa = 3 - d$. It can easily be verified that the weighted average of the sigma points equals the mean,

$$\boldsymbol{\mu}^{\boldsymbol{X}} = \sum_{i=0}^{2d} W_i\, \boldsymbol{X}_i\,, \tag{2.43}$$

and the weighted outer product equals the covariance,

$$\boldsymbol{\Sigma}^{\boldsymbol{X}} = \sum_{i=0}^{2d} W_i\, (\boldsymbol{X}_i - \boldsymbol{\mu}^{\boldsymbol{X}})\, (\boldsymbol{X}_i - \boldsymbol{\mu}^{\boldsymbol{X}})^T\,. \tag{2.44}$$

Thus the sigma points can be taught of as samples from the distribution of $\boldsymbol{x}$ that capture its mean and covariance exactly, i.e. the clever choice of points eliminates the statistical variance that is usually induced by having a limited number of samples.

The mean and covariance of $\boldsymbol{Y}$ are obtained by transforming the sigma points though $\boldsymbol{f}$ and calculating the above statistics using $\boldsymbol{f}(\boldsymbol{X}_i)$. Thus the mean of $\boldsymbol{y}$ is given by the weighted average of the transformed points,

$$\boldsymbol{\mu}^{\boldsymbol{Y}} = \sum_{i=0}^{2d} W_i\, \boldsymbol{f}(\boldsymbol{X}_i)\,, \tag{2.45}$$

and the covariance of $\boldsymbol{y}$ is given by the weighted outer product,

$$\boldsymbol{\Sigma}^{\boldsymbol{Y}} = \sum_{i=0}^{2d} W_i \left(\boldsymbol{f}(\boldsymbol{X}_i) - \boldsymbol{\mu}^{\boldsymbol{Y}}\right)\left(\boldsymbol{f}(\boldsymbol{X}_i) - \boldsymbol{\mu}^{\boldsymbol{Y}}\right)^T . \tag{2.46}$$

It can be shown that the estimates provided by the unscented transform are significantly better than when linearizing $\boldsymbol{f}(\boldsymbol{X})$ around the mean of the distribution and using the affine transformation property of the normal distribution to calculate the transformed statistics.

Alternatively the Cholesky decomposition, see below, can be used instead of the matrix square root in (2.42). To do so we set

$$\mathcal{S} \triangleq \mathrm{chol}[(d+\kappa)\Sigma^{\boldsymbol{X}}] \tag{2.47}$$

and notice that eqs. (2.43) and (2.44) still hold for the sigma points obtained by doing so; thus they exactly capture the mean and covariance of the distribution as before. The mean and covariance of the transformed distribution are calculated as above using eqs. (2.45) and (2.46). Using the Cholesky decomposition has the advantage that its derivative is available, see eq. (2.48), and thus $\boldsymbol{\mu}^{\boldsymbol{y}}$ and $\boldsymbol{\Sigma}^{\boldsymbol{y}}$ can be differentiated w.r.t. parameters occurring in $\boldsymbol{f}$ by application of the chain rule.

Cholesky Decomposition

The Cholesky decomposition of a symmetric, positive-definite matrix $A \in \mathbb{R}^{d\times d}$ is the *lower triangular* matrix $\mathrm{chol}(A) \in \mathbb{R}^{d\times d}$, such that

$$A = \mathrm{chol}(A)\,\mathrm{chol}(A)^T .$$

It can be shown that the Cholesky decomposition is unique and the elements on the diagonal of $\mathrm{chol}(A)$ are positive, $\mathrm{chol}(A)_{ii} \geq 0$. The Cholesky decomposition can be computed using a modified form of the Gaussian elimination algorithm (Press et al., 1992).

Following (Murray, 2016) the Jacobian of the Cholesky decomposition is given by

$$\frac{\partial L_{ij}}{\partial A_{kl}} = \left(\sum_{m=j+1}^{d} L_{im}\, L^{-1}_{mk} + \frac{1}{2} L_{ij}\, L^{-1}_{jk}\right) L^{-1}_{jl} + (1-\delta_{kl}) \left(\sum_{m=j+1}^{d} L_{im}\, L^{-1}_{ml} + \frac{1}{2} L_{ij}\, L^{-1}_{jl}\right) L^{-1}_{jk} \tag{2.48}$$

with $L \triangleq \mathrm{chol}(A)$ and δ_{kl} is the Kronecker delta with

$$\delta_{kl} \triangleq \begin{cases} 1 & \text{if } k = l \\ 0 & \text{otherwise} \end{cases} . \tag{2.49}$$

However, it is more efficient to adapt the Cholesky decomposition algorithm for numerical computation of the derivative using automatic differentiation as described in (S. P. Smith, 1995) than to employ the above formula.

2.2.12 Variational Inference

Consider a model that consists of two sets of random variables, $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ and $\boldsymbol{Z} = \{Z_1, \ldots, Z_N\}$. Furthermore the joint distribution of these variables,

$$\mathrm{P}(\boldsymbol{X}, \boldsymbol{Z}) = \mathrm{P}(\boldsymbol{X} \mid \boldsymbol{Z})\,\mathrm{P}(\boldsymbol{Z})\,, \tag{2.50}$$

is specified. The random variables $\boldsymbol{Z}$ can be thought of as latent variables with a prior distribution $\mathrm{P}(\boldsymbol{Z})$. The variables $\boldsymbol{X}$ belong to the part of the model that can be observed. Their distribution $\mathrm{P}(\boldsymbol{X})$ could be obtained by marginalizing out $\boldsymbol{Z}$ in (2.50), which is, however, intractable in many cases.

Given observed values for $\boldsymbol{X}$ we want to perform Bayesian inference of the latent variables, i.e. calculate the posterior distribution

$$\mathrm{P}(\boldsymbol{Z} \mid \boldsymbol{X}) = \frac{\mathrm{P}(\boldsymbol{X} \mid \boldsymbol{Z})\,\mathrm{P}(\boldsymbol{Z})}{\mathrm{P}(\boldsymbol{X})}\,, \tag{2.51}$$

which is intractable due the occurrence of $\mathrm{P}(\boldsymbol{X})$. The technique of variational inference (Bishop, 2006; Fox et al., 2012) finds an approximative distribution $\mathrm{Q}(\boldsymbol{Z}) \approx \mathrm{P}(\boldsymbol{Z} \mid \boldsymbol{X})$ for the posterior instead. Variational inference uses the KL-divergence to measure how closely $\mathrm{Q}(\boldsymbol{Z})$ resembles $\mathrm{P}(\boldsymbol{Z})$, thus the best approximative distribution $\mathrm{Q}^*(\boldsymbol{Z})$ is given by

$$\mathrm{Q}^*(\boldsymbol{Z}) = \arg\min_{\mathrm{Q}} \mathrm{KL}\big(\mathrm{Q}(\boldsymbol{Z}) \,||\, \mathrm{P}(\boldsymbol{Z} \mid \boldsymbol{X})\big)\,.$$

Expanding the KL-divergence and rewriting the posterior gives

$$\begin{aligned}\mathrm{KL}\big(\mathrm{Q}(\boldsymbol{Z}) \,||\, \mathrm{P}(\boldsymbol{Z} \,|\, \boldsymbol{X})\big) &= \int \mathrm{Q}(\boldsymbol{Z}) \log \frac{\mathrm{Q}(\boldsymbol{Z})}{\mathrm{P}(\boldsymbol{Z} \,|\, \boldsymbol{X})} \,\mathrm{d}\boldsymbol{Z} \\ &= \int \mathrm{Q}(\boldsymbol{Z}) \log \frac{\mathrm{Q}(\boldsymbol{Z})\,\mathrm{P}(\boldsymbol{X})}{\mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})} \,\mathrm{d}\boldsymbol{Z} \\ &= \int \mathrm{Q}(\boldsymbol{Z}) \log \frac{\mathrm{Q}(\boldsymbol{Z})}{\mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})} \,\mathrm{d}\boldsymbol{Z} + \log \mathrm{P}(\boldsymbol{X})\,.\end{aligned}$$

Reordering gives

$$\log \mathrm{P}(\boldsymbol{X}) = \mathcal{L}(\mathrm{Q}) + \mathrm{KL}\big(\mathrm{Q}(\boldsymbol{Z}) \,||\, \mathrm{P}(\boldsymbol{Z} \,|\, \boldsymbol{X})\big) \tag{2.52}$$

where

$$\mathcal{L}(\mathrm{Q}) \triangleq \int \mathrm{Q}(\boldsymbol{Z}) \log \frac{\mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})}{\mathrm{Q}(\boldsymbol{Z})} \,\mathrm{d}\boldsymbol{Z} \tag{2.53}$$

is called the evidence lower bound (ELBO). Since $\log \mathrm{P}(\boldsymbol{X})$ is constant with respect to $Q(\boldsymbol{Z})$ and a KL-divergence is non-negative, the KL-divergence $\mathrm{KL}\big(\mathrm{Q}(\boldsymbol{Z}) \,||\, \mathrm{P}(\boldsymbol{Z} \,|\, \boldsymbol{X})\big)$ is minimized by maximizing $\mathcal{L}(\mathrm{Q})$. Furthermore, this implies

$$\log \mathrm{P}(\boldsymbol{X}) \geq \mathcal{L}(\mathrm{Q})$$

and thus we obtain a lower bound for the log model evidence by maximizing $\mathcal{L}(\mathrm{Q})$. Maximizing $\mathcal{L}(\mathrm{Q})$ is a much easier task, because usually the joint distribution $\mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})$ is available and tractable while the posterior is not. There are a variety of methods to perform the maximization of $\mathcal{L}(\mathrm{Q})$. We will briefly describe two of them.

Mean-field Approach

The mean-field approach factors $\mathrm{Q}(\boldsymbol{Z})$ into independent partitions of random variables $\boldsymbol{Z}_1$, $\boldsymbol{Z}_2$, $\ldots$, $\boldsymbol{Z}_M$, so that

$$\mathrm{Q}(\boldsymbol{Z}) = \prod_{p=1}^{M} \mathrm{Q}_p(\boldsymbol{Z}_p)\,. \tag{2.54}$$

It can be shown that $\mathcal{L}(\mathrm{Q})$ is maximized by iteratively setting

$$\mathrm{Q}_j(\boldsymbol{Z}_j) = \frac{\exp\big(\mathrm{E}_{i\neq j}[\log \mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})]\big)}{\int \exp\big(\mathrm{E}_{i\neq j}[\log \mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})]\big)\,\mathrm{d}\boldsymbol{Z}_j} \tag{2.55}$$

until convergence, where the occurring expectation is defined as

$$\mathrm{E}_{i\neq j}[\log \mathrm{P}(\boldsymbol{X}, \boldsymbol{Z})] \triangleq \int \log \mathrm{P}(\boldsymbol{X}, \boldsymbol{Z}) \prod_{\substack{i=1 \\ i\neq j}}^{M} \mathrm{Q}_p(\boldsymbol{Z}_p)\, \mathrm{d}\boldsymbol{Z}_p \,.$$

In this approach the only aspect that must be specified by the user is the factorization of $\mathrm{Q}(\boldsymbol{Z})$ in (2.54). The functional forms of the distribution factors $\mathrm{Q}_p(\boldsymbol{Z}_p)$ follow automatically from (2.55). However, care must be taken to choose the factorization so that the occurring expectations remain analytically tractable, which can be challenging in practice.

Parameterized Approximative Distribution

Another approach is to choose a parameterized distribution for $\mathrm{Q}_\theta(\boldsymbol{Z})$ and perform the maximization of $\mathcal{L}(\mathrm{Q}_\theta)$ numerically. For that purpose the derivatives $\partial \mathrm{Q}_\theta / \partial \theta$ are calculated and a gradient-based method is used optimize θ step by step. The advantage of this approach is that it works even for problems where the expectations in (2.55) are intractable. However, it is prone to getting stuck in local minima and an unsuitable choice of the functional form of $\mathrm{Q}_\theta(\boldsymbol{Z})$ will result in a poor approximation. Nevertheless, this approach has seen widespread adoption lately in the context of variational autoencoders (D. P. Kingma et al., 2013). In that instance $\mathrm{Q}_\theta(\boldsymbol{Z})$ was modeled as a neural network.

2.2.13 Markov Chain Monte Carlo Sampling

Markov chain Monte Carlo (MCMC) sampling is a class of methods for obtaining samples from probability distributions when the PDF is not available or prohibitively expensive to compute. All MCMC methods have in common that a Markov chain is formed to obtain samples from the distribution. These methods differ in the speed of convergence of the aforementioned Markov chain and the requirements on the availability of PDFs of distributions related to the distribution to be approximated.

Metropolis-Hastings Sampling

The Metropolis-Hastings algorithm (Metropolis et al., 1953) only requires a function that is proportional to the PDF of the target distribution and thus it can be used to sample from unnormalized probability densities. Let $\boldsymbol{X} = (X_1, X_2, \ldots, X_D) \in \mathbb{R}^D$ be a vector of random variables with density function $\mathrm{P}(\boldsymbol{X}) \propto f(\boldsymbol{X})$. Furthermore let $\mathrm{Q}(\boldsymbol{X}^* \,|\, \boldsymbol{X})$ be a distribution over $\boldsymbol{X}^* = (X_1^*, X_2^*, \ldots, X_D^*) \in \mathbb{R}^D$ that can be efficiently sampled from. A Markov chain $\boldsymbol{X}^{(s)}$,

Algorithm 1: Metropolis-Hastings sampling

Input: unnormalized density $f(\boldsymbol{X}) \propto \mathrm{P}(\boldsymbol{X})$; proposal distribution $\mathrm{Q}(\boldsymbol{X}^* \mid \boldsymbol{X})$
Output: Markov chain $\boldsymbol{X}^{(s)}$, $s \in \{0, 1, 2, \dots\}$, with stationary distribution $P(\boldsymbol{X})$

$\boldsymbol{X}^{(0)} \longleftarrow$ random state `// start from random initial state`
$s \longleftarrow 1$
while *true* **do**
  Sample $\boldsymbol{X}^* \sim \mathrm{Q}(\boldsymbol{X}^* \mid \boldsymbol{X}^{(s-1)})$ `// obtain proposal state`
  $\rho \longleftarrow \min\left(1, \dfrac{f(\boldsymbol{X}^*)}{f(\boldsymbol{X}^{(s-1)})} \dfrac{\mathrm{Q}(\boldsymbol{X}^{(s-1)} \mid \boldsymbol{X}^*)}{\mathrm{Q}(\boldsymbol{X}^* \mid \boldsymbol{X}^{(s-1)})}\right)$ `// calculate acceptance prob.`
  Sample $a \sim \begin{cases} \text{true} & \text{with probability } \rho \\ \text{false} & \text{with probability } 1-\rho \end{cases}$ `// accept or reject proposal`
  if a **then**
    $\boldsymbol{X}^{(s)} \longleftarrow \boldsymbol{X}^*$
    $s \longleftarrow s + 1$

$s \in \{0, 1, 2, \dots\}$, that has stationary distribution $\mathrm{P}(\boldsymbol{X})$ can then be obtained by algorithm 1. It works by proposing state changes $\boldsymbol{X}^*$ and accepting them with a probability ρ that is chosen so that the resulting Markov chain will converge to $\mathrm{P}(\boldsymbol{X})$. Since ρ is calculated from *ratios* of probability densities only, the algorithm can accept an unnormalized PDF.

It is necessary to run the Markov chain for a number of iterations before samples from it are distributed according to the stationary distribution $\mathrm{P}(\boldsymbol{X})$. The necessary number of iterations for that purpose is referred to as the burn-in period S_{burnin}. Also, nearby samples from the Markov chain will not be independent samples from $\mathrm{P}(\boldsymbol{X})$ but have autocorrelation. If this is not desired, the Markov chain must be run for S_{cor} intermediate steps between taking samples, where S_{cor} should be of the order of the autocorrelation time of the Markov process.

The rate with which we can obtain uncorrelated samples from the Metropolis-Hastings algorithm depends mainly on the choice of the proposal distribution $\mathrm{Q}(\boldsymbol{X}^* \mid \boldsymbol{X})$. If proposed values are too dissimilar from the current state, then the acceptance probability will be low, resulting in many trails before a new state is accepted and thus sample generation will be slow. This problem is intensified in high-dimensional spaces, since with increasing number of variables it becomes less likely that a sample from a proposal distribution moves the Markov chain into a region of higher probability. Thus, in high-dimensional spaces commonly a proposal distribution is used that only changes the state in a single dimension by a small values. This leads to an acceptable acceptance probability, but the autocorrelation between adjacent states of the Markov chain will be high, making it necessary to run it for a large number of intermediate steps between consuming samples to ensure that they are uncorrelated.

Algorithm 2: Gibbs sampling

Input: conditional distributions $\mathrm{P}(\boldsymbol{X}_p \,|\, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_{p-1}, \boldsymbol{X}_{p+1}, \ldots, \boldsymbol{X}_P)$, $p \in \{1, 2, \ldots P\}$, that can be sampled from

Output: Markov chain $\boldsymbol{X}^{(s)}$, $s \in \{0, 1, 2, \ldots\}$, with stationary distribution $P(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_P)$

$\boldsymbol{X}^{(0)} \longleftarrow$ random state `// start from random initial state`

for $s \in \{1, 2, \ldots\}$ **do**

  for $p \in \{1, 2, \ldots, P\}$ **do**

    $\boldsymbol{X}_p^{(s)} \longleftarrow$ sample from $\mathrm{P}(\boldsymbol{X}_p^{(s)} \,|\, \boldsymbol{X}_1^{(s)}, \ldots, \boldsymbol{X}_{p-1}^{(s)}, \boldsymbol{X}_{p+1}^{(s-1)}, \ldots, \boldsymbol{X}_P^{(s-1)})$

Gibbs Sampling

Gibbs Sampling (Casella et al., 1992) requires that conditional sampling of partitions of the random variables is efficiently possible. For example, if sampling from the target distribution $\mathrm{P}(X_1, X_2, X_3)$ is not directly possible, but sampling from the conditionals $\mathrm{P}(X_1 \,|\, X_2, X_3)$, $\mathrm{P}(X_2 \,|\, X_1, X_3)$ and $\mathrm{P}(X_3 \,|\, X_1, X_2)$ is possible, then Gibbs Sampling can be applied to obtain samples from that target distribution.

Let the random variables $\boldsymbol{X}$ be split into P partitions, each denoted by $\boldsymbol{X}_p$. Further assume that sampling from the conditionals

$$\mathrm{P}(\boldsymbol{X}_p \,|\, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_{p-1}, \boldsymbol{X}_{p+1}, \ldots, \boldsymbol{X}_P), \quad p \in \{1, 2, \ldots P\}, \tag{2.56}$$

is possible. Then algorithm 2 generate a Markov chain with stationary distribution $P(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_P)$. Provided that all conditional distribution in (2.56) have non-zero probabilities, this Markov chain is ergodic and will thus eventually converge to the target distribution.

Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo (HMC) algorithm (Duane et al., 1987) is a variant of the Metropolis-Hastings sampling algorithm that employs a proposal distribution tailored to the distribution to sample from and thus obtains a high acceptance rate in high-dimensional state spaces while still proposing samples that are mostly uncorrelated. It is inspired by the Hamiltonian function (Arnol'd, 2013), which measures the total energy of a physical system, consisting of kinetic and potential energy. By the laws of physics, energy is conserved in a closed system and thus the value of the Hamiltonian function is constant. This idea is translated into a sampling algorithm by treating the unnormalized PDF as potential energy and introducing auxiliary variables that represent impulses and thus contribute the kinetic energy. Movement is then

simulated using the Hamiltonian's differential equations and samples from $\mathrm{P}(\boldsymbol{X})$ are proposed using snapshots of this system.

For each variable X_i of the distribution $\mathrm{P}(\boldsymbol{X})$ HMC introduces an auxiliary variable ρ_i. These variables are referred to as impulses. The joint distribution $\mathrm{P}(\boldsymbol{X}, \boldsymbol{\rho})$ is given by

$$\mathrm{P}(\boldsymbol{X}, \boldsymbol{\rho}) = \frac{1}{Z} \exp\left(-\frac{H(\boldsymbol{X}, \boldsymbol{\rho})}{T}\right) \tag{2.57}$$

where

$$H(\boldsymbol{X}, \boldsymbol{\rho}) \triangleq U(\boldsymbol{X}) + K(\boldsymbol{\rho}) \tag{2.58}$$

is the so-called Hamiltonian, $T > 0$ is the temperature of the system and Z is the partition function, so that the joint probability is properly normalized. $K(\boldsymbol{\rho})$ is called the kinetic energy of the system and is given by

$$K(\boldsymbol{\rho}) \triangleq \frac{1}{2} \boldsymbol{\rho}^T \mathrm{diag}(\boldsymbol{m})^{-1} \boldsymbol{\rho} \tag{2.59}$$

where $m_i > 0$ is called the mass of variable X_i. The so-called potential energy $U(\boldsymbol{X})$ is determined up to a constant by the distribution we wish to sample from,

$$U(\boldsymbol{X}) \triangleq -\log \mathrm{P}(\boldsymbol{X}) + \mathrm{const} \tag{2.60}$$

and thus an unnormalized density $\mathrm{P}(\boldsymbol{X})$ can be used.

HMC uses Gibbs sampling to alternatively sample the state $\boldsymbol{X}$ and impulse $\boldsymbol{\rho}$ from $\mathrm{P}(\boldsymbol{X}, \boldsymbol{\rho})$. Since $\boldsymbol{X}$ and $\boldsymbol{\rho}$ are *independent* random variables under (2.57), we have

$$\mathrm{P}(\boldsymbol{\rho} \,|\, \boldsymbol{X}) = \mathrm{P}(\boldsymbol{\rho}) = \mathcal{N}(\boldsymbol{\rho} \,|\, \boldsymbol{0}, \mathrm{diag}(\boldsymbol{m}))$$

and can easily sample $\boldsymbol{\rho}$. However, we cannot sample from $\mathrm{P}(\boldsymbol{X} \,|\, \boldsymbol{\rho}) = \mathrm{P}(\boldsymbol{X})$ directly, thus here HMC uses the Metropolis-Hastings sampling method (algorithm 1) to sample from that conditional. The employed proposal distribution Q is *deterministic*. The proposal state $\{\boldsymbol{X}^*, \boldsymbol{\rho}^*\}$, where $\boldsymbol{\rho}^*$ stands for the proposed impulse, is computed as follows: Start at the state $\boldsymbol{X}(t = 0) \triangleq \boldsymbol{X}$ and $\boldsymbol{\rho}(t = 0) \triangleq \boldsymbol{\rho}$ and evaluate it according to Hamilton's differential equations

$$\frac{\mathrm{d}X_d}{\mathrm{d}t} = \frac{\partial H}{\partial \rho_d}, \qquad \frac{\mathrm{d}\rho_d}{\mathrm{d}t} = -\frac{\partial H}{\partial X_d}, \qquad d \in \{1, 2, \ldots, D\} \tag{2.61}$$

for a fixed period of time T. This can be done by either finding an analytic solution of these differential equations or using a numerical simulation method. The proposed state is given by $\boldsymbol{X}^* \triangleq \boldsymbol{X}(t = T)$ and $\boldsymbol{\rho}^* \triangleq -\boldsymbol{\rho}(t = T)$. Consequently, the Metropolis-Hasting acceptance

probability in line 5 of algorithm 1 calculates to

$$\rho = \min[1, \exp(-H(\boldsymbol{X}^*, \boldsymbol{\rho}^*) + H(\boldsymbol{X}, \boldsymbol{\rho}))] \; . \tag{2.62}$$

Note that the quotient of the probability of the proposed step and the reverse step $\mathrm{Q}(\boldsymbol{X}^{(s-1)} \mid \boldsymbol{X}^*)$ / $\mathrm{Q}(\boldsymbol{X}^* \mid \boldsymbol{X}^{(s-1)})$ cancels out, since the proposal state is given deterministically by the Hamiltonian dynamics and the reverse step would also occur with probability one due to negating the impulse in the final state $\boldsymbol{\rho}^*$.

By calculating the temporal derivative of the Hamiltonian and applying the dynamics (2.61) we get

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \sum_{d=1}^{D} \left(\frac{\partial H}{\partial X_d} \frac{\mathrm{d}X_d}{\mathrm{d}t} + \frac{\partial H}{\partial \rho_d} \frac{\mathrm{d}\rho_d}{\mathrm{d}t} \right) = \sum_{d=1}^{D} \left(\frac{\partial H}{\partial X_d} \frac{\partial H}{\partial \rho_d} - \frac{\partial H}{\partial \rho_d} \frac{\mathrm{d}H}{\mathrm{d}X_d} \right) = 0 \, ,$$

and thus we see that the Hamiltonian is conserved when following Hamilton's differential equations. Hence, if the dynamics are simulated perfectly the acceptance probability (2.62) becomes one. However, in practice we will seldom be able to find an analytic solution for eq. (2.61) and will thus have to revert to numerical simulation. The accuracy will be limited by the necessity to discretize the differential equations and compute the states at fixed time steps. Since each time step depends on the step before, errors accumulate over time and the invariance of the Hamiltonian will only be preserved approximately.

In general any numerical integration method can be used to simulate the system dynamics eq. (2.61). One simple, yet numerically stable, method is the leapfrog method (Griebel et al., 2003). The leapfrog method works by evaluating the state variables $\boldsymbol{X}$ at discretized times that are integer multiples of a step size ε, while the impulse variables ρ are evaluated at times that are shifted by half the step size. Thus $\boldsymbol{X}(t)$ is evaluated for $t \in \{0, \varepsilon, 2\varepsilon, 3\varepsilon, \dots\}$ and $\boldsymbol{\rho}(t)$ is evaluated for $t \in \{0.5\varepsilon, 1.5\varepsilon, 2.5\varepsilon, 3.5\varepsilon, \dots\}$. The full method is described in algorithm 3.

It can be shown (Neal et al., 2011) that the interleaved calculation of state and impulse ensures that $H(\boldsymbol{X}, \boldsymbol{\rho})$ is preserved exactly, despite the finite discretization of time, because each update step is a volume-preserving shear transformation. Therefore, using the leapfrog method within HMC ensures that the acceptance probability (2.62) is nearly one.

2.3 Optimization

Mathematical optimization (Lange, 2013) deals with finding minimum or maximum values of a function. Functions to be minimized are often called "cost" or "loss" functions. Given a loss function $\mathscr{L} : \mathbb{R}^d \to \mathbb{R}$ we are usually concerned with the problem of finding optimal parameters

Algorithm 3: HMC state proposal using leapfrog integration

Input: current state $\boldsymbol{X}(0) \triangleq \boldsymbol{X}$ and current impulse $\boldsymbol{\rho}(0) \triangleq \boldsymbol{\rho}$; potential energy $U(\boldsymbol{X})$
Output: proposal state $\boldsymbol{X}^* \triangleq \boldsymbol{X}(T)$ and proposal impulse $\boldsymbol{\rho}^* \triangleq \boldsymbol{\rho}(T)$
Parameters: step size ϵ; run time T; masses $\boldsymbol{m}$

$\forall i\colon \rho_i(\varepsilon/2) \longleftarrow \rho_i - \frac{\varepsilon}{2}\frac{\partial U}{\partial X_i}$ // initial half-step for impulses

for $t \in \{1\epsilon, 2\epsilon, \ldots, T\}$ **do**

  $\forall i\colon X_i(t) \longleftarrow X_i(t-\varepsilon) + \varepsilon\,\frac{\rho_i(t-\varepsilon/2)}{m_i}$ // full-step for states

  $\forall i\colon \rho_i(t+\varepsilon/2) \longleftarrow \rho_i(t-\varepsilon/2) - \varepsilon\,\frac{\partial U}{\partial X_i}\Big|_{\boldsymbol{X}=\boldsymbol{X}(t)}$ // full-step for impulses

$\forall i\colon \rho_i(T) \longleftarrow \rho_i(T-\varepsilon/2) - \frac{\varepsilon}{2}\,\frac{\partial U}{\partial X_i}\Big|_{\boldsymbol{X}=\boldsymbol{X}(T)}$ // final half-step for impulses

$\boldsymbol{\theta}^* \in \mathbb{R}^d$ determined by

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathscr{L}(\boldsymbol{\theta})\,. \tag{2.63}$$

When dealing with neural networks, GPs and related models the loss function is usually non-convex, so an infinite number of local minima can exist and finding the global minimum is usually infeasible due to the high dimensionality of the parameter vector $\boldsymbol{\theta}$. However, in these applications $\mathscr{L}$ is differentiable at least almost everywhere, thus iterative optimization methods based on local first and second order derivative information are a good choice here.

In machine learning the structure of the loss function is often a sum of losses over individual training samples,

$$\mathscr{L}(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T} L_t(\boldsymbol{\theta})\,, \tag{2.64}$$

where t is the sample index and T is the number of training samples.

2.3.1 Gradient Descent

Gradient descent uses the gradient of the loss function w.r.t. $\boldsymbol{\theta}$ to perform incremental steps towards a local minimum. It uses the fact that the gradient of a function points to the direction of its steepest ascent. The parameters $\boldsymbol{\theta}^0$ are initialized randomly. In each step s we set

$$\boldsymbol{\theta}^{s+1} \longleftarrow \boldsymbol{\theta}^s - \eta\,\nabla\mathscr{L}(\boldsymbol{\theta}^s) \tag{2.65}$$

where $\nabla\mathscr{L}$ denotes the gradient of $\mathscr{L}$ w.r.t. $\boldsymbol{\theta}$ and $\eta > 0$ is called the step rate. It determines the size of each step and a good choice is crucial for the success of this method. If η is too large,

the algorithm becomes unstable and the sequence of $\boldsymbol{\theta}$ diverges. If η is too small, the method takes more time than necessary and is more prone to getting stuck in small local minimal. Many heuristics for automatic adaption and scheduling of η exist.

Assuming that $\mathscr{L}$ is a convex function and Lipschitz continuous with constant $K > 0$, i.e.

$$\left|\nabla\mathscr{L}(\boldsymbol{\theta}) - \nabla\mathscr{L}(\boldsymbol{\theta'})\right| \leq K\left|\boldsymbol{\theta} - \boldsymbol{\theta'}\right| \tag{2.66}$$

for any $\boldsymbol{\theta}$ and $\boldsymbol{\theta'}$, then gradient descent with fixed step size $\eta \leq 1/K$ satisfies

$$|\mathscr{L}(\boldsymbol{\theta}^s) - \mathscr{L}(\boldsymbol{\theta}^*)| \leq \frac{\left|\boldsymbol{\theta}^0 - \boldsymbol{\theta}^*\right|^2}{2\,\eta\,s}\,, \tag{2.67}$$

meaning that gradient descent converges for convex functions with a rate of $\mathcal{O}(1/\eta s)$ (Boyd et al., 2004).

This form of gradient descent is also called batch training, because the loss is taken over all training samples, i.e. the whole batch of training samples is consulted to perform an update of the parameters.

2.3.2 Stochastic Gradient Descent

Stochastic gradient descent (Y. A. LeCun et al., 2012) exploits the structure (2.64) of the loss functions and works as follows. Start with epoch r set to zero, $r \longleftarrow 0$. Shuffle the training set and then iterate over the training set performing the following update step iteratively,

$$\boldsymbol{\theta}^{rT+t+1} \longleftarrow \boldsymbol{\theta}^{rT+t} - \frac{\eta}{T}\nabla L_t(\boldsymbol{\theta}^{rT+t})\,. \tag{2.68}$$

Increase epoch counter r, reshuffle the training set and repeat until convergence.

Thus in contrast to batch learning this method performs an update of the parameters after *each* training sample. A compromise between these two extremes is mini-batch training. This method partitions the training set into random, disjoint subsets called mini-batches of size M. Training is performed by iterating over the mini-batch index $b \in \{1, \ldots, B\}$ using the update rule

$$\boldsymbol{\theta}^{rB+b+1} \longleftarrow \boldsymbol{\theta}^{rB+b} - \frac{\eta}{T}\sum_{t=Mb}^{Mb+M-1}\nabla L_t(\boldsymbol{\theta}^{rB+b}) \tag{2.69}$$

where $B \triangleq T/M$ is the number of mini-batches. After the iteration over all mini-batches b is completed, the repetition counter r is increment and the process restarted.

Mini-batch training has two advantages over batch training. One, it is computationally more efficient, because often a subset of the training set is sufficient to calculate a good enough

gradient. Two, it leads to better optima since the randomness of the mini-batches adds a small amount of noise to the gradient signal, which can help escape local minima. However it is also more prone to instabilities as fewer training samples enter the gradient estimate; thus η must be reduced appropriately.

2.3.3 Momentum

The momentum technique (Rumelhart, G. E. Hinton, et al., 1988) mimics the physical behavior of a particle moving on the surface of the loss function. In this physical model the acceleration of the particle is determined by the gradient of the loss function and its position is updated according to its velocity. In the case of batch learning this leads to the following update rules for velocities $\dot{\boldsymbol{\theta}}$ and parameters $\boldsymbol{\theta}$ from step s to step $s+1$,

$$\dot{\boldsymbol{\theta}}^{s+1} \longleftarrow \alpha \dot{\boldsymbol{\theta}}^{s} - \eta \nabla \mathscr{L}(\theta^{s}) \,, \tag{2.70a}$$

$$\boldsymbol{\theta}^{s+1} \longleftarrow \boldsymbol{\theta}^{s} + \dot{\boldsymbol{\theta}}^{s+1} \,, \tag{2.70b}$$

where $0 \leq \alpha < 1$ is the momentum factor. Momentum can also be applied to mini-batch learning by adapting the update rules in the same way.

2.3.4 Optimization Methods for Neural Networks

In multi-layer neural networks the gradient w.r.t. to the weights of the lower layers can get very small due to iterated application of the logistic function, leading to very small update steps on these weights. A method to avoid this is to just use the sign (positive or negative) of each gradient element to decide whether to increase or decrease the corresponding parameter. An optimization method based on this concept for batch learning is resilient backpropagation (Rprop) by Riedmiller et al. (1992).

For mini-batch learning Tieleman et al. (2012) proposed the RMSProp (root mean square propagation) method that averages the signs of the gradient elements from multiple mini-batches of training data. It works by keeping a running average of the gradient in the auxiliary variable $\boldsymbol{\psi}$. The method iterates over the mini-batch index b and uses the following update equations,

$$\boldsymbol{\psi}^{rB+b+1} \longleftarrow \gamma \, \boldsymbol{\psi}^{rB+b} + \frac{1-\gamma}{T} \left(\sum_{t=Mb}^{Mb+M-1} \nabla L_t(\boldsymbol{\theta}^{rB+b}) \right)^2 \,, \tag{2.71a}$$

$$\boldsymbol{\theta}^{rB+b+1} \longleftarrow \boldsymbol{\theta}^{rB+b} - \frac{\eta}{T} \frac{\sum_{t=Mb}^{Mb+M-1} \nabla L_t(\boldsymbol{\theta}^{rB+b})}{\sqrt{\boldsymbol{\psi}^{rB+b+1}}} \,, \tag{2.71b}$$

where the square in (2.71a) and the quotient in (2.71b) are to be taken *elementwise*. It divides each gradient element by its average magnitude, leading to a sign estimate being used for the parameter updates. The parameter γ controls the decay rate of the average and a usual setting is $\gamma = 0.9$. RMSProp leads to a significant improvement of the learning speed and results of deep neural networks.

D. Kingma et al. (2014) combined the ideas of RMSProp and momentum leading to the development of the Adam (adaptive moment estimation) optimizer. It works in a mini-batch setting and keeps running averages for both the gradient and second moment of the gradient. Iterating over the mini-batch index b, Adam uses the following update equations,

$$\dot{\boldsymbol{\theta}}^{rB+b+1} \longleftarrow \alpha\, \dot{\boldsymbol{\theta}}^{rB+b} + \frac{1-\alpha}{T} \sum_{t=Mb}^{Mb+M-1} \nabla L_t(\boldsymbol{\theta}^{rB+b})\,, \tag{2.72a}$$

$$\boldsymbol{\psi}^{rB+b+1} \longleftarrow \gamma\, \boldsymbol{\psi}^{rB+b} + \frac{1-\gamma}{T} \left(\sum_{t=Mb}^{Mb+M-1} \nabla L_t(\boldsymbol{\theta}^{rB+b}) \right)^2\,, \tag{2.72b}$$

$$\boldsymbol{\theta}^{rB+b+1} \longleftarrow \boldsymbol{\theta}^{rB+b} - \eta \frac{\dot{\boldsymbol{\theta}}^{rB+b+1}/(1-\alpha)}{\sqrt{\boldsymbol{\psi}^{rB+b+1}/(1-\gamma)} + \varepsilon}\,, \tag{2.72c}$$

where the parameter α controls the decay rate of the gradient, γ controls the decay rate of the second moment of the gradient, η is the step rate and ε is a small number to prevent division by zero.

2.4 Gaussian Processes

A Gaussian process (Lawrence, 2005; Neal, 1997; O'Hagan et al., 1978; C. K. Williams et al., 1996, 2006) describes a distribution over scalar-valued functions $f(\boldsymbol{x})$ with multivariate inputs $\boldsymbol{x}$. Consider a matrix $X \in \mathbb{R}^{N\times D}$ of N input values with D dimensions. Any finite number of function values $f(X_{i\star})$, $i \in \{1, 2, \dots N\}$, has a joint multivariate normal distribution, which is defined by the mean function $m : \mathbb{R}^D \to \mathbb{R}$ and covariance function $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ of the GP. Let $\boldsymbol{f} \in \mathbb{R}^N$ be the vector of function values given by $f_i \triangleq f(X_{i\star})$. If the function f follows a GP,

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'}))\,, \tag{2.73}$$

this means that the function values $\boldsymbol{f}$ are normally distributed,

$$\boldsymbol{f} \sim \mathcal{N}(\boldsymbol{m}, K(X, X))\,, \tag{2.74}$$

with mean and covariance determined by the inputs X,

$$m_i \triangleq m(X_{i\star})\,, \tag{2.75a}$$

$$K(X,X)_{ij} \triangleq k(X_{i\star}, X_{j\star})\,. \tag{2.75b}$$

This implies that $k(\boldsymbol{x},\boldsymbol{x}')$ must by symmetric and positive-definite to be a valid covariance function.

GPs can be defined with a variety of mean and covariance functions. A common choice is to use the zero mean function, $m(\boldsymbol{x}) = 0$, thus regularizing the function magnitude. The squared exponential (SE) covariance function is given by

$$k_{\mathrm{SE}}(\boldsymbol{x},\boldsymbol{x'}) = \sigma_f^2 \exp\left(-\frac{|\boldsymbol{x}-\boldsymbol{x}'|^2}{2\,l^2}\right)\,. \tag{2.76}$$

Since it can be differentiated infinitely many times, GPs using the SE covariance function produce smooth functions. The parameter l controls the characteristic length-scale of the covariance function, which can be understood as the increase in distance that leads to a decrease in covariance by factor $1/\mathrm{e}$. Function samples from a GP with SE covariance function using lengthscale l will have similar values within distance l with high probability.

Another choice is the automatic relevance determination (ARD) covariance function given by

$$k_{\mathrm{ARD}}(\boldsymbol{x},\boldsymbol{x'}) = \sigma_f^2 \exp\left[-\frac{1}{2}\sum_{d=1}^{D}\left(\frac{x_d - x'_d}{l_d}\right)^2\right]\,. \tag{2.77}$$

It is an extension of the SE covariance function and uses a separate lengthscale parameter l_d for each input dimension. This makes it possible to automatically weight the importance of each input dimension in the prediction and ignore dimensions that are not correlated with the targets.

The parameter σ_f^2 common to both covariance functions specifies the variance of the samples and thus controls the average deviation from the mean function.

2.4.1 Gaussian Process Regression

GPs can be used to perform regression by conditioning the GP on a set of observed function values. Consider a function $f(\boldsymbol{x})$ distributed according to (2.73). Given a finite set of observation points $X_{i\star}$, $i \in \{1,2,\ldots,N\}$, with corresponding observed values $\boldsymbol{y} = (y_1, y_2, \ldots, y_N)^T$ where $y_i = f(X_{i\star})$ the conditional distribution of the predicted function values $f(\boldsymbol{x}^*)$ at some

test points $\boldsymbol{x}^*$ is another GP,

$$f(\boldsymbol{x}^*) \,|\, X, \boldsymbol{y} \sim \mathcal{GP}(m^*(\boldsymbol{x}^*), k^*(\boldsymbol{x}^*, \boldsymbol{x}^{*\prime}))\,, \tag{2.78}$$

with predictive mean and covariance functions given by

$$m^*(\boldsymbol{x}^*) \triangleq m(\boldsymbol{x}^*) + K(\boldsymbol{x}^*, X)\, K(X, X)^{-1}\, (\boldsymbol{y} - \boldsymbol{m})\,, \tag{2.79a}$$

$$k^*(\boldsymbol{x}^*, \boldsymbol{x}^{*\prime}) \triangleq k(\boldsymbol{x}^*, \boldsymbol{x}^{*\prime}) - K(\boldsymbol{x}^*, X)\, K(X, X)^{-1}\, K(X, \boldsymbol{x}^{*\prime})\,, \tag{2.79b}$$

where $K(\boldsymbol{x}^*, X)_i \triangleq k(\boldsymbol{x}^*, X_{i\star})$, $K(X, \boldsymbol{x}^{*\prime}) \triangleq k(X_{i\star}, \boldsymbol{x}^{*\prime})$ and $m_i \triangleq m(X_{i\star})$. This follows directly from the equations for the mean and covariance (2.34) of a conditional, multivariate normal distribution, since any finite number of points from the GP must be consistent with it.

The above result assumes that the function values were observed exactly and may lead to suboptimal results because the GP will fit the observations without any tolerance, leading to overfitting. To avoid this, we can assume that each observation is afflicted with iid. Gaussian noise, i.e. $y_i = f(X_{i\star}) + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$. This corresponds to adding σ_n^2 to the diagonal of the covariance matrix $K(X, X)$. Thus, in the case of noisy observations we obtain for the mean and covariance functions of the predictive GP,

$$m^*(\boldsymbol{x}^*) \triangleq m(\boldsymbol{x}^*) + K(\boldsymbol{x}^*, X)\, [K(X, X) + \sigma_n^2 \mathbb{1}]^{-1}\, (\boldsymbol{y} - \boldsymbol{m})\,, \tag{2.80a}$$

$$k^*(\boldsymbol{x}^*, \boldsymbol{x}^{*\prime}) \triangleq k(\boldsymbol{x}^*, \boldsymbol{x}^{*\prime}) - K(\boldsymbol{x}^*, X)\, [K(X, X) + \sigma_n^2 \mathbb{1}]^{-1}\, K(X, \boldsymbol{x}^{*\prime})\,, \tag{2.80b}$$

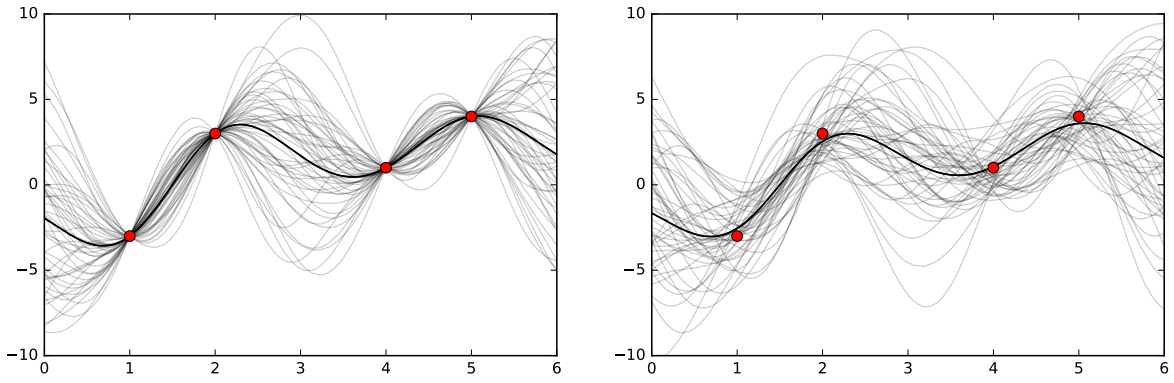
where $\mathbb{1}$ is the identity matrix.

An example for GP regression in one dimensional space without and with observation noise is shown in fig. 2.2.

2.4.2 Marginal Likelihood

So far it was assumed that the parameters σ_f, σ_n and l are known in advance, for example by some domain knowledge of the underlying process generating the function. However, in many regression problems we are just given the observations without additional information. Thus it becomes necessary to estimate good values for these parameters from the data alone. Since a GP is a probabilistic model we can calculate the likelihood of the observed data and maximize it w.r.t. the parameters. The marginal log-likelihood of the observations is calculated using the PDF of the multivariate normal distribution (2.27) and evaluates to

$$\log \mathrm{P}(\boldsymbol{y} \,|\, X) = -\frac{1}{2} \boldsymbol{y}^T (K(X, X) + \sigma_n^2 \mathbb{1})^{-1} \boldsymbol{y} - \frac{1}{2} \log \left| K(X, X) + \sigma_n^2 \mathbb{1} \right| - \frac{D}{2} \log 2\pi\,. \tag{2.81}$$



(a) without noise (b) with noise

Figure 2.2: Gaussian process regression using the squared exponential covariance function. Observations are shown as red dots. The black line shows the predictive mean and the grey lines are samples from the predictive GP. (a) The observations are treated as exact with $\sigma_n = 0$; thus all GP samples must pass through them. (b) The observations are treated as afflicted with noise ($\sigma_n = 1.3$); thus the predictive GP does not need to fit them exactly.

It is a measure of how well the GP fits the data. Maximizing it using an iterative optimization method of our choice (section 2.3) w.r.t. σ_f, σ_n and l results in a maximum likelihood estimate of these parameters.

2.4.3 Derivative Observations and Predictions

Since the derivative is a linear operation, the derivative of a Gaussian process is also a Gaussian process (Riihimäki et al., 2010; Solak et al., 2003). It can be shown that given two points $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ with values $y^i = f(\boldsymbol{x}^i)$ and $y^j = f(\boldsymbol{x}^j)$ respectively, it holds for the covariances of the derivatives that

$$\operatorname{Cov}\left(\frac{\partial y^i}{\partial x_g^i}, y^j\right) = \frac{\partial}{\partial x_g^i}\operatorname{Cov}(y^i, y^j)\,, \tag{2.82a}$$

$$\operatorname{Cov}\left(\frac{\partial y^i}{\partial x_g^i}, \frac{\partial y^j}{\partial x_h^j}\right) = \frac{\partial^2}{\partial x_g^i\, \partial x_h^j}\operatorname{Cov}(y^i, y^j)\,. \tag{2.82b}$$

For the SE covariance function these expressions evaluate to

$$\operatorname{Cov}\left(\frac{\partial y^i}{\partial x_g^i}, y^j\right) = \frac{\partial k_{\mathrm{SE}}}{\partial x_g^i} = -\sigma_f^2 \exp\left(-\frac{\left|\boldsymbol{x^i} - \boldsymbol{x^j}\right|^2}{2\,l^2}\right) \frac{x_g^i - x_g^j}{l^2}\,, \tag{2.83a}$$

$$\operatorname{Cov}\left(\frac{\partial y^i}{\partial x_g^i}, \frac{\partial y^j}{\partial x_h^j}\right) = \frac{\partial^2 k_{\mathrm{SE}}}{\partial x_g^i\, \partial x_h^i} = \sigma_f^2 \exp\left(-\frac{\left|\boldsymbol{x^i} - \boldsymbol{x^j}\right|^2}{2\,l^2}\right) \left(\frac{\delta_{gh}}{l^2} - \frac{x_g^i - x_g^j}{l^2}\frac{x_h^i - x_h^j}{l^2}\right)\,, \tag{2.83b}$$

where δ_{gh} is the Kronecker delta function. Thus we can introduce observations of the derivatives into the training data of the GP. To do so the sets of training points and target values are extended with the derivative observations and the covariance matrix uses (2.82a) and (2.82b) for the appropriate entries. The joint vector of targets and target derivatives becomes

$$\widehat{\boldsymbol{y}} = \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y'} \end{bmatrix}$$

where $\boldsymbol{y'}$ denotes the derivative targets and the covariance matrix K is composed according to

$$\widehat{K} = \begin{bmatrix} K^{yy} & K^{yy'} \\ (K^{yy'})^T & K^{y'y'} \end{bmatrix}$$

with

$$K_{ij}^{yy} = k(\boldsymbol{x}^i, \boldsymbol{x}^j)\,, \qquad K_{ij}^{yy'} = \frac{\partial k}{\partial x_g^j}(\boldsymbol{x}^i, \boldsymbol{x}^j)\,, \qquad K_{ij}^{y'y'} = \frac{\partial^2 k}{\partial x_g^i\, \partial x_h^j}(\boldsymbol{x}^i, \boldsymbol{x}^j)\,.$$

Predictions are then obtained by using eq. (2.79) with $\boldsymbol{y}$ and K replaced by $\widehat{\boldsymbol{y}}$ and $\widehat{K}$ respectively.

We can also calculate the mean and the variance of the derivative for a test point $\boldsymbol{x}^*$. Since the mean of the derivative is equal to the derivative of the mean we obtain

$$\mathrm{E}\left[\frac{\partial f^*}{\partial x_d^*}\right] = \frac{\partial \mathrm{E}[f^*]}{\partial x_d^*} = \left(\frac{\partial \boldsymbol{k^*}}{\partial \boldsymbol{x^*}}\right)^T (K + \sigma_n^2 I)^{-1}\, \boldsymbol{y}\,, \tag{2.84}$$

and for the variance of the derivative we get

$$\mathrm{Var}\left(\frac{\partial f^*}{\partial x_d^*}\right) = \frac{\partial^2 k(\boldsymbol{x^*}, \boldsymbol{x^*})}{\partial (x_d^*)^2} - \left(\frac{\partial \boldsymbol{k^*}}{\partial x_d^*}\right)^T (K + \sigma_n^2 I)^{-1}\, \frac{\partial \boldsymbol{k^*}}{\partial x_d^*}\,. \tag{2.85}$$

A possible application of this technique is to sample random but smooth trajectories (for example for robots) with constraints on the positions, velocities and accelerations at the start and the end of the trajectory.

2.5 Artificial Neural Networks

An artificial neural network (Haykin, 1994) is a mathematical model of a biological neural network, as it occurs in the brain. An ANN consists of units called neurons that receive inputs from other neurons, perform a computation on their inputs and output a single value. Neurons are usually arranged in so-called layers, which are groups of neurons that receive their inputs

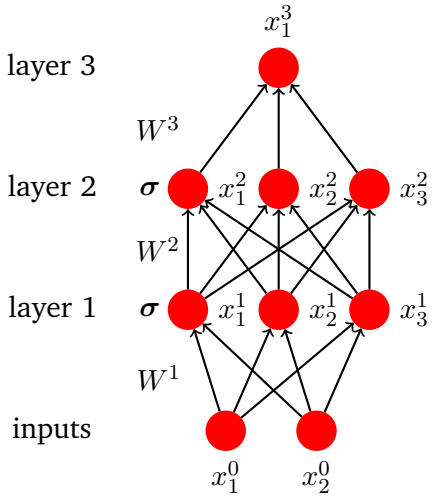


Figure 2.3: A feed-forward neural network consisting of three layers. Each filled circle represents a neuron x_i^l. The directed connections show the input a neuron receives. With each connection from neuron x_j^{l-1} to neuron x_i^l there is a weight W_{ij}^l associated.

from the same set of neurons. There are different neural network architectures. The most basic architecture is the feed-forward, in which one layer is stacked upon another and the data always flows from one layer to the next. For illustration let us consider the feed-forward neural network with multidimensional input and scalar output, as shown in fig. 2.3.

Each neuron x_i^l, where $l \in \{1, \dots, L\}$ denotes the layer and $i \in \{1, \dots, N_l\}$ denotes the index of the neuron within the layer, receives input from the neurons x_j^{l-1}, $j \in \{1, \dots, M\}$, of the previous layer. The output of a neuron is given by a weighted sum of its inputs plus a so-called bias propagated through a non-linear activation function. The weights are stored in a weights matrix $W^l \in \mathbb{R}^{N_l \times N_{l-1}}$ and the biases are given by a vector $\boldsymbol{b} \in \mathbb{R}^{N_l}$. The activation of a neuron is

$$a_i^l(\boldsymbol{x}^{l-1}) = b_i^l + \sum_{j=1}^{N_{l-1}} W_{ij}^l x_j^{l-1} \,. \tag{2.86}$$

and its output is given by

$$x_i^l(\boldsymbol{x}^{l-1}) = \sigma(a_i^l) = \sigma\left(b_i^l + \sum_{j=1}^{N_{l-1}} W_{ij}^l x_j^{l-1} \right) \,. \tag{2.87}$$

The purpose of the activation function $\sigma(t)$ is to introduce a non-linearity into the neural network. Otherwise the whole network could be described by one matrix multiplication and its output would thus be an affine function of its input. A typical choice for the activation function

$\sigma(t)$ is the logistic function

$$\sigma(t) = \frac{1}{1 + \mathrm{e}^{-t}} \tag{2.88}$$

or another sigmoid-shaped function like the hyperbolic tangent $\sigma(t) = \tanh(t)$. This choice was historically inspired by the thresholding behavior of biological neurons and because the derivative of the logistic function is inexpensive to compute.

Matrix multiplication can be used to jointly compute the activations of all neurons in one layer more efficiently, resulting in

$$\boldsymbol{a}^l(\boldsymbol{x}^{l-1}) = W^l\,\boldsymbol{x}^{l-1} + \boldsymbol{b}^l \tag{2.89}$$

and $\boldsymbol{x}^l(\boldsymbol{a}^l) = \boldsymbol{\sigma}(\boldsymbol{a}^l)$ where the activation function is applied elementwise implicitly, that is $\boldsymbol{\sigma}_i(\boldsymbol{t}) \triangleq \sigma(t_i)$. For convenience, the parameters of a neural network are usually combined into a parameter vector $\theta \triangleq \{W^1, \boldsymbol{b}^1, \ldots, W^L, \boldsymbol{b}^L\}$. A feed-forward network defines a function $\boldsymbol{f}_\theta(\boldsymbol{x}^0)$ depending on its inputs $\boldsymbol{x}^0$ and parameterized by θ, given by

$$\boldsymbol{f}_\theta(\boldsymbol{x}^0) = \boldsymbol{x}^L(\boldsymbol{x}^{L-1}(\cdots \boldsymbol{x}^1(\boldsymbol{x}^0)\cdots))\,. \tag{2.90}$$

2.5.1 Regression and Classification

Feed-forward networks can be used to approximate an arbitrary function $\boldsymbol{g}(\boldsymbol{z})$ from a training set $\{(\boldsymbol{z}_s, \boldsymbol{t}_s) : s \in \{1, \ldots, S\}\}$ of S samples, consisting of pairs of inputs $\boldsymbol{z}_s$ and corresponding targets $\boldsymbol{t}_s = \boldsymbol{g}(\boldsymbol{z}_s)$. To make the output of the feed-forward network $\boldsymbol{f}_\theta(\boldsymbol{z})$ approximate the target values $\boldsymbol{t}$, appropriate values for the parameters θ must be determined. To do so, a loss function is defined; for regression tasks a usual choice is the average mean-squared error over the training set,

$$\mathscr{L}_{\mathrm{reg}}(\theta) = \frac{1}{S}\sum_{s=1}^{S} (\boldsymbol{f}_\theta(\boldsymbol{z}_s) - \boldsymbol{t}_s)^2\,. \tag{2.91}$$

For classification problems it is desirable to obtain the predictions in the form of probabilities that a given input is in a particular class. However, the network outputs cannot be directly interpreted as class probabilities since they are not normalized. Thus the function $\mathrm{softmax} : \mathbb{R}^C \to \mathbb{R}^C$, defined by

$$\mathrm{softmax}(\boldsymbol{o})_i \triangleq \frac{\exp o_i}{\sum_j \exp o_j}\,, \tag{2.92}$$

is used to translate raw network outputs $\boldsymbol{o}$ into probabilities. It can easily be verified that $0 \le \mathrm{softmax}(\boldsymbol{o})_i \le 1$ and $\sum_i \mathrm{softmax}(\boldsymbol{o})_i = 1$; thus the softmax provides valid probabilities for a categorical distribution as defined in section 2.2.6. We have for the probability that input $\boldsymbol{z}$

belongs to class $t \in \{1, 2, \dots, C\}$,

$$\mathrm{P}(t \,|\, \boldsymbol{z}) = \mathrm{Cat}(t \,|\, \mathrm{softmax}(\boldsymbol{f}_\theta(\boldsymbol{z}))) \,. \tag{2.93}$$

Since this model gives a probabilistic interpretation of the predictions of the neural network, it is natural to train it by maximizing the likelihood of the parameters θ. Thus we use the negative log-likelihood as the loss,

$$\mathscr{L}_{\text{class}}(\theta) = -\frac{1}{S}\sum_{s=1}^{S} \log \mathrm{P}(t_s \,|\, \boldsymbol{z}_s) = -\frac{1}{S}\sum_{s=1}^{S} \log[\mathrm{softmax}(\boldsymbol{f}_\theta(\boldsymbol{z}_s))_{t_s}] \,, \tag{2.94}$$

where $\boldsymbol{z}_s$ is input sample s and t_s is the corresponding target class. A one-hot encoding $T \in \{0,1\}^{S\times C}$ encodes the target class t_s as a vector of length C with $T_{sc} = 1$ if $t_s = c$ and $T_{sc} = 0$ otherwise. If such a one-hot encoding is used, then the loss can be written as

$$\mathscr{L}_{\text{class}}(\theta) = -\frac{1}{S}\sum_{s=1}^{S} T_{s\star} \cdot \log \mathrm{softmax}(\boldsymbol{f}_\theta(\boldsymbol{z}_s)) \,, \tag{2.95}$$

where the $\cdot$ denotes the scalar product between two vectors.

For both regression and classification the optimal parameters θ^* are found by minimizing $\mathscr{L}(\theta)$, i.e.

$$\theta^* = \arg\min_\theta \mathscr{L}(\theta) \,. \tag{2.96}$$

Since this minimization cannot be performed analytically due to the non-linearities induced by the activation functions, iterative optimization techniques are employed as described in section 2.3. Historically the gradient was calculated using a technique called backpropagation (Y. LeCun, B. Boser, et al., 1989) based on iterative applications of the chain rule. Modern software packages that implement neural networks use a more flexible approach. They interpret the neural network as a generic function and use reverse mode automatic differentiation, which is described in detail in section 4.1, to compute all its derivatives. This allows extending the basic ANN model with additional elements with the only requirement that each function element the model is composed of must be differentiable at least almost everywhere.

2.5.2 Universal Approximation Theorem

It was shown by Hornik et al. (1989) that feed-forward networks with at least one hidden layer and a sigmoidal activation function are able to approximate any function arbitrarily well given a sufficient number of hidden units. However, even though such a network is an universal

function approximator, there is no guarantee that it can approximate a function *efficiently*. If the architecture of the neural network is not a good match for a particular problem, a very large number of neurons may be required to obtain acceptable results. Increasing the number of layer can make a feed-forward network more powerful, because upper layers can compose more complex functions from basic functions learned by neurons in the lower layers (Bengio, 2009).

Chapter 3

A Continuum between Addition and Multiplication

Common feed-forward neural networks are widely successful at learning various datasets, but they struggle to learn certain classes of functions efficiently, which can be attributed to the lack of multiplicative interactions. Vice versa, networks solely build on multiplicative interactions excel at certain problems but fail at many problems for which common neural networks work well. Existing approaches to combine both additive and multiplicative networks either use a fixed assignment of operations (which may be inspired from domain knowledge) or require discrete optimization to determine what function a neuron should perform. This leads either to an inefficient distribution of computational resources or an extensive increase in the computational complexity of the training procedure due to the necessity of empirically evaluating different assignments of neurons to the additive or multiplicative regime.

In this chapter we introduce a novel, parameterizable activation function based on the mathematical concept of non-integer functional iteration. Non-integer functional iteration generalizes the concept of iterated application of a function to non-integer number of iterations; for example the 1/3-iterate of the exponential function, is the function that needs to be applied iteratively 3 times to calculate the exponential. We will show how this concept allows the operation each neuron performs to be smoothly and, most importantly, differentiablely adjusted between addition and multiplication. Finally, we show how this enables the decision between addition and multiplication to be integrated into the standard backpropagation training procedure of a neural network. Issues regarding numerical precision are discussed and the proposed family of activation functions is validated on a synthetic polynomial dataset and on the standard MNIST digit recognition task.

Contributions to this Chapter

The original proposal of this idea and the formulation of the concepts in this chapter were done by me and originally published in Urban and Smagt (2015). Implementation, benchmarking and experiments were done by Wiebke Köpp under my guidance leading to publications in Köpp (2015) and Köpp et al. (2016). The results of these experiments are reproduced here for completeness and explicitly marked as such.

3.1 Examples for the Utility of Multiplicative Interactions

We have already mentioned in chapter 1 that multiplicative interactions have proven useful for attention mechanisms in RNNs and for modeling the relations between two images. Here we present a theoretic argument involving the representability of polynomials and show how multiplicative interactions can make the task of learning *variable* pattern shifts efficiently representable.

Variable Pattern Shift

As an example for why additive and multiplicative interactions in neural networks are useful, consider the dynamic pattern shift task shown in fig. 3.1a. For simplicity we consider this problem in one dimension, but this argument can be extended to two dimensions (representing images for example) without much effort. The input consists of a binary vector $\boldsymbol{x}$ of N elements and an integer $m \in \{0, 1, \ldots, N-1\}$. The desired output $\boldsymbol{y} \in \mathbb{R}^N$ is $\boldsymbol{x}$ circularly shifted by m elements to the right,

$$y_n = x_{(n-m) \bmod N} ,$$

where

$$(n-m) \bmod N \triangleq \begin{cases} n-m & \text{if } n-m > 0 \\ N+n-m & \text{if } n-m \leq 0 \end{cases} .$$

A method to implement this task efficiently in a neural architecture is based on the shift theorem of the discrete Fourier transform (DFT) (Brigham, 1988). Let $\mathcal{F}(\boldsymbol{x})_k$ denote the k-th element of the DFT of $\boldsymbol{x}$. By definition we have

$$\mathcal{F}(\boldsymbol{x})_k = \sum_{n=0}^{N-1} x_{n+1} \, \mathrm{e}^{-2\pi i \frac{(k-1)n}{N}}$$

and its inverse is given by

$$\mathcal{F}^{-1}(\boldsymbol{X})_n = \frac{1}{N}\sum_{k=0}^{N-1} X_{k+1}\, \mathrm{e}^{2\pi i \frac{k(n-1)}{N}} .$$

The shift theorem states that a shift by m elements in the time domain is equivalent to a multiplication by factor $\mathrm{e}^{-2\pi i(k-1)m/N}$ in the frequency domain,

$$\mathcal{F}(\boldsymbol{y})_k = \mathcal{F}(\boldsymbol{x})_k\, \mathrm{e}^{-2\pi i \frac{(k-1)m}{N}} .$$

Hence the shifted pattern can be calculated using

$$\boldsymbol{y} = \mathcal{F}^{-1}(\mathcal{F}(\boldsymbol{x})_k\, \mathrm{e}^{-2\pi i \frac{(k-1)m}{N}}) .$$

Using the above definitions its v-th component of $\boldsymbol{y}$ is given by

$$y_v = \frac{1}{N}\sum_{k=0}^{N-1} \mathrm{e}^{2\pi i \frac{k(v-1)}{N}} \left[\mathrm{e}^{-2\pi i \frac{km}{N}} \sum_{n=0}^{N-1} x_{n+1}\, \mathrm{e}^{-2\pi i \frac{kn}{N}}\right] .$$

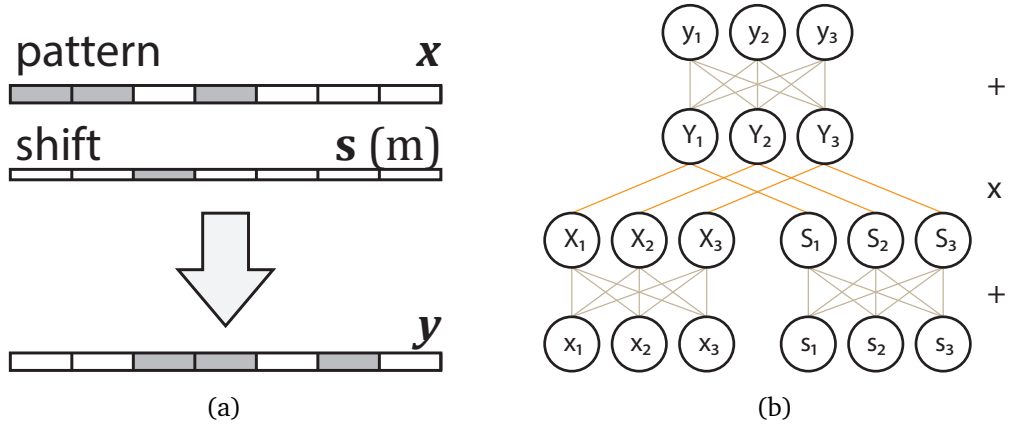
If we encode the shift amount m as a one-hot vector $\boldsymbol{s}$ of length N, i.e. $s_j = 1$ if $j = m$ else $s_j = 0$, we can further rewrite this as

$$y_v = \frac{1}{N}\sum_{k=0}^{N-1} \mathrm{e}^{2\pi i \frac{k(v-1)}{N}}\, S_{k+1}\, X_{k+1} \tag{3.1a}$$

with

$$S_k = \sum_{m=0}^{N-1} s_{m+1}\, \mathrm{e}^{-2\pi i \frac{(k-1)m}{N}} , \quad X_k = \sum_{n=0}^{N-1} x_{n+1}\, \mathrm{e}^{-2\pi i \frac{(k-1)n}{N}} . \tag{3.1b}$$

This corresponds to a neural network with two hidden layers (one additive, one multiplicative) and an additive output layer as shown in fig. 3.1b. The optimal weights of this network are given by the corresponding coefficients from (3.1). This example can also be extended to patterns in two or more dimensions. A neural network without multiplications could still be trained on this task since any network with at least one hidden layer is an universal function approximator (Hornik et al., 1989). However, the number of neurons required might be significantly increased and it is unclear if good generalization would be possible, as no ideal weight assignment is known.



(a) (b)

Figure 3.1: (a) Variable pattern shift problem. Given a random binary pattern $\boldsymbol{x} \in \mathbb{R}^N$ and an integer $m \in \{0, 1, \ldots, N-1\}$ presented in one-hot encoding, the neural network should output the pattern $\boldsymbol{x}$ circularly shifted to the right by m grid cells. (b) An ANN with two hidden layers can solve this problem by employing the Fourier shift theorem. The first hidden layer is additive, the second is multiplicative and the output layer is additive. All neurons use linear activation functions. The first hidden layer computes the DFT of the input pattern $\boldsymbol{x}$ and shift mount $\boldsymbol{s}$. The second hidden layer applies the Fourier shift theorem by multiplying the DFTs of $\boldsymbol{x}$ and $\boldsymbol{s}$. The output layer computes the inverse DFT of the shifted pattern.

Polynomials

Considering the representability of a polynomial in one or multiple variables is important because any analytic function can be written as an (infinite) polynomial by means of its Taylor series (Swokowski, 1979). Since in practice infinite polynomials cannot be handled, the Taylor series is cut off after a number of terms, leading to an approximation of the function. Given a polynomial of degree d in n variables with k non-zero coefficients, it can be shown (Andoni et al., 2014; Barron, 1993, 1994) that it is representable by an additive neural network with a single hidden layer and $nkd^{\mathcal{O}(d)}$ hidden units. Note that d^d grows exponentially and thus for a moderate polynomial of degree seven, which is for example the degree of the Taylor series necessary to represent one period of the sine function reasonably well, this already leads to a factor of $7^7 \approx 800\,000$. A neural network using product units in its hidden layer can represent the same polynomial with k coefficients, regardless of the dimensionality of the input space and the degree of the polynomial.

3.2 Additive and Multiplicative Neurons

As described in section 2.5 in standard ANNs the value of a neuron is given by a weighted sum of its inputs propagated through a non-linear activation function. Considering a single layer of neurons $\boldsymbol{y}$ receiving inputs from a preceding layer denoted by $\boldsymbol{x}$, the value of $\boldsymbol{y}$ is given by

$$\boldsymbol{y} = \boldsymbol{\sigma}(W\boldsymbol{x})\,, \tag{3.2}$$

where we have absorbed the bias term into the weight matrix W by assuming that there exists an x_b with $x_b = 1$. In the context of this chapter we will call neural networks consisting of such layers additive ANNs.

In Durbin et al. (1989) an alternative neural unit in which the weighted summation is replaced by a product, where each input is raised to a power determined by its corresponding weight, was proposed. The value of such a *product unit* is given by

$$y_i = \sigma\left(\prod_j x_j^{W_{ij}}\right)\,. \tag{3.3}$$

Using laws of the exponential function this can be written as

$$y_i = \sigma\left[\exp\left(\sum_j W_{ij}\log x_j\right)\right]$$

and thus the values of such a layer can be computed efficiently using matrix multiplication, i.e.

$$\boldsymbol{y} = \boldsymbol{\sigma}\big(\exp(W\log\boldsymbol{x})\big) \tag{3.4}$$

where $\exp$ and $\log$ are taken elementwise. If the incoming values $\boldsymbol{x}$ can be negative, the complex exponential and logarithm must be used. Often no non-linearity is applied to the output of a product unit, i.e. $\boldsymbol{\sigma}(\boldsymbol{t}) = \boldsymbol{t}$.

We have seen that having both additive and multiplicative interaction in a neural networks is useful to solve complex problems. Yet this poses the problem of how to distribute additive and multiplicative units over the network, i.e. how to determine whether a specific neuron should be an additive or multiplicative unit to obtain the best results. For the tasks described in section 3.1 an optimal allocation could be determined by mathematical analysis of the problem; however such an analysis is not possible in general for complex problems.

We propose an approach, in which the distinction between additive and multiplicative

neurons is not discrete but continuous and differentiable, resulting in a continuum between addition and multiplication. Hence, the optimal distribution of additive and multiplicative units can be determined during standard gradient-based optimization. Our approach is organized as follows. First, we introduce non-integer iterates of the exponential function in the real and complex domains. This is based on the mathematical theory of fractional functional iteration. We then use these iterates to smoothly interpolate between addition (3.2) and multiplication (3.4). Finally, we show how this interpolation can be integrated and implemented in neural networks by means of a novel class of activation functions.

3.3 Iterates of the Exponential Function

Let $f : \mathbb{C} \to \mathbb{C}$ be an invertible function. For integer $n \in \mathbb{Z}$ we write $f^{(n)}$ for the n-times iterated application of f,

$$f^{(n)}(z) \triangleq \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}(z)\,. \tag{3.5}$$

Further let $f^{(-n)} = (f^{-1})^{(n)}$ where f^{-1} denotes the inverse of f. We set $f^{(0)}(z) = z$ to be the identity function. It can be easily verified that functional iteration with respect to the composition operator, i.e.

$$f^{(n)} \circ f^{(m)} = f^{(n+m)} \tag{3.6}$$

for $n, m \in \mathbb{Z}$, forms an Abelian group.

Equation (3.5) cannot be used to define functional iteration for non-integer n. Thus, in order to calculate non-integer iterations of a function, we have to find an alternative definition. The sought generalization should extend the additive property (3.6) of the composition operation to non-integer $n, m \in \mathbb{R}$.

3.3.1 Abel's Functional Equation

Consider the following functional equation given by (Abel, 1826),

$$\psi(f(x)) = \psi(x) + \beta \tag{3.7}$$

with constant $\beta \in \mathbb{C}$. We are concerned with $f(x) = \exp(x)$. A continuously differentiable solution (Kneser, 1950) for $\beta = 1$ and $x \in \mathbb{R}$ is given by

$$\psi(x) = \log^{(k)}(x) + k \tag{3.8}$$

with $k \in \mathbb{Z}$ s.t. $0 \leq \log^{(k)}(x) < 1$. Note that for $x < 0$ we have $k = -1$ and thus ψ is well defined on whole $\mathbb{R}$.

The function ψ is differentiable, as can be seen from the following argument (Köpp, 2015). Obviously ψ is differentiable at points where k does not change. At a point $x_0 = \exp^{(k)} 1$ where k changes, we have for the left derivative

$$\left.\frac{\partial_- \psi}{\partial x}\right|_{x=x_0} = \lim_{x \to x_0^-} \frac{\psi(x) - \psi(\exp^{(k)} 1)}{x - \exp^{(k)} 1} = \lim_{x \to x_0^-} \frac{\log^{(k)} x - 1}{x - \exp^{(k)} 1} \,. \tag{3.9}$$

Applying L'Hôpital's rule allows the limit to be resolved, resulting in

$$\frac{\partial_- \psi}{\partial x} = \lim_{x \to x_0^-} \prod_{j=0}^{k-1} \frac{1}{\log^{(j)} x} = \prod_{j=0}^{k-1} \frac{1}{\exp^{(k-j)} 1} \,. \tag{3.10}$$

For the right derivative we obtain

$$\left.\frac{\partial_+ \psi}{\partial x}\right|_{x=x_0} = \lim_{x \to x_0^+} \frac{\psi(x) - \psi(\exp^{(k)} 1)}{x - \exp^{(k)} 1} = \lim_{x \to x_0^+} \frac{\log^{(k+1)} x}{x - \exp^{(k)} 1} \,, \tag{3.11}$$

and, again by application of L'Hôpital's rule, this evaluates to

$$\frac{\partial_+ \psi}{\partial x} = \lim_{x \to x_0^+} \prod_{j=0}^{k} \frac{1}{\log^{(j)} x} = \prod_{j=0}^{k} \frac{1}{\exp^{(k-j)} 1} \,. \tag{3.12}$$

Since $\exp^{(k-k)} = \exp^{(0)}$ is the identity function, the left and right derivative are identical and thus the function ψ is differentiable at all points.

The function ψ is shown in Fig. 3.2a. Since $\psi : \mathbb{R} \to (-1, \infty)$ is strictly increasing, the inverse $\psi^{-1} :]-1, \infty[\to \mathbb{R}$ exists and is given by

$$\psi^{-1}(\psi) = \exp^{(k)}(\psi - k) \tag{3.13}$$

with $k \in \mathbb{N}$ s.t. $0 \leq \psi - k < 1$. Furthermore it follows that ψ^{-1} is differentiable. For practical reasons we set $\psi^{-1}(\psi) = -\infty$ for $\psi \leq -1$. The functions ψ and ψ^{-1} can be computed numerically using algorithms 4 and 5 respectively.

The derivative of ψ is given by

$$\psi'(x) = \prod_{j=0}^{k-1} \frac{1}{\log^{(j)}(x)} \tag{3.14a}$$

Algorithm 4: Computation of Abel's solution $\psi(x)$

Input: $x \in \mathbb{R}$
Output: $\psi \in \mathbb{R}$

```
if x < 0 then
    ψ ⟵ exp(x) − 1
else
    k ⟵ 0
    while x > 1 do
        x ⟵ log x
        k ⟵ k + 1
    ψ ⟵ x + k
```

Algorithm 5: Computation of Abel's solution inverse $\psi^{-1}(\psi)$

Input: $\psi \in \mathbb{R}$
Output: $x \in \mathbb{R} \cup \{-\infty\}$

```
k ⟵ ⌈ψ − 1⌉
if k < 0 then
    x ⟵ log(ψ − k)                    // negative infinity can occur
else
    x ⟵ ψ − k
    while k > 0 do
        x ⟵ exp x
        k ⟵ k − 1
```

with $k \in \mathbb{N}$ s.t. $0 \leq \log^{(k)}(x) < 1$ and the derivative of its inverse is

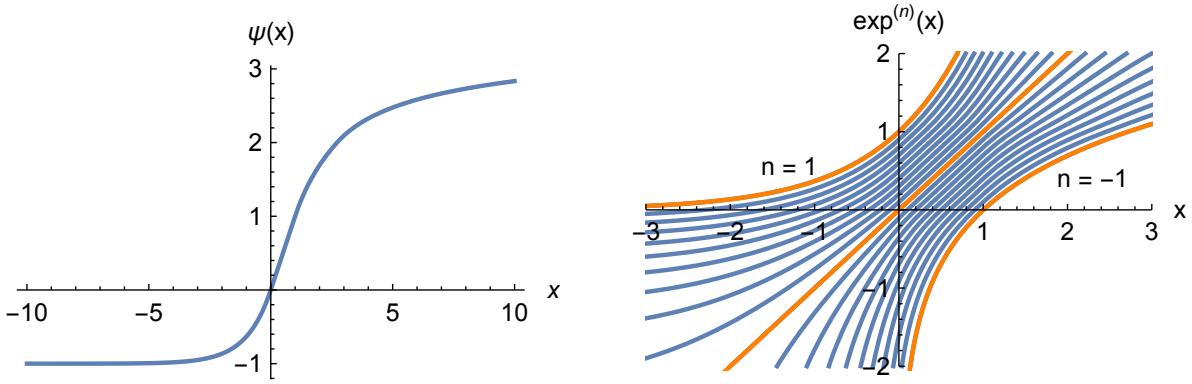
$$\psi^{-1'}(\psi) = \prod_{j=0}^{k-1} \exp^{(j)}\left(\psi^{-1}(\psi - j)\right) \tag{3.14b}$$

with $k \in \mathbb{N}$ s.t. $0 \leq \psi - k < 1$. Using an analogous proof as for the differentiability of ψ it can be shown that ψ' and $\psi^{-1'}$ are continuous.

Non-integer iterates using Abel's equation

By inspection of Abel's equation (3.7), we see that the n-th iterate of the exponential function can be written as

$$\exp^{(n)}(x) = \psi^{-1}(\psi(x) + n) \ . \tag{3.15}$$



(a) (b)

Figure 3.2: (a) A continuously differentiable solution $\psi(x)$ to Abel's equation (3.7) for the exponential function in the real domain. (b) Iterates of the exponential function $\exp^{(n)}(x)$ for $n \in \{-1, -0.9, \ldots, 0, \ldots, 0.9, 1\}$ obtained using the solution (3.15) of Abel's equation.

This family of functions is shown in fig. 3.2b. While the above equation is equivalent to (3.5) for integer n, we are now also free to choose $n \in \mathbb{R}$ and thus (3.15) can be seen as a generalization of functional iteration to non-integer iterates. It can easily be verified that the composition property (3.6) holds. Hence we can understand the function $\varphi(x) = \exp^{(1/2)}(x)$ as the function that gives the exponential function when applied to itself. φ is called the *functional square root* of $\exp$ and we have $\varphi(\varphi(x)) = \exp(x)$ for all $x \in \mathbb{R}$. Likewise $\exp^{(1/N)}$ is the function that gives the exponential function when iterated N times.

Since n is a continuous parameter in definition (3.15) we can take the derivative of $\exp$ with respect to its argument as well as n. They are given by

$$\exp'^{(n)}(x) \triangleq \frac{\partial \exp^{(n)}(x)}{\partial x} = \psi'^{-1}(\psi(x) + n)\, \psi'(x) \tag{3.16a}$$

$$\exp^{(n')}(x) \triangleq \frac{\partial \exp^{(n)}(x)}{\partial n} = \psi'^{-1}(\psi(x) + n)\ . \tag{3.16b}$$

Thus (3.8) provides a method to interpolate between the exponential function, the identity function and the logarithm in a continuous and differentiable way.

3.3.2 Schröder's Functional Equation

The method presented so far only works for $x > 0$ when n is arbitrary since the real logarithm is undefined for zero and negative arguments. Motivated by the necessity to calculate *negative* fractional iterates for negative arguments as well, we derive a solution of Abel's equation for

the *complex* exponential function. Applying the substitution

$$\psi(x) \triangleq \frac{\beta}{\log \gamma} \log \chi(x)$$

in Abel's equation (3.7) produces Schröder's function equation, first examined by Schröder (1870) and analyzed in detail by Kneser (1950),

$$\chi(f(z)) = \gamma\, \chi(z) \tag{3.17}$$

with constant $\gamma \in \mathbb{C}$. As before we are interested in solutions of this equation for $f(x) = \exp(x)$. We have

$$\chi(\exp(z)) = \gamma\, \chi(z) \tag{3.18}$$

but now we are considering the complex $\exp : \mathbb{C} \to \mathbb{C}$. The complex exponential function is not injective, since

$$\exp(z + 2\pi n i) = \exp(z)\,, \qquad n \in \mathbb{Z}\,,$$

where $i \triangleq \sqrt{-1}$. Thus the imaginary part of the codomain of its inverse, i.e. the complex logarithm, must be restricted to an interval of size 2π. Here we define $\log : \mathbb{C} \to \{z \in \mathbb{C} : \beta \leq \operatorname{Im} z < \beta + 2\pi\}$ with $\beta \in \mathbb{R}$. For now let us consider the principal branch of the logarithm, that is $\beta = -\pi$.

To derive a solution, we examine the behavior of the exponential function around one of its fixed points. A fixed point of a function f is a point c with the property that $f(c) = c$. The exponential function has an infinite number of fixed points (Agarwal, 2001); here we select the fixed point closest to the real axis in the upper complex half plane. Since $\log$ is a contraction mapping, according to the Banach fixed-point theorem (Khamsi et al., 2001) the fixed point of $\exp$ can be found by starting at an arbitrary point $z \in \mathbb{C}$ with $\operatorname{Im} z \geq 0$ and repetitively applying the logarithm until convergence. Numerically we find

$$\exp(c) = c \approx 0.318132 + 1.33724\, i$$

where $i = \sqrt{-1}$ is the imaginary unit.

Close enough to c the exponential function behaves like an affine map. To show this, let $z' = z - c$ and consider

$$\begin{aligned} \exp(c + z') - c &= \exp(c)\exp(z') - c = c\,[\exp(z') - 1] \\ &= c\,[1 + z' + O(|z'|^2) - 1] = c\,z' + O(|z'|^2)\,. \end{aligned} \tag{3.19}$$

Here we used the Taylor expansion of the exponential function, $\exp(z') = 1 + z' + O(z'^2)$. This means that points on a circled centered on c with radius $r \ll 1$ are mapped to a circle around c with radius $|c|\, r$ and rotated around c by $\operatorname{Im} c \approx 76.62°$. For any point z in a circle of radius r_0 around c, we have

$$\exp(z) = cz + c - c^2 + O(r_0^2)\,. \tag{3.20}$$

By substituting this approximation into (3.18) it becomes apparent that a solution to Schröder's equation around c is given by

$$\chi(z) = z - c \quad \text{for } |z - c| \leq r_0 \tag{3.21}$$

where we have set $\gamma = c$.

We will now compute the continuation of the solution to points outside the circle around c. From (3.18) we obtain

$$\chi(z) = c\,\chi(\log(z))\,. \tag{3.22}$$

If for a point $z \in \mathbb{C}$ repeated application of the logarithm leads to a point inside the circle of radius r_0 around c, we can obtain the function value of $\chi(z)$ from (3.21) via iterated application of (3.22). We will show later that this is indeed the case for nearly every $z \in \mathbb{C}$. Hence the solution to Schröder's equation is given by

$$\chi(z) = c^k\,(\log^{(k)}(z) - c) \tag{3.23}$$

with $k = \min_{k' \in \mathbb{N}} k'$ s.t. $|\log^{(k')}(z) - c| \leq r_0$. Solving for z gives

$$\chi^{-1}(\chi) = \exp^{(k)}(c^{-k}\chi + c) \tag{3.24}$$

with $k = \min_{k' \in \mathbb{N}} k'$ s.t. $|c^{-k'}\chi| \leq r_0$. Obviously we have $\chi^{-1}(\chi(z)) = z$ for all $z \in \mathbb{C}$. However $\chi(\chi^{-1}(\xi)) = \xi$ only holds if $\operatorname{Im}(c^{-k}\xi + c) \in [\beta, \beta + 2\pi[$. The solution (3.23) is only defined for a point z if iterated application of the logarithm starting from z will converge to the fixed point c. We will show below which conditions are necessary for that.

Kneser (1950) demonstrated that χ is holomorphic on $\mathbb{C} \setminus \{0, 1, \mathrm{e}, \mathrm{e}^{\mathrm{e}}, \dots\}$. The complex derivative of χ evaluates to

$$\chi'(z) = \prod_{j=0}^{k-1} \frac{c}{\log^{(j)} z} \tag{3.25a}$$

with $k = \min_{k' \in \mathbb{N}} k'$ s.t. $|\log^{(k')}(z) - c| \leq r_0$ and we have

$$\chi'^{-1}(\chi) = \frac{1}{c} \prod_{j=1}^{k} \exp^{(j)}\Big(\chi^{-1}\Big(\frac{\chi}{c^j}\Big)\Big) \tag{3.25b}$$

with $k = \min_{k' \in \mathbb{N}} k'$ s.t. $|c^{-k'}\chi| \leq r_0$.

The solution χ is defined on almost $\mathbb{C}$

The principal branch of the logarithm, i.e. restricting its imaginary part to the interval $[-\pi, \pi[$, has the drawback that iterated application of $\log$ starting from a point on the lower complex half-plane will converge to the complex conjugate $\bar{c}$ instead of c. Thus $\chi(z)$ would be undefined for $\operatorname{Im} z < 0$.

To avoid this problem, we use the branch defined by $\log : \mathbb{C} \to \{z \in \mathbb{C} : \beta \leq \operatorname{Im} z < \beta + 2\pi\}$ with $-1 < \beta < 0$. Using such a branch the series z_n, where

$$z_{n+1} \triangleq \log z_n \,,$$

converges to c, provided that there is no n such that $z_n = 0$. Thus χ is defined on $\mathbb{C} \setminus D$ where $D = \{0, \mathrm{e}, \mathrm{e}^{\mathrm{e}}, \mathrm{e}^{\mathrm{e}^{\mathrm{e}}}, \dots\}$.

If $\operatorname{Im} z_n \geq 0$ then $\arg z_n \in [0, \pi]$ and thus $\operatorname{Im} z_{n+1} \geq 0$. Hence, if we have $\operatorname{Im} z_n \geq 0$ for $n \in \mathbb{N}$, then $\operatorname{Im} z_{n'} \geq 0$ for all $n' > n$. Now, consider the conformal map (Kneser, 1950)

$$\xi(z) \triangleq \frac{z - c}{z - \bar{c}}$$

which maps the upper complex half-plane to the unit disk and define the series $\xi_{n+1} = \zeta(\xi_n)$ with

$$\zeta(t) \triangleq \xi\big(\log \xi^{-1}(t)\big) \,.$$

We have $\zeta : D_1 \to D_1$, where $D_1 = \{t \in \mathbb{C} : |t| < 1\}$ is the unit disk; furthermore $\zeta(0) = 0$. Thus by Schwarz lemma $|\zeta(t)| < |t|$ for all $t \in D_1$ (since $\zeta(t) \neq \lambda t$ with $\lambda \in \mathbb{C}$) and hence $\lim_{n\to\infty} \xi_n = 0$. This implies $\lim_{n\to\infty} z_n = c$.

On the other hand, if $\operatorname{Im} z_n < 0$ and $\operatorname{Re} z_n < 0$, then $\operatorname{Im} \log z_n > 0$ and z_n converges as above. Finally, if $\operatorname{Im} z_n < 0$ and $\operatorname{Re} z_n \geq 0$, then, using $-1 < \beta$, we have $\operatorname{Re} z_{n+1} \leq |\log z_n| \leq 1 + \log(\operatorname{Re} z_n) < \operatorname{Re} z_n$ and thus at some element n' in the series we will have $\operatorname{Re} z_{n'} < 1$ which leads to $\operatorname{Re} z_{n'+1} < 0$.

Algorithm and error estimates for numeric computation

Numerical computation of χ and its inverse is possible using algorithms 6 and 7 respectively. The computation procedure is visualized in fig. 3.3a and $\chi(z)$ is plotted in fig. 3.4. The number of loop iterations to compute $\chi(z)$ for an arbitrary point $z \in \mathbb{C}$ is on average 30; a detailed plot of the iteration count is shown in fig. 3.3b. Since the algorithms are computationally intensive, it is advisable to precompute $\chi(z)$ and $\chi^{-1}(\chi)$ in forms of interpolation tables.

Algorithm 6: Computation of a solution $\chi(z)$ to Schröder's equation

Input: $z \in \mathbb{C} \setminus \{0, 1, \mathrm{e}, \mathrm{e}^{\mathrm{e}}, \dots\}$
Output: $\chi \in \mathbb{C}$

1 $c \longleftarrow 0.318132 + 1.33724\,i$ // fixed point of exp
2 $\rho \longleftarrow 1$
3 **while** $|z - c| \geq r_0$ **do** // $r_0 \ll 1$
4 $\quad z \longleftarrow \log z$
5 $\quad \rho \longleftarrow c\,\rho$
6 $\chi \longleftarrow \rho(z - c)$

Algorithm 7: Computation of the inverse $\chi^{-1}(\chi)$ of a solution to Schröder's equation

Input: $\chi \in \mathbb{C}$
Output: $z \in \mathbb{C}$

1 $c \longleftarrow 0.318132 + 1.33724\,i$ // fixed point of exp
2 $k \longleftarrow 0$
3 **while** $|\chi| \geq r_0$ **do** // $r_0 \ll 1$
4 $\quad \chi \longleftarrow \frac{\chi}{c}$
5 $\quad k \longleftarrow k + 1$
6 $z \longleftarrow \chi + c$
7 **while** $k > 0$ **do**
8 $\quad z \longleftarrow \exp z$
9 $\quad k \longleftarrow k - 1$

Computing $\chi(z)$ is unproblematic with regards to numeric accuracy since the logarithm, which is iteratively applied in line 4 of algorithm 6, is a contractive map and thus numeric errors are not amplified with each application. The computation of $\chi^{-1}(\chi)$, however seems vulnerable to numeric instabilities since the exponential function is iteratively applied in line 8 of algorithm 7. We therefore derive an estimation of how much an error in χ is amplified when computing z using algorithm 7. First we note that lines 4 and 6 do not increase an error on χ

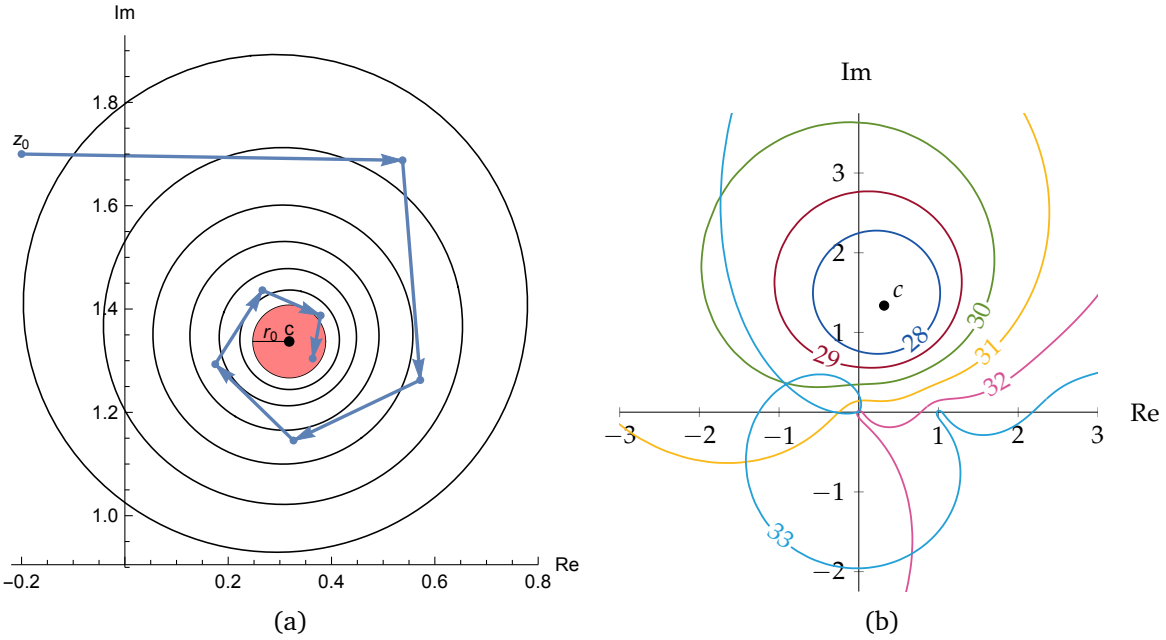


Figure 3.3: (a) Calculation of $\chi(z)$. Starting from point $z_0 = z$ the series $z_{n+1} = \log z_n$ is evaluated until $|z_n - c| \leq r_0$ for some n. Inside this circle of radius r_0 the function value can then be evaluated using $\chi(z) = c^n(z_n - c)$. The contours are generated by iterative application of $\exp$ to the circle of radius r_0 around c. Near its fixed point the exponential behaves like a scaling by $|c| \approx 1.374$ and a rotation of $\operatorname{Im} c \approx 76.6°$ around c. (b) Number of iterated applications of $\exp$ required to cover a marked area when starting from a point very close to the fix point c. This roughly equals the number of loop iterations in algorithm 6 necessary to compute $\chi(z)$ for a point z on the complex plane. Starting with 30 iterations the areas start to intersect since the complex exponential is not injective. This part has been reproduced from (Köpp, 2016).

since $|c| > 1$. Thus the increase of error comes from application of $\exp^{(n)}$ in line 8. It is given by

$$\Delta = \frac{\left|\exp^{(n)}(z+\delta) - \exp^{(n)} z\right|}{|\delta|}$$

where δ is the error of z and the for the initial z we have $|z - c| \leq r_0$. An estimation of the error factor Δ can be obtained by calculating how much area of the complex plane is covered by iterative applications of the exponential to the points of a circle with radius r_0 around c, i.e.

$$\Delta \approx \sqrt{\frac{\operatorname{area}(\{\exp^{(n)} z \mid z \in \mathbb{C} \wedge |z - c| \leq r_0\})}{2\pi r_0}}\,. \tag{3.26}$$

The assumption here is that an error will be scaled like the area of the complex plane it is contained in. As noted before, around c the exponential acts like an affine transform that

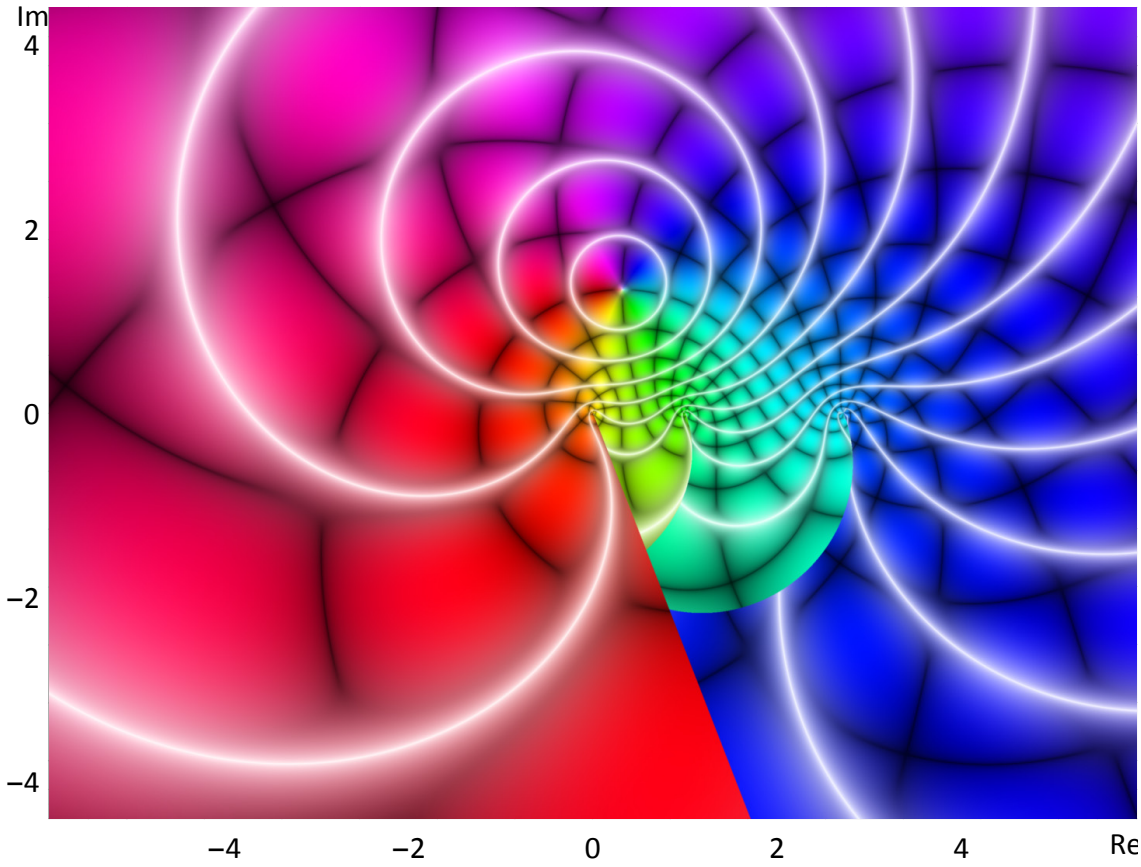


Figure 3.4: Domain coloring plot of $\chi(z)$. Domain coloring (Wegert, 2012) refers to the visualization of complex-valued functions based on the HSB-color model. HSB stands for hue, saturation and brightness and each value can vary between 0 and 1. The hue is used to represent the phase φ of the function value $f(z) = r\,\mathrm{e}^{i\varphi}$, the saturation encodes the magnitude r and for brightness the real and imaginary parts are combined. The exact computation for the latter two involves taking the sine of either $f(z)$ or its real or imaginary part, leading to repetitive patterns. For $\chi(z)$ the discontinuities that arise at 0, 1, e and stretch into the negative complex half-plane are clearly visible. They are caused by $\log$ being discontinuous at the polar angles β and $\beta + 2\pi$. The fixed point c of the exponential function can also be seen as well as the affine behavior of the exponential around it.

n	Δ_{circ}	Δ_{area}	Δ_{num}
0	1	1	—
1	1.375	1.375	—
2	1.889	1.889	—
⋮	⋮	⋮	⋮
26	3 910	3 978	3 949
27	5 375	5 552	5 463
28	7 389	7 860	7 604
29	10 156	11 441	10 764
30	13 960	17 621	15 733
31	19 189	30 623	24 504
32	26 376	70 222	43 765
33	36 255	328 253	131 981

Table 3.1: Average relative errors of $\exp^{(n)} z$ for $|z - c| \leq r_0$ and different number of iterations n, where the coverage of the iterations can be seen from fig. 3.3b. The values for Δ_{circ} were computed using eq. (3.27) and the values for Δ_{area} were computed using eq. (3.26) where the area was computed numerically. Starting with 30 iterations the circle approximation does not hold anymore. Numerical error estimates obtained using the arbitrary precision calculation feature of Mathematica are shown in column Δ_{num}. This table has been adapted from (Köpp, 2015).

multiplies distances $|z - c|$ with a factor of $|c| \approx 1.375$, cf. fig. 3.3a, and from fig. 3.3b we see that this approximation is reasonable for up to 30 iterations of exponentiation. Thus for $n < 30$ we obtain the explicit formula

$$\Delta_{\text{circ}} \approx \sqrt{\frac{\pi(|c|^n\, r_0)^2}{\pi r_0^2}} = |c|^n \ . \tag{3.27}$$

For $n \geq 30$ we turn to estimating the area by numerically computing the area of the set $\{\exp^{(n)} z \mid z \in \mathbb{C} \wedge |z - c| \leq r_0\}$ on the complex plane. Results were calculated by Köpp (2015) and are shown as Δ_{area} in table 3.1. For verification the error has also been calculated numerically using the arbitrary precision calculation feature of the software package Mathematica (Wolfram Research, 2017).

To cover the interesting part of $\mathbb{C}$ we need 33 iterations and thus a conservative estimate for the relative error factor of $\chi^{-1}(\chi)$ is 10^6. Since the machine epsilon (Quarteroni, 2000) for 32-bit single precision floating-point operations is 10^{-7}, we conclude that 64-bit double precision arithmetic with a machine epsilon of $2 \cdot 10^{-16}$ must be used to calculate χ^{-1} with reasonable accuracy.

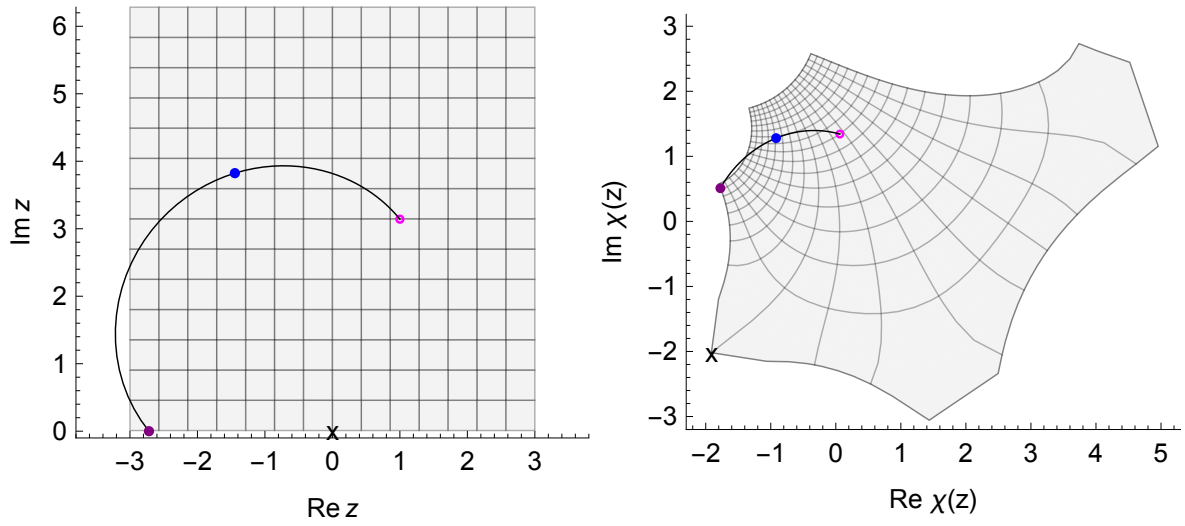


Figure 3.5: Structure of the function $\chi(z)$ and calculation of $\exp^{(n)}(1+\pi i)$ for $n \in [0,1]$ in z-space (left) and χ-space (right). The uniform grid with the cross at the origin is mapped using $\chi(z)$. Since the map is conformal, local angles are preserved. The points $1+\pi i$ and $\xi = \chi(1+\pi i)$ are shown as magenta circles. The black line shows $\chi^{-1}(c^n\,\xi)$ and $c^n\,\xi$ for $n \in [0,1]$. The blue and purple points are placed at $n = 1/2$ and $n = 1$ respectively.

Non-integer iterates using Schröder's equation

Repetitive application of Schröder's equation (3.17) on an iterated function (3.5) leads to

$$\chi(f^{(n)}(z)) = \gamma^n\,\chi(z)\,. \tag{3.28}$$

Thus the n-th iterate of the exponential function on the whole complex plane is given by

$$\exp^{(n)}(z) = \chi^{-1}(c^n\,\chi(z)) \tag{3.29}$$

where $\chi(z)$ and $\chi^{-1}(z)$ are given by (3.23) and (3.24) respectively. Since χ is injective, we can think of it as a mapping from the complex plane, called z-plane, to another complex plane, called χ-plane. By (3.29) the operation of calculating the exponential of a number y in the z-plane corresponds to complex multiplication by factor c of $\chi(y)$ in the χ-plane. This is illustrated in fig. 3.5. Samples from $\exp^{(n)}$ are shown in fig. 3.6.

While the definition for $\exp^{(n)}$ given by (3.29) can be evaluated on the whole complex plane $\mathbb{C}$, it only has meaning as a non-integer iterate of $\exp$, if composition $\exp^{(n)}[\exp^m(z)] =$

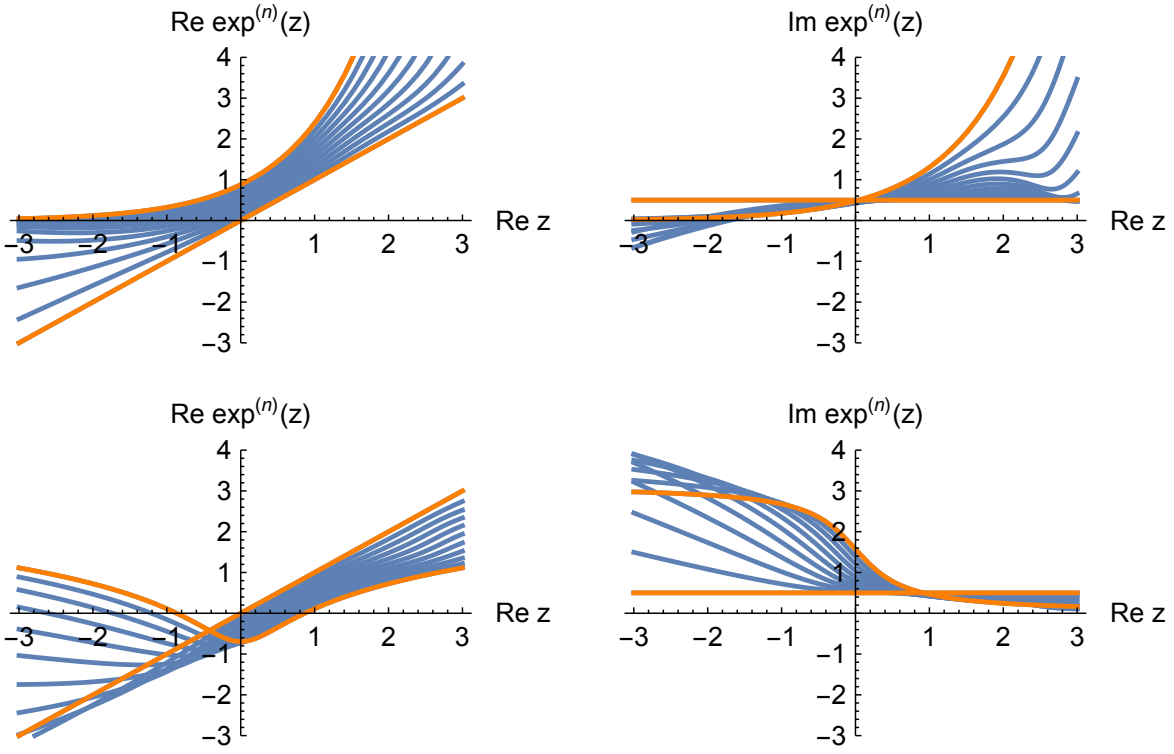


Figure 3.6: Iterates of the exponential function $\exp^{(n)}(x + 0.5i)$ for $n \in \{0, 0.1, \ldots, 0.9, 1\}$ (upper plots) and $n \in \{0, -0.1, \ldots, -0.9, -1\}$ (lower plots) obtained using the solution (3.29) of Schröder's equation. The exponential, logarithm and identity function are highlighted in orange.

$\exp^{(n+m)}(z)$ holds. Since this requires that $\chi(\chi^{-1}(\xi)) = \xi$, let us define the sets

$$\mathcal{E}' \triangleq \{\xi \in \mathbb{C} \,|\, \chi[\chi^{-1}(c^m\,\xi)] = c^m\,\xi \quad \forall m \in [-1, 1]\}$$

and $\mathcal{E} \triangleq \chi^{-1}(\mathcal{E}')$. Then, for $z \in \mathcal{E}$, $n \in \mathbb{R}$ and $m \in [-1, 1]$, the composition of the iterated exponential function is given by

$$\begin{aligned}\exp^{(n)}[\exp^{(m)}(z)] &= \chi^{-1}\left[c^n\,\chi\big(\chi^{-1}(c^m\,\chi(z))\big)\right] \\ &= \chi^{-1}\left[c^{n+m}\,\chi(z)\right] = \exp^{(n+m)}(z)\end{aligned}$$

and the composition property is satisfied. The subset $\mathcal{E} \subset \mathbb{C}$ of the complex plane where composition of $\exp^{(n)}$ for non-integer n is shown in figure 3.7.

The derivatives of $\exp^{(n)} x$, defined using Schröder's equation, w.r.t. x and n can be calcu-

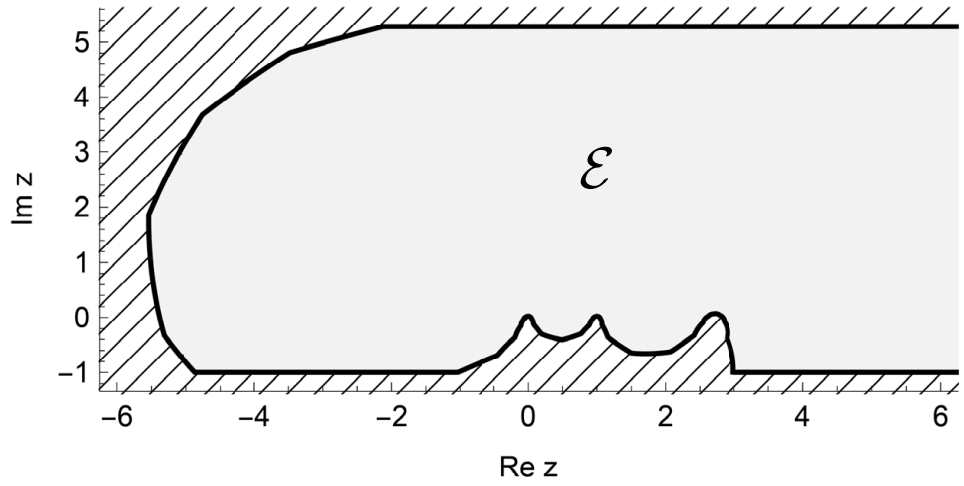


Figure 3.7: Function composition holds in the set $\mathcal{E} \subset \mathbb{C}$ (gray-shaded area) for non-integer iteration numbers, there we have $\exp^{(n)} \circ \exp^{(m)}(z) = \exp^{(n+m)}(z)$ for $n \in \mathbb{R}$ and $m \in [-1, 1]$. We defined $\log$ such that $\mathrm{Im}\, \log z \in [-1, -1 + 2\pi]$. In the striped area function composition does not hold, but $\exp^{(n)}$ can still be computed.

lated using the chain rule and evaluate to

$$\exp'^{(n)}(z) = c^n \, \chi'^{-1}[c^n \chi(z)] \, \chi'(z) \tag{3.30a}$$

$$\exp^{(n')}(z) = c^n \, \chi'^{-1}[c^n \chi(z)] \, \chi(z) \log(c) \,. \tag{3.30b}$$

In conclusion, this section defined the continuously differentiable function $\exp^{(n)} : \mathbb{C} \setminus D \to \mathbb{C}$ on almost the whole complex plane and showed that it has the meaning of a non-integer iterate of the exponential function on the subset $\mathcal{E}$.

3.4 Interpolation between Addition and Multiplication

Using fundamental properties of the exponential function we can write every multiplication of two numbers $x, y \in \mathbb{R}$ as

$$xy = \exp(\log x + \log y) = \exp(\exp^{(-1)} x + \exp^{(-1)} y) \,.$$

If $x < 0$ or $y < 0$ then the complex logarithm and exponential function must be used. We define the operator $\oplus_n$ for $x, y \in \mathbb{R}$ and $n \in \mathbb{R}$ as

$$x \oplus_n y \triangleq \exp^{(n)}\Big(\exp^{(-n)}(x) + \exp^{(-n)}(y)\Big) \,. \tag{3.31}$$

Note that we have $x \oplus_0 y = x + y$ and $x \oplus_1 y = xy$. Thus for $0 < n < 1$ the above operator continuously interpolates between the elementary operations of addition and multiplication. We will refer to $\oplus_n$ as the "addiplication" operator. Analogous to the n-ary sum and product we will employ the following notation for the n-ary addiplication operator,

$$\bigoplus_{\substack{j=k \\ n}}^{K} x_j \triangleq x_k \oplus_n x_{k+1} \oplus_n \cdots \oplus_n x_K \,. \tag{3.32}$$

The derivative of the addiplication operator w.r.t. its operands and the interpolation parameter n are calculated using the chain rule. Using the shorthand

$$E \triangleq \exp^{(-n)}(x) + \exp^{(-n)}(y) \tag{3.33a}$$

we have

$$\frac{\partial(x \oplus_n y)}{\partial x} = \exp'^{(n)}(E)\ \exp'^{(-n)}(x)\,, \tag{3.33b}$$

$$\frac{\partial(x \oplus_n y)}{\partial y} = \exp'^{(n)}(E)\ \exp'^{(-n)}(y)\,, \tag{3.33c}$$

$$\frac{\partial(x \oplus_n y)}{\partial n} = \exp^{(n')}(E) + \exp'^{(n)}(E) \cdot \left[\exp^{(-n')}(x) + \exp^{(-n')}(y)\right]\,. \tag{3.33d}$$

For positive arguments $x, y > 0$ we can use the iterates of $\exp$ based either on the solution of Abel's equation (3.15) or Schröder's equation (3.29). However, if we also want to deal with negative arguments, we must use iterates of $\exp$ based on Schröder's equation (3.29), since the real logarithm is only defined for positive arguments. Thus we will call solution (3.15) the "real fractional exponential" and solution (3.29) the "complex fractional exponential".

From the exemplary "addiplication" shown in fig. 3.8 we can see that the interpolations produced by these two methods are not monotonic functions w.r.t. the interpolation parameter n. In both cases local maxima exist, however interpolation based on Schröder's equation has higher extrema in this case and also in general (personal experiments). It is well known, that the existence of local extrema can pose a problem for gradient-based optimizers.

3.5 Neurons that can Add, Multiply and Everything In-Between

We propose two methods to construct neural nets that have units the operation of which can be adjusted using a continuous parameter. The straightforward approach is to use neurons that

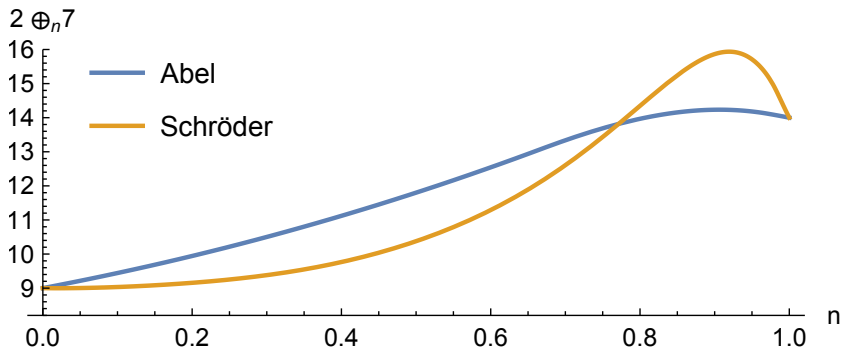


Figure 3.8: The interpolation between addition and multiplication using (3.31) with $x = 2$ and $y = 7$. The iterates of the exponential function are either calculated using Abel's equation (blue) or Schröder's equation (orange). In both cases the interpolated values exceed the range between $2 + 7 = 9$ and $2 \cdot 7 = 14$ and therefore a local maximum exists. However in the Abel case the local maximum is not very pronounced.

employ "addiplication" instead of summation, i.e. the value of neuron y_i is given by

$$y_i = \sigma\left(\bigoplus_{\substack{j \\ n_i}} W_{ij} x_j\right) = \sigma\left[\exp^{(n_i)}\left(\sum_j W_{ij} \exp^{(-n_i)}(x_j)\right)\right] . \tag{3.34}$$

where the operator $\oplus$ has been defined in section 3.4. For $n_i = 0$ the neuron behaves like an additive neuron and for $n_i = 1$ it computes the product of its inputs. Because we sum over $\exp^{(-n_i)}(x_j)$ which has dependence on the parameter n_i of neuron y_i, this calculation corresponds to a network in which each neuron in layer $\boldsymbol{x}$ has separate outputs for each neuron in the following layer $\boldsymbol{y}$; see fig. 3.9a. Compared to conventional ANNs this architecture has only one additional real-valued parameter per neuron (n_i) but also poses a significant increase in computational complexity due to the necessity of separate outputs. Since $\exp^{(-n_i)}(x_j)$ is complex it might be sensible (but is not required) to allow a complex weight matrix W_{ij}.

The computational complexity of separate output units can be avoided by calculating the value of a neuron according to

$$y_i = \sigma\left[\exp^{(\hat{n}_{y_i})}\left(\sum_j W_{ij} \exp^{(\widetilde{n}_{x_j})}(x_j)\right)\right] . \tag{3.35}$$

This corresponds to the architecture shown in fig. 3.9b. The interpolation parameter n_i has been split into a pre-activation-function part $\hat{n}_{y_i}$ and a post-activation-function part $\widetilde{n}_{x_j}$. Since $\hat{n}_{y_i}$ and $\widetilde{n}_{x_j}$ are not tied together, the network is free to implement arbitrary combinations of iterates

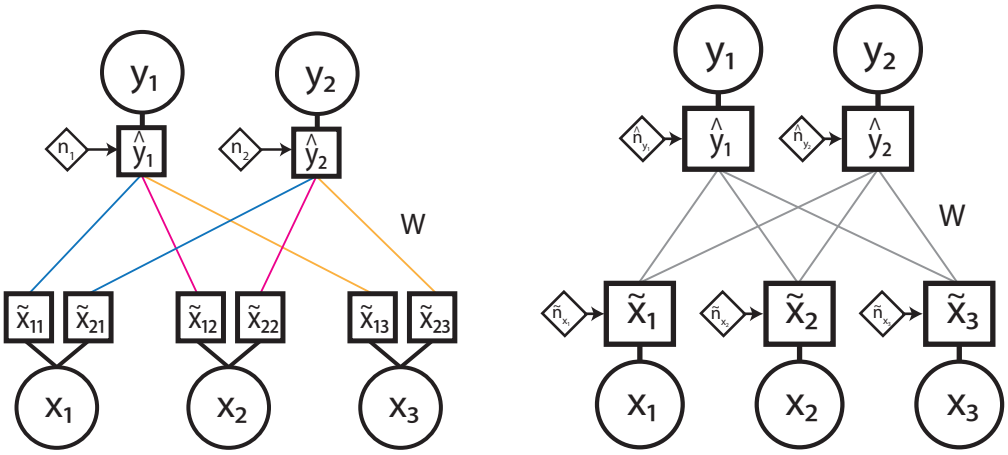


Figure 3.9: Two proposed neural network architectures that can implement "addiplication". (a) Neuron y_i calculates its value according to (3.34). We have $y_i = \sigma(\hat{y}_i)$ and the subunits compute $\widetilde{x}_{ij} = \exp^{(-n_i)}(x_j)$ and $\hat{y}_i = \exp^{(n_i)}\left(\sum_j W_{ij}\widetilde{x}_{ij}\right)$. The weights between the subunits are shared. (b) Neuron y_i calculates its value according to (3.35). We have $y_i = \sigma(\hat{y}_i)$ and the subunits compute $\widetilde{x}_j = \exp^{(-\widetilde{n}_j)}(x_j)$ and $\hat{y}_i = \exp^{(\hat{n}_i)}\left(\sum_j W_{ij}\widetilde{x}_j\right)$. This matches the conventional neural network architecture, the only difference being the parameterized activation function individual to each neuron.

of the exponential function. "Addiplication" occurs as the special case $\hat{n}_{y_i} = -\widetilde{n}_{x_j}$. Compared to conventional neural nets each neuron has two additional parameters, namely $\hat{n}_{y_i}$ and $\widetilde{n}_{x_j}$. However the asymptotic computational complexity of the network is unchanged. In fact, this architecture corresponds to a conventional, additive neural net, as defined by (2.87), with a neuron-dependent, parameterized activation function. For neuron z_i the activation function given by

$$\sigma_{z_i}(t) = \exp^{(\widetilde{n}_{z_i})}\left[\sigma\left(\exp^{(\hat{n}_{z_i})}(t)\right)\right] . \tag{3.36}$$

where σ is a standard activation function like the logistic function or just the identity. Using the identity does not pose a problem, because even then σ_{z_i} is not linear for $z_i \neq 0$. Consequently, implementation in existing neural network frameworks is possible by replacing the standard sigmoidal activation function with this function and optionally using a complex weight matrix.

3.6 Benchmarks and Experimental Results

Implementation and experiments were done by Köpp (2015) under my guidance; the corresponding results are reproduced here in excerpts.

We investigate how ANNs using fractional iterates of the exponential function as activa-

tion functions and therefore interpolation between summation and multiplication perform on different datasets. Alongside standard gradient descent we apply adaptive optimizers such as Adam (D. Kingma et al., 2014), RMSProp (Tieleman et al., 2012) or Adadelta (Zeiler, 2012) which are provided by Climin (Bayer, 2013) for simple use with the Theano software package (Al-Rfou et al., 2016). An overview over these algorithms is given in section 2.3.

To achieve good performance the functions ψ, χ and their inverses, which are used to calculate the fractional exponential, have been implemented using fast linear interpolation over a precomputed lookup tabled stored in texture memory on the graphics processing unit (GPU) and accessed through Compute Unified Device Architecture (CUDA)-provided hardware interpolation operations.

3.6.1 Recognition of Handwritten Digits

We perform multi-class logistic regression with one hidden layer on the MNIST dataset (Lecun et al., 1998). It contains $60\,000$ images of handwritten digits zero to nine in its training set and $10\,000$ images in its test set. Furthermore $10\,000$ samples of the training set are split off for use as a validation set. Each image has 784 grayscale pixels, which are used as raw values for the input layer.

We compare a conventional additive ANN to ANNs using the real- and complex-valued fractional exponential function as follows. The ANN with the fractional exponential function uses complex weights, stored as real and imaginary part. Thus, in order for all networks to have approximately the same number of trainable parameters, the additive network and the network using the real fractional exponential function use 200 hidden units while the ANN with the complex fractional exponential function has 100 hidden units. The additional parameters $\widetilde{n}$ and $\hat{n}$ do not affect the total parameter count significantly, since their number scales only linearly with the number of neurons. The setup resembles the one used for benchmarking Theano (Bergstra et al., 2010).

All networks are trained with stochastic gradient descent using an initial learning rate of 10^{-1}, a momentum of 0.5, and a mini-batch size of 60 samples. The learning rate is reduced whenever the validation loss does not improve for 100 consecutive iterations. Initially, all weights are set to zero with exception of the real part of the weights within the hidden layer. These are initialized by randomly sampling from the uniform distribution over the interval

$$\left[-4\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, 4\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right]$$

following a suggestion from Glorot, Bordes, et al. (2011) for weight initialization in ANNs

ANN type	num. parameters	samples per sec.	iterations	error rate
additive	159 010	32 387	283	0.0203
$\mathbb{R}$ frac. exp.	159 994	21 253	245	0.0253
$\mathbb{C}$ frac. exp.	159 894	17 147	263	0.0267

Table 3.2: Results for using ANNs of different types for classification of handwritten digits. Training was terminated when no improvement was seen on the validation set. The error rate denotes the fraction of how many digits in the test set have been wrongly classified after training.

with a sigmoid activation function, where n_{in} and n_{out} denote the number of input and output connections respectively for the considered neuron.

Table 3.2 shows the results for training the three networks until no improvement happens on the validation set. Networks using interpolation between summation and multiplication achieve competitive though not surpassing performance on the MNIST dataset. Although their activation function is more complicated than the standard sigmoid, their execution performance (processed samples per second) only suffers by a factor of $1/2$. Figure 3.10 displays a histogram for the iteration parameters $\widetilde{n}$ and $\hat{n}$ in the interpolating complex-valued ANN at the iteration with the best validation loss. While many of the parameters remain close to initialization value zero, some have changed notably and thus the trained ANN performs operations beyond pure additive interactions, and even super-exponential computations for some neurons.

3.6.2 Synthetic Polynomial Regression

In order to analyze long-range generalization performance, we use a polynomial of fourth degree with two inputs and a dataset with a non-random split into training and test set. The function to be learned is of the form

$$f(x, y) = \sum_{k=0}^{4} \sum_{l=0}^{4-k} a_{kl}\, x^k\, y^l\,, \tag{3.37}$$

with two inputs x and y and randomly generated coefficients a_{kl}. The polynomial can be computed exactly by an ANN with one hidden layer. The hidden layer is multiplicative and calculates the products between inputs and the output layer is additive using the coefficients a_{kl} as weights. We randomly sample 600 input values (x, y) from the uniform distribution with support $[0, 1]^2$ and compute targets according to (3.37). All points within a circle of radius 0.33 around $x = 0.5, y = 0.5$ are used as the test set and all points outside that circle make up the training set, see fig. 3.11a. This is done to test the long-range generalization ability of the ANN.

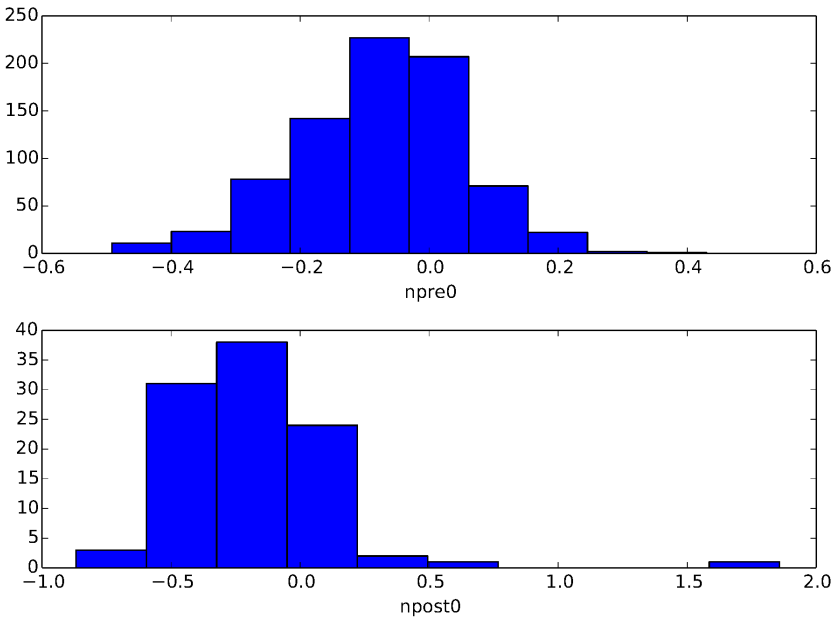
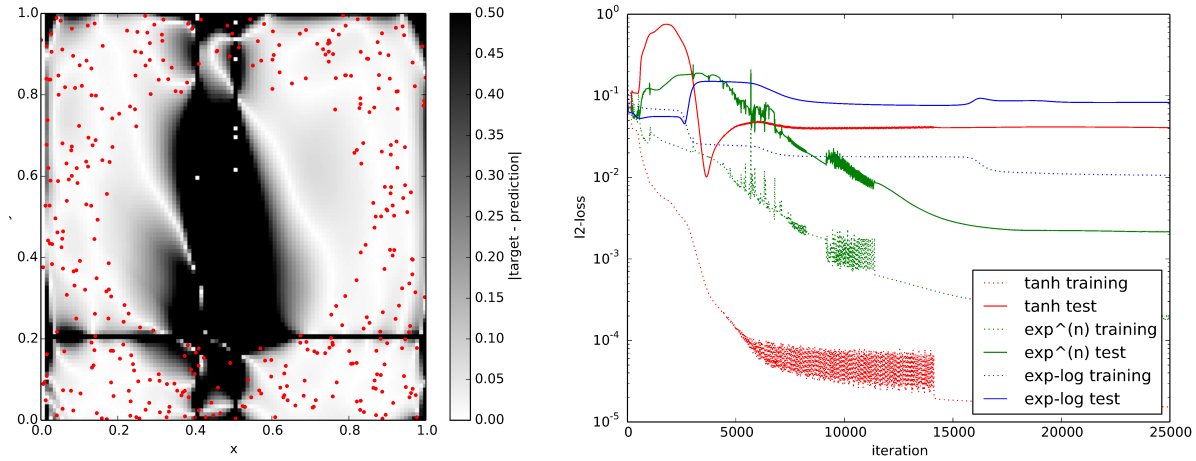


Figure 3.10: Histogram of the parameters $\hat{n}$ (top) and $\widetilde{n}$ (bottom) within the hidden layer of an ANN using the fractional exponential function after being trained on the MNIST dataset. Both were initialized to zero before training. A value of 0 corresponds to the identity function, 1 is the exponential and -1 is the logarithm.

We train three different ANNs with each having one hidden layer. The loss is optimized using Adam and the output layer is additive using the identity activation function in all cases. The first neural network is purely additive with hyperbolic tangent as the activation function for the input and hidden layers. The second network uses (3.36) in the input and hidden layer, thus allowing interpolation between addition and multiplication. The third and final network uses $\log$ and $\exp$ as activation functions in the input and hidden layer respectively, yielding fixed multiplicative interaction between the inputs and thus representing the structure of a polynomial. All weights of all networks, including the activation function parameters of our model, are initialized with zero mean and variance 10^{-2}. Hence, our model starts with a mostly additive configuration of neurons. The initial learning rate is set to 10^{-2} and all hyper-parameter combinations use a batch size of 100 training samples. The learning rate is reduced upon no improvement on the validation set until a minimum learning rate is reached; in that case training is terminated. The progression of training and test loss for all three structures is displayed in fig. 3.11b.

Our proposed activation function generalizes best in this experiment. The area where no training data is provided is approximated well. Surprisingly, our model even surpasses the $\log$-$\exp$ network, which perfectly resembles the data structure due to its fixed multiplicative



(a) training points and prediction error

(b) loss progression

Figure 3.11: Training of an "addiplication" neural network on a two-dimensional polynomial of fourth degree. (a) The training samples are shown as red dots in x-y-space. The color of the background shows the magnitude of the prediction error after training is finished using a $\exp^{(n)}$ network. (b) The curves show the mean-squared error versus the training iteration for a standard tanh neural network, a network with neurons that can interpolate between addition and multiplication, and a fixed addition-multiplication network with a structure suitable for the given polynomial. While the tanh networks overfits on this dataset, the proposed $\exp^{(n)}$ network generalizes well on the test set.

interactions. We hypothesize that training a neural network with multiplicative interactions but otherwise randomly initialized weights is hindered by very complex error landscapes that make it difficult for gradient-based optimizers to escape deep local minima. Our model seems unaffected by this issue, supposedly because it starts with additive interactions and can thus find reasonable values for the weights before moving into the multiplicative regime.

3.7 Discussion

We proposed a method to continuously and differentiably interpolate between addition and multiplication and showed how it can be integrated into neural networks by replacing the standard logistic activation function with a parameterized activation function. We presented the mathematical formulation of these concepts and showed how to integrate them into neural networks with a small increase in the number of parameters. It was further shown that efficient numerical implementation is possible with only a 35% increase in computational time compared to a purely additive neural network.

On a synthetic dataset of polynomials we have seen that multiplicative units resulted in greater accuracy and better long range generalization. However, on the standard digit classification benchmark MNIST a purely additive ANN slightly but consistently outperformed the proposed model. We suspect two reasons for this result. First, the transition from addition to multiplication is not monotonic, thus inducing local minima in the error surface, which will make a good minimum of the loss hard to find, especially when using first-order optimization methods as it is common with neural networks. While second order methods may help here, their use with large datasets is problematic due to memory consumption and the stochasticity of mini-batch training. Second, the expressive power introduced by multiplicative interactions may lead to overfitting because there is no regularizing force driving the neurons back into the additive regime. Especially the exponential function has a very strong growth and thus a sigmoid function can quickly be driven into saturation by its outputs. This encourages overfitting. While an ad-hoc regularization would be possible, we believe it is better to use a theoretically founded Bayesian approach to prevent overfitting.

For this purpose in chapter 5 we generalize the approach of a parameterized activation function to non-parametric stochastic activation functions with an appropriate prior that encourages smooth functions. These activation functions are a superset of the family of functions presented in this chapter and can thus also interpolate between addition and multiplication. However, they have two advantages. First, due to the prior, simple function will be preferred, making the model resilient to overfitting. Second, the flexibility of non-parametric functions allows for many possibilities (paths in function space) to transition from addition to multiplication. This results in a smooth error surface with far less local minima in which the network can get trapped, thus leading to a better discoverability of good network parameters.

Chapter 4

Elementwise Automatic Differentiation

Traditionally the derivatives necessary for training of ANNs, cf. section 2.5, have been computed using the backpropagation algorithm (Hecht-Nielsen et al., 1988; Y. A. LeCun et al., 2012; Y. LeCun, B. Boser, et al., 1989), which is a special case of reverse accumulation automatic differentiation (Griewank et al., 2008; Rall, 1981). While backpropagation is only applicable to standard ANNs, automatic differentiation can be used to calculate the derivatives of any model that provides a training objective in the form of a loss function, such as "addiplication" and Gaussian process neuron (GPN) networks do. When applied to a standard ANN, automatic differentiation performs the exact same steps as the backpropagation algorithm.

Currently automatic differentiation is implemented in many frameworks for deep learning, such as Theano (Al-Rfou et al., 2016) and TensorFlow (Martin Abadi et al., 2015). The implementation used in these software packages works at the tensor level, i.e. an example expression is

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) \triangleq W\boldsymbol{x} + \boldsymbol{y}$$

where W is a matrix of appropriate shape. When such an expression is evaluated on a GPU for each operation a separate compute kernel is invoked. A compute kernel is a massively parallel routine that is executed on the GPU. In this case two invocations would take place: one for the dot product and one for the addition.

These frameworks also support operations like reshaping (to change the shape of a tensor) and broadcasting (to repeat a tensor over a specified axis). This is useful to perform operations that do not correspond to well-known operations in linear algebra such as the dot product, elementwise additions and norms. For example, the matrix of pairwise differences

$$D_{ij}(\boldsymbol{x}, \boldsymbol{y}) \triangleq x_i - y_j$$

where $\boldsymbol{x} \in \mathbb{R}^N$, $\boldsymbol{y} \in \mathbb{R}^M$ could be written in tensor form as

$$D = \mathrm{broadcast}_2(\mathrm{reshape}_{N,1}(\boldsymbol{x})) - \mathrm{broadcast}_1(\mathrm{reshape}_{1,M}(\boldsymbol{y})) \tag{4.1}$$

where $\mathrm{broadcast}_d(\bullet)$ repeats its argument over dimension d so that the resulting shape fits subsequent operations and $\mathrm{reshape}_{N_1,N_2,\ldots,N_D}(\bullet)$ reshapes a tensor into shape $\mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D}$ without changing the total number of elements in the tensor. Working with expressions at the tensor level is a good fit for architectures like neural networks, which consist mostly of dot products and elementwise applications of (activation) functions.

However consider the following function which occurs in section 5.5. The function is specified elementwise, i.e. we are given an expression for element s, n of the matrix-valued function f which explicitly depends on elements of its argument matrices and tensors. The expression is

$$f_{s,n}(V, U, \kappa, \mu, \Sigma) = \sum_{r=1}^{R} \sum_{t=1}^{R} \left(\sum_{i=1}^{I} \kappa_{r,i,n} U_{i,n} \right) \left(\sum_{j=1}^{I} \kappa_{t,j,n} U_{j,n} \right) \cdot \tag{4.2}$$
$$\sqrt{\frac{1}{1 + 4\Sigma_{s,n,n}} \exp\left(- \frac{2\left[\mu_{s,n} - (V_{r,n} + V_{t,n})/2\right]^2}{1 + 4\Sigma_{s,n,n}} - \frac{(V_{r,n} - V_{t,n})^2}{2} \right)}.$$

It would be possible to rewrite it in tensor form, i.e. like (4.1), however this comes with four drawbacks. First, rewriting such complicated expressions into tensor form by hand is error-prone and hinders quick changes and experiments. Second, evaluating this expression in tensor form requires temporary memory of at least $R^2 \times S \times N$ elements to store the intermediate values of the square root before summation over r and t is performed. This may become prohibitive for large tensors. The elementwise form (4.2) does not require that memory as the elements of the occurring sums can be evaluated on the fly. Third, depending on the structure and the sizes of the involved tensors, evaluating the tensor form can be computationally inefficient, especially on GPUs. Since cache management on CUDA GPUs is manual and *local* to a compute kernel, intermediate values (for example all elements of the square root) must be stored and then retrieved from main graphics memory, because the computation of such an expression is split over multiple compute kernels. Fourth, if the involved tensors are small, then an expression like (4.2) will invoke many short-running compute kernels on GPUs. While the launch overhead of a compute kernel is generally small, it becomes significant in that case and can even become the performance limiting factor. Thus in this case it is desirable to evaluate (4.2) directly as given in elementwise form, instead of translating it into tensor form.

While describing a method for efficient evaluation of such expressions on a GPU is not in the scope of this thesis, we want to tackle a sub-problem that occurs when developing such an

implementation. As already stated, training is usually performed using gradient descent and thus derivatives of all expressions of a model are necessary. Frameworks that work on the tensor level handle this by composing the derivative of an expression also at the tensor level, i.e. for each tensor-level operation like dot product, broadcast or reshape, a corresponding tensor-level operation to calculate the derivative is specified. If we work at the element level, we have to follow a different strategy and derive derivative expressions that are also specified elementwise. This might seems straightforward at first, but this is not the case, since the way how indices appear on the arguments of such an elementwise defined function affects the derivative expression. Thus let us first review the issues that occur when handling such expressions on a few toy examples.

For our applications it is necessary to derive an expression for the gradient of a loss function, for example $l(a, b, c, \boldsymbol{d}) = l(f(a, b, c, \boldsymbol{d}))$, with respect to model parameters $a, b, c, \boldsymbol{d}$. This derivatives will be denoted by $(\mathrm{d}a)_{ik} \triangleq \partial l / \partial a_{ik}$. When the mapping between function indices and argument indices is not 1:1, special attention is required. For example, for the function $f_{ij}(x) = x_i^2$, the derivative of the loss w.r.t. x is $(\mathrm{d}x)_i \triangleq \partial l / \partial x_i = \sum_j (\mathrm{d}f)_{ij}\, 2\, x_i$; the sum is necessary because index j does not appear in the indices of f. Another example is $f_i(x) = x_{ii}^2$, where x is a matrix; here we have $(\mathrm{d}x)_{ij} = \delta_{ij}\, (\mathrm{d}f)_i\, 2\, x_{ii}$; the Kronecker delta is necessary because the derivative is zero for off-diagonal elements. Another indexing scheme is used by $f_{ij}(x) = \exp x_{i+j}$; here the correct derivative is $(\mathrm{d}x)_k = \sum_i (\mathrm{d}f)_{i,k-i}\, \exp x_k$, where the range of the sum must be chosen appropriately.

In this chapter we present an algorithm that can handle any case in which the indices of an argument are an *arbitrary linear combination* of the indices of the function, thus all of the above examples are governed. Sums (and their ranges) and Kronecker deltas are automatically inserted into the derivatives as necessary. Additionally, the indices are transformed, if required (as in the last example). The algorithm outputs a symbolic expression that defines the derivative elementwise and can thus be subsequently evaluated like form (4.2).

This work is also interesting outside the field of machine learning. For instance, recently Kjolstad et al. (2017) proposed a "tensor algebra compiler" that takes elementwise expressions of the form (4.2) and generates code to efficiently evaluate them. The algorithm presented in this chapter can thus be combined with such a tensor algebra compiler to automatically generate implementation code for an arbitrary elementwise expression and all its derivatives.

We first review the basic automatic differentiation algorithm (sections 4.1 and 4.2) and necessary algorithms for integer matrix inversion and for solving systems of linear inequalities (section 4.3). Then, in section 4.4, we show how to extend automatic differentiation to generate derivative expressions for elementwise defined tensor-valued functions. An example and numeric verification of our algorithm are presented in section 4.5.

Contributions to this Chapter

The method presented in this chapter was developed by me in the context of improving the computational performance of GPNs. A prototype was implemented together with Marcus Basalla and extended by me.

4.1 Symbolic Reverse Accumulation Automatic Differentiation

Every function f can be written as a composition of elementary functions such as addition, subtraction, multiplication, division, trigonometric functions, the exponential function, the logarithm and so on. For now let us assume that the elementary functions take one or more *scalar* arguments, thus f will also be a function accepting scalar arguments. For example, the function $f(x_1, x_2) = \exp(x_1 + x_2)$ can be written as $f(x_1, x_2) = f_1(f_2(x_1, x_2))$ with parts $f_1(t) = \exp(t)$ and $f_2(t_1, t_2) = t_1 + t_2$. It is also possible that parts appear more than once in a function. As an example $f(x) = \sin(x^2)\cdot\cos(x^2)$ can be decomposed into $f(x) = f_1\big[f_2\big(f_4(x)\big), f_3\big(f_4(x)\big)\big]$ where the parts $f_1(s, t) = s \cdot t$, $f_2(t) = \sin(t)$, $f_3(t) = \cos(t)$ are used once and $f_4(t) = t^2$ is used twice. A decomposition of a function into parts can be represented by a computational graph, that is a directed acyclic graph in which each node represents a function part f_i and an edge between two node represents that the target node is used as the value to an argument of the source node. An exemplary computational graph for the function $f(x) = f_1\big[f_2\big(f_3\big(f_4(x), f_5(x)\big)\big)\big]$ is shown by the blue nodes in fig. 4.1.

Automatic differentiation (Rall, 1981) is based on the well-known chain rule, which states that for a scalar function of the form $f(x) = g(h(x))$ the derivative can be written as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial g}{\partial h}\,\frac{\partial h}{\partial x}\,.$$

Given a function f and its decomposition into parts f_i, the following algorithm uses reverse accumulation automatic differentiation to obtain a computational graph for the derivatives of f. Since $f(\boldsymbol{x}) = f_1(\dots)$ the derivative of f w.r.t. f_1 is

$$\frac{\partial f}{\partial f_1} = 1\,. \tag{4.3}$$

Then iteratively do the following: Find a part f_i for which the derivative of all its consumers is available but $\partial f/\partial f_i$ is yet unknown. A part f_c is a consumer of part f_i, if f_i occurs as a direct argument to f_c in the function f. Thus, in the graphical representation of f part f_c is a consumer of f_i, if there exists an edge from f_c to f_i. Since the computational graph of a function is acyclic, there will always exist a part f_i for which this condition is fulfilled. Let $\mathrm{csmr}(f_i)$ be

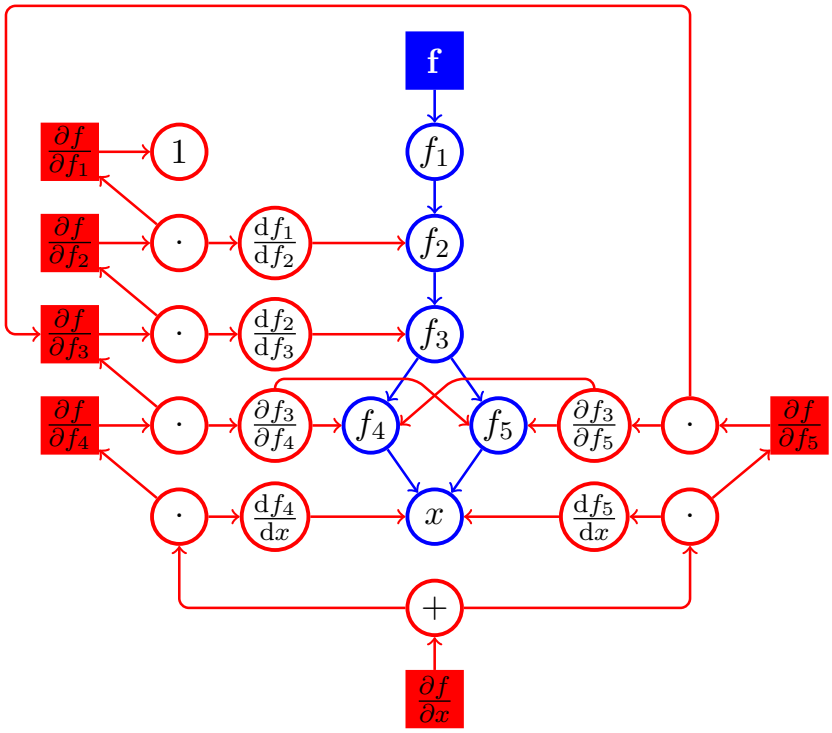


Figure 4.1: The blue nodes show a computational graph for the function $f(x) = f_1\big[f_2\big(f_3\big(f_4(x), f_5(x)\big)\big)\big]$. Each node f_1, f_2, f_3, f_4, f_5 represents a part of the function and each edge represents an argument. By applying reverse accumulation automatic differentiation as described in section 4.1 the computational graph for the derivatives (shown in red) is obtained.

the set of consumers of part f_i. Following the chain rule, the derivative of f w.r.t. f_i is given by

$$\frac{\partial f}{\partial f_i} = \sum_{d \in \mathrm{csmr}(f_i)} \frac{\partial f}{\partial f_d} \frac{\partial f_d}{\partial f_i} \,. \tag{4.4}$$

Repeat this process until the derivatives w.r.t. all parts $\partial f / \partial f_i$ have been calculated. Once completed, the derivatives of f w.r.t. its arguments x_j, $j \in \{1, \ldots, n\}$, follow immediately,

$$\frac{\partial f}{\partial x_j} = \sum_{d \in \mathrm{csmr}(x_j)} \frac{\partial f}{\partial f_d} \frac{\partial f_d}{\partial x_j} \,. \tag{4.5}$$

Note, that this algorithm requires a single pass only to complete the derivatives of f w.r.t. to *all* of its arguments.

By performing this algorithm on the computational graph shown in blue in fig. 4.1, the derivative represented by the red nodes and edges is obtained. The computation proceeds from top to bottom in a breadth-first order of traversation. In general the partial derivatives of the

function parts can depend on all of its arguments, as it can be seen in the dependencies of the nodes for $\partial f_3/\partial f_4$ and $\partial f_3/\partial f_5$. Symbolic derivatives can be obtained from the resulting computational graph by starting from the node $\partial f/\partial x_i$ and following the dependencies until reaching the leafs of the graph. However, for numerical evaluation it is more efficient to insert numerical values for the parameters $\boldsymbol{x}$ into the graph and then evaluate it node by node. This ensures that intermediate values are only computed once and thus the possibility of an exponential blow up of the number of terms that can occur during classical symbolic differentiation is avoided. To evaluate the derivative $\partial f/\partial x$ numerically, the function $f(x)$ must be evaluated followed by the derivatives of all parts. This corresponds to the forward and backward passes of the backpropagation algorithm for neural networks.

An example of a computational graph and its derivative for the concrete function

$$f(x_1, x_2, x_3) = \sin\big[\sin\big(x_1 \cdot (x_2 + x_3) + \sinh(x_2 + x_3)\big)\big]$$

is shown in fig. 4.2.

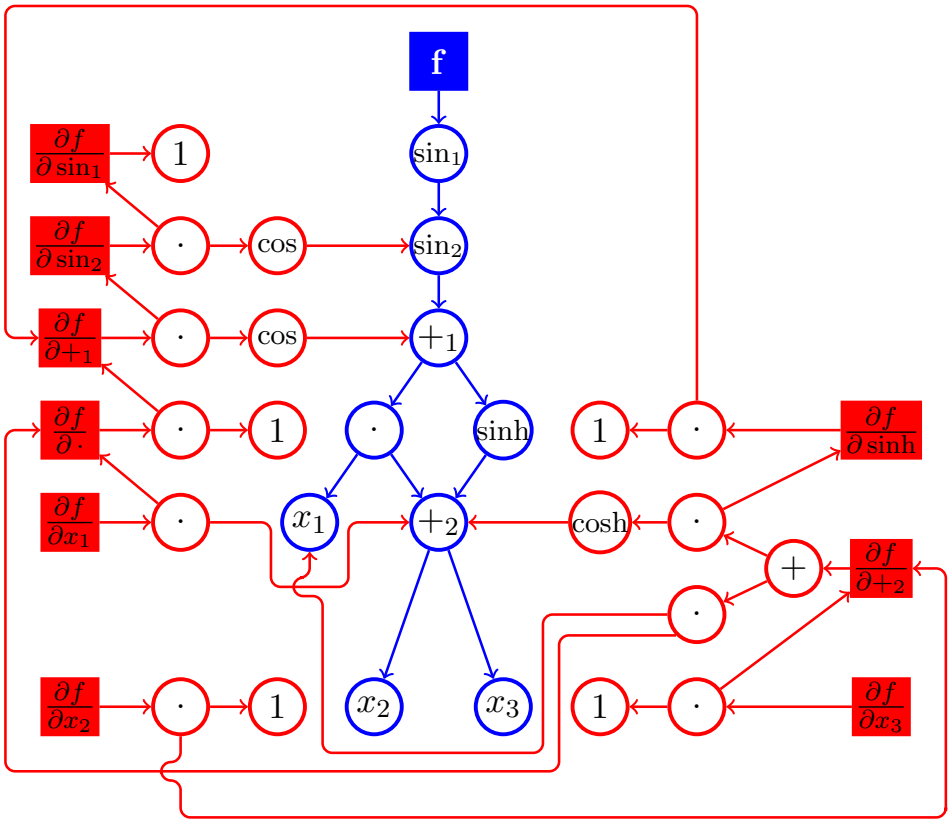


Figure 4.2: A practical example for reverse accumulation automatic differentiation. The computational graph for the function $f(x_1, x_2, x_3) = \sin\big[\sin\big(x_1 \cdot (x_2 + x_3) + \sinh(x_2 + x_3)\big)\big]$ is shown in blue. The computational graph for the derivatives obtained by automatic differentiation is shown in red. Note how intermediate values are reused automatically and the derivatives w.r.t. different x_i share most parts of the computational graph. Symbolic derivatives can be extracted from the graph or it can be evaluated numerically by substituting values for x_1, x_2 and x_3.

4.2 Handling Multidimensional Functions

So far we have shown automatic differentiation for scalar functions. However, in the context of ANNs and GPs we will mostly be dealing with functions that deal with tensor-valued functions. While any tensor-valued function can be written as a scalar function by splitting it into separate functions for each element of the tensor, it may be beneficial to directly work with tensor-valued functions if the employed primitive operations work on matrices or tensors. For example an ANN is defined as a series of dot products and elementwise applications of an activation function.

Also some operations benefit directly from calculating all elements of a matrix simultaneously. Matrix multiplication is such an operation. Calculating each element of $C = A \cdot B$ separately using $C_{ij} = \sum_k A_{ik} B_{kj}$ requires a total of $\mathcal{O}(n^3)$ operations where n is the size of the square matrices A and B. Contrary to that, calculating the elements simultaneously can be done in $\mathcal{O}(n^{2.807})$ using the Strassen algorithm (Strassen, 1969) or even more efficiently in $\mathcal{O}(n^{2.375})$ using the Coppersmith-Winograd algorithm (Coppersmith et al., 1987).[1] Thus we will show how automatic differentiation is performed on multidimensional functions. To our knowledge this is the working principle of the derivation functions in frameworks like Theano and TensorFlow.

For functions working in two- or higher dimensional space, we use the vectorization operator vec to transform them into vector-valued functions. For a D-dimensional tensor $A \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D}$ the vectorization operator is defined elementwise by

$$(\operatorname{vec} A)_{\sum_d s_d i_d} \triangleq A_{i_1, i_2, \ldots, i_D}\,, \qquad i_d \in \{1, \ldots, N_d\}\,, \tag{4.6}$$

where the strides $\boldsymbol{s}$ are given by

$$s_d \triangleq \prod_{b=2}^{d} N_{b-1}\,.$$

As an example, for a matrix $A \in \mathbb{R}^{N \times M}$ this operator takes the columns of the matrix and stacks them on top of one another,

$$\operatorname{vec} A = \left(A_{11}, A_{21}, \ldots, A_{N1}, A_{12}, A_{22}, \ldots, A_{N2}, \ldots, A_{1M}, A_{2M}, \ldots, A_{NM}\right)^T.$$

Thus the derivatives of a tensor-valued function $F : \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_D} \to \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_{D'}}$ can be dealt with by defining a helper function $\widehat{F} : \mathbb{R}^{N_1\, N_2 \cdots N_D} \to \mathbb{R}^{M_1\, M_2 \cdots M_{D'}}$ with $\widehat{F}(\operatorname{vec} X) =$

[1] While these algorithms are asymptotically faster than naive matrix multiplication, they also have a larger constant factor in their running time that is not captured by the big $\mathcal{O}$ notation. Therefore in practice they are only beneficial for matrices larger than a certain size.

$\operatorname{vec} F(X)$ and considering the derivatives of this vector-valued function $\widehat{F}$ instead.

It remains to show how to apply automatic differentiation to vector-valued functions. To do so, let us first see how the chain rule works on vector-valued functions. Consider two functions, $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^N$ and $\boldsymbol{h} : \mathbb{R}^M \to \mathbb{R}^K$, and a composite function $\boldsymbol{f} : \mathbb{R}^M \to \mathbb{R}^N$ with $\boldsymbol{f}(\boldsymbol{x}) \triangleq \boldsymbol{g}(\boldsymbol{h}(\boldsymbol{x}))$. By expanding $\boldsymbol{g}(\boldsymbol{r})$ as $\boldsymbol{g}(r_1, r_2, \ldots, r_K)$ and $\boldsymbol{h}(\boldsymbol{x})$ as $\big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_K(\boldsymbol{x})\big)^T$ we can write

$$f_i(\boldsymbol{x}) = g_i\big(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_K(\boldsymbol{x})\big)$$

and apply the chain rule on each argument of g_i, resulting in

$$\frac{\partial f_i}{\partial x_j} = \sum_{k=1}^{K} \frac{\partial g_i}{\partial h_k} \frac{\partial h_k}{\partial x_j} \,. \tag{4.7}$$

By introducing the Jacobian

$$\left(\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}}\right)_{ij} \triangleq \frac{\partial f_i}{\partial x_j}$$

we can rewrite (4.7) as a vectorized equation,

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}} \cdot \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} \,, \tag{4.8}$$

and thus obtain the chain rule for vector-valued functions. As we see, it is like the chain rule for scalars but with scalar multiplication replaced by matrix multiplication.

The algorithm for automatic differentiation for vector-valued functions is thus equal to scalar automatic differentiation described in section 4.1, but with eq. (4.3) replaced by

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_1}} = \mathbb{1} \tag{4.9}$$

and eq. (4.4) replaced by

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_i}} = \sum_{\boldsymbol{d} \in \operatorname{csmr}(\boldsymbol{f_i})} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{f_d}} \cdot \frac{\partial \boldsymbol{f_d}}{\partial \boldsymbol{f_i}} \,. \tag{4.10}$$

For many common operations the size of the Jacobian $\partial \boldsymbol{f_d} / \partial \boldsymbol{f_i}$ may become very large. For example, the Jacobian of a matrix multiplication is of size n^4 for two matrices of size $n \times n$. However, since most elements are indeed zero, it is possible and vastly more efficient to directly compute the product $(\partial \boldsymbol{f} / \partial \boldsymbol{f_d}) \cdot (\partial \boldsymbol{f_d} / \partial \boldsymbol{f_i})$ without explicitly evaluating the Jacobian. This is also the case for all elementary operations that work elementwise, such as addition, subtraction and the Hadamard product, which result in a diagonal Jacobian matrix. Consequently the explicit form (4.10) should only be used as a fall-back when such a shortcut computation is not

available.

4.3 Systems of Integer Equalities and Inequalities

This section reviews methods to solve systems of integer equalities and inequalities. The algorithms presented here are a prerequisite to compute the elementwise derivative expressions of tensor-valued functions, which will be described in the following section.

4.3.1 Systems of Linear Integer Equations

Consider a system of linear equations

$$\begin{aligned} A_{11}\,x_1 + A_{12}\,x_2 + \cdots + A_{1m}\,x_m &= b_1 \\ A_{21}\,x_1 + A_{22}\,x_2 + \cdots + A_{2m}\,x_m &= b_2 \\ \vdots \qquad\qquad & \vdots \qquad\quad \vdots \;\; \vdots \\ A_{n1}\,x_1 + A_{n2}\,x_2 + \cdots + A_{nm}\,x_m &= b_n\,, \end{aligned}$$

with *integer* coefficients $A \in \mathbb{Z}^{N\times M}$, *integer* variables $\boldsymbol{x} \in \mathbb{Z}^M$ and *integer* targets $\boldsymbol{b} \in \mathbb{Z}^N$. In matrix notation this system can be expressed much briefer as

$$A\,\boldsymbol{x} = \boldsymbol{b}\,. \tag{4.11}$$

To determine the set of solutions the matrix A must be transformed into Smith normal form, which is a diagonal matrix of the form

$$S = \operatorname{diag}(\alpha_1, \alpha_2, \ldots, \alpha_R, 0, \ldots, 0) \tag{4.12}$$

with the property that

$$\alpha_i \mid \alpha_{i+1}, \quad 1 \le i < r\,, \tag{4.13}$$

where $a \mid b$ should be read as "a *divides* b". Analogously $a \nmid b$ should be read as "a *does not divide* b". The number of non-zero entries R in the diagonal corresponds to the rank of A. It can be shown (Adkins et al., 1999) that for each non-zero matrix $A \in \mathbb{Z}^{N\times M}$ there exist invertible matrices $U \in \mathbb{Z}^{N\times N}$ and $V \in \mathbb{Z}^{M\times M}$ so that

$$S = U\,A\,V \tag{4.14}$$

where $S \in \mathbb{Z}^{N\times M}$ is the Smith normal form of A. Using the Smith normal form, the equation system (4.11) can be rewritten as

$$S\,\boldsymbol{x}' = \boldsymbol{b}' \tag{4.15}$$

with

$$\boldsymbol{x} = V\,\boldsymbol{x}'\,, \qquad \boldsymbol{b}' = U\,\boldsymbol{b}\,. \tag{4.16}$$

Since S is diagonal, the solutions can be read off from (4.15), as we describe in the following.

For rows of S that are zero the corresponding entries of $\boldsymbol{b}'$ must also be zero, otherwise the equation system would be inconsistent and no solution exists. Thus for the system to be solvable we must have

$$C\,\boldsymbol{b} = \boldsymbol{0} \tag{4.17}$$

where $C \in \mathbb{Z}^{(N-R)\times N}$ with $C_{ij} = U_{R+i,j}$ is the sub-matrix consisting of the rows $R+1$ to N of U. It is called the cokernel of A.

For each non-zero entry α_i of S we must have

$$x'_i = \frac{b'_i}{\alpha_i} \tag{4.18}$$

and thus a solution exists only if b'_i is dividable by α_i. We can define a so-called pseudo-inverse $I : \mathbb{Q}^{M\times N}$ by

$$I \triangleq V\,S^\dagger\,U \tag{4.19}$$

where $S^\dagger \in \mathbb{Q}^{N\times M}$ is defined by

$$S^\dagger = \mathrm{diag}(1/\alpha_1, 1/\alpha_2, \ldots, 1/\alpha_R, 0, \ldots, 0)\,, \tag{4.20}$$

with the factors α_i given by (4.12). This pseudo-inverse has the property that $A\,I\,A = A$. Thus, for every $\boldsymbol{b}$ that is in the cokernel of A, we can obtain an $\boldsymbol{x}$ so that $A\,\boldsymbol{x} = \boldsymbol{b}$ is fulfilled by setting $\boldsymbol{x} = I\,\boldsymbol{b}$.

For the columns of S that are zero the corresponding entries of $\boldsymbol{x}'$ do not affect the value of $\boldsymbol{b}'$. Consequently, the columns of the matrix $K \in \mathbb{Z}^{M\times(M-R)}$, with $K_{ij} = V_{i,R+j}$, are a basis for the kernel (also called null-space) of A. This means that $A\,K = 0$ and thus for every $\boldsymbol{b}$ that is in the cokernel of A we can write $\boldsymbol{b} = A(I\,\boldsymbol{b} + K\,\boldsymbol{z})$ where $\boldsymbol{z} \in \mathbb{Z}^{M-R}$ is a vector of arbitrary integers.

In summary, the equation system $A\,\boldsymbol{x} = \boldsymbol{b}$ has no integer solution for a particular $\boldsymbol{b}$, if $C\,\boldsymbol{b} \neq \boldsymbol{0}$ or $I\,\boldsymbol{b} \notin \mathbb{Z}^N$. Otherwise, if A has full rank, that is $R = N = M$, a unique integer solution exists, determined by $\boldsymbol{x} = I\,\boldsymbol{b}$. If A has non-full rank, infinitely many integer solutions

exist and are given by $\boldsymbol{x} = I\,\boldsymbol{b} + K\,\boldsymbol{z}$ where $\boldsymbol{z} \in \mathbb{Z}^{M-R}$ is a vector of arbitrary integers.

Computation of the Smith Normal Form

An algorithm (H. S. Smith, 1860) that, given a matrix A, computes the Smith normal form S and two matrices U and V, such that $S = U\,A\,V$ is shown in algorithm 8. The algorithm transforms the matrix A into Smith normal form by a series of elementary row and column operations. Matrices U and V are initialized to be identity matrices and the same row and column operations are applied to them, so that in the end the relation $S = U\,A\,V$ holds. Since all operations are elementary, it follows that U and V are invertible as required. By following the description of the algorithm it is clear that the resulting matrix S will be diagonal and fulfill the property (4.13). To find the factors β, σ and τ of Bézout's identity in steps 10 and 19 the extended Euclidean algorithm (Knuth, 1997) is used, which is presented in algorithm 9.

What remains to be shown is that the described algorithm terminates. With each iteration of the loop in step 9 the absolute value of the element S_{aa} decreases, because it is replaced with the greatest common divisor (GCD) of itself and another element. Thus, this loop will terminate since, in worst case, $S_{aa} = +1$ or $S_{aa} = -1$ will divide all following rows and columns. The same argument holds, when the matrix must be rediagonalized due to the execution of step 36. It is easy to verify that the first diagonalization step executed thereafter will set $S_{aa} = \gcd(S_{aa}, S_{a+1,a+1})$ and thus the absolute value of S_{aa} decreases. Thus, in the worst case, the loop terminates as soon as $S_{11} = S_{22} = \cdots = S_{R-1,R-1} = 1$, which then divides S_{RR}.

Algorithm 8: Smith normal form of an integer matrix

Input: non-zero matrix $A \in \mathbb{Z}^{N\times M}$

Output: Smith normal form $S \in \mathbb{Z}^{N\times M}$, invertible matrices $U \in \mathbb{Z}^{N\times N}$, $V \in \mathbb{Z}^{M\times M}$, rank R

1 $U \longleftarrow \mathbb{1}_N$; $V \longleftarrow \mathbb{1}_M$ // initialize U and V with identity matrices
2 $S \longleftarrow A$ // initialize S with A
3 $a \longleftarrow 1$ // initialize active row and column
4 **while** $\exists i, j : i \geq a \wedge j \geq a \wedge S_{ij} \neq 0$ **do** // diagonalize S
// Bring non-zero pivot element into position S_{aa}
5 Simultaneously $S_{\star a} \longleftarrow S_{\star j}$ and $S_{\star j} \longleftarrow S_{\star a}$
6 Simultaneously $V_{\star a} \longleftarrow V_{\star j}$ and $V_{\star j} \longleftarrow V_{\star a}$
7 Simultaneously $S_{a\star} \longleftarrow S_{i\star}$ and $S_{i\star} \longleftarrow S_{a\star}$
8 Simultaneously $U_{a\star} \longleftarrow U_{i\star}$ and $U_{i\star} \longleftarrow U_{a\star}$
9 **while** S *is changing* **do** // zero all elements below and right of S_{aa}
10 **while** $\exists i : i > a \wedge S_{aa} \nmid S_{ia}$ **do** // ensure divisibility of rows
11 Find β, σ, τ so that $\beta = \gcd(S_{aa}, S_{ia}) = \sigma\, S_{aa} + \tau\, S_{ia}$. // algorithm 9
12 $\gamma \longleftarrow \frac{S_{ia}}{\beta}$; $\alpha \longleftarrow \frac{S_{aa}}{\beta}$
13 Simultaneously $S_{a\star} \longleftarrow \sigma\, S_{a\star} + \tau S_{i\star}$ and $S_{i\star} \longleftarrow -\gamma\, S_{a\star} + \alpha S_{i\star}$
14 Simultaneously $U_{a\star} \longleftarrow \sigma\, U_{a\star} + \tau U_{i\star}$ and $U_{i\star} \longleftarrow -\gamma\, U_{a\star} + \alpha U_{i\star}$
15 **while** $\exists i : i > a \wedge S_{ia} \neq 0$ **do** // eliminate first element of rows
16 $f \longleftarrow \frac{S_{ia}}{S_{aa}}$
17 $S_{i\star} \longleftarrow S_{i\star} - f\, S_{a\star}$
18 $U_{i\star} \longleftarrow U_{i\star} - f\, U_{a\star}$
19 **while** $\exists j : j > a \wedge S_{aa} \nmid S_{aj}$ **do** // ensure divisibility of columns
20 Find β, σ, τ so that $\beta = \gcd(S_{aa}, S_{aj}) = \sigma\, S_{aa} + \tau\, S_{aj}$. // algorithm 9
21 $\gamma \longleftarrow \frac{S_{aj}}{\beta}$; $\alpha \longleftarrow \frac{S_{aa}}{\beta}$
22 Simultaneously $S_{\star a} \longleftarrow \sigma\, S_{\star a} + \tau S_{\star j}$ and $S_{\star j} \longleftarrow -\gamma\, S_{\star a} + \alpha S_{\star j}$
23 Simultaneously $V_{\star a} \longleftarrow \sigma\, V_{\star a} + \tau V_{\star j}$ and $V_{\star j} \longleftarrow -\gamma\, V_{\star a} + \alpha V_{\star j}$
24 **while** $\exists j : j > a \wedge S_{aj} \neq 0$ **do** // eliminate first element of columns
25 $f \longleftarrow \frac{S_{aj}}{S_{aa}}$
26 $S_{\star j} \longleftarrow S_{\star j} - f\, S_{\star a}$
27 $V_{\star j} \longleftarrow V_{\star j} - f\, V_{\star a}$
28 $a \longleftarrow a + 1$ // next diagonal element
29 $R \longleftarrow a - 1$ // rank is number of non-zero diagonal elements
30 **for** $a \in \{1, \ldots, R\}$ **do**
31 **if** $S_{aa} < 0$ **then** // ensure positive diagonal
32 $S_{\star a} \longleftarrow -S_{\star a}$
33 $V_{\star a} \longleftarrow -V_{\star a}$
34 **if** $a \leq R - 1 \wedge S_{aa} \nmid S_{a+1,a+1}$ **then** // ensure divisibility constraints
35 $S_{\star a} \longleftarrow S_{\star a} + S_{\star,a+1}$
36 $V_{\star a} \longleftarrow V_{\star a} + V_{\star,a+1}$
37 Go back to step 4.

Algorithm 9: Extended Euclidean algorithm

Input: positive numbers $a \in \mathbb{Z}^+$, $b \in \mathbb{Z}^+$
Output: factors $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $z \in \mathbb{Z}$ fulfilling Bézout's identity $z = \gcd(a, b) = a\,x + b\,y$

$r_0 \longleftarrow a$; $r_1 \longleftarrow b$
$s_0 \longleftarrow 1$; $s_1 \longleftarrow 0$
$t_0 \longleftarrow 0$; $t_1 \longleftarrow 1$
$i \longleftarrow 1$
while $r_i \neq 0$ **do**
    $q \longleftarrow \frac{r_{i-1}}{r_i}$ `// integer division`
    $r_{i+1} \longleftarrow r_{i-1} - q\,r_i$
    $s_{i+1} \longleftarrow s_{i-1} - q\,s_i$
    $t_{i+1} \longleftarrow t_{i-1} - q\,t_i$
    $i \longleftarrow i + 1$
$z \longleftarrow r_{i-1}$; $x \longleftarrow s_{i-1}$; $y \longleftarrow t_{i-1}$

4.3.2 Systems of Linear Inequalities

Consider a system of linear inequalities

$$\begin{aligned} A_{11}\,x_1 + A_{12}\,x_2 + \cdots + A_{1M}\,x_M &\geq b_1 \\ A_{21}\,x_1 + A_{22}\,x_2 + \cdots + A_{2M}\,x_M &\geq b_2 \\ \vdots \qquad\qquad \vdots \qquad\qquad &\vdots\ \vdots \\ A_{N1}\,x_1 + A_{N2}\,x_2 + \cdots + A_{NM}\,x_M &\geq b_N\,, \end{aligned} \tag{4.21}$$

with coefficients $A \in \mathbb{R}^{N\times M}$, variables $\boldsymbol{x} \in \mathbb{R}^M$ and biases $\boldsymbol{b} \in \mathbb{R}^N$. Note that this notation can also describe equalities by including the same line twice, where one occurrence is multiplied by -1 on both sides. In matrix notation this inequality system can be expressed much briefer as

$$A\,\boldsymbol{x} \geq \boldsymbol{b}\,. \tag{4.22}$$

The objective is to transform the inequality system into the form

$$\max(L^M\boldsymbol{b}) \leq x_M \quad \leq \min(H^M\boldsymbol{b}) \tag{4.23a}$$
$$\max(L^{M-1}\boldsymbol{b} + \widehat{L}^{M-1}\boldsymbol{x}_M) \leq x_{M-1} \leq \min(H^{M-1}\boldsymbol{b} + \widehat{H}^{M-1}\boldsymbol{x}_M) \tag{4.23b}$$
$$\max(L^{M-2}\boldsymbol{b} + \widehat{L}^{M-2}\boldsymbol{x}_{M-1\,\ldots\,M}) \leq x_{M-2} \leq \min(H^{M-2}\boldsymbol{b} + \widehat{H}^{M-2}\boldsymbol{x}_{M-1\,\ldots\,M}) \tag{4.23c}$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots$$
$$\max(L^2\boldsymbol{b} + \widehat{L}^2\boldsymbol{x}_{3\,\ldots\,M}) \leq x_2 \quad \leq \min(H^2\boldsymbol{b} + \widehat{H}^2\boldsymbol{x}_{3\,\ldots\,M}) \tag{4.23d}$$
$$\max(L^1\boldsymbol{b} + \widehat{L}^1\boldsymbol{x}_{2\,\ldots\,M}) \leq x_1 \quad \leq \min(H^1\boldsymbol{b} + \widehat{H}^1\boldsymbol{x}_{2\,\ldots\,M})\,, \tag{4.23e}$$

so that the range of each element x_i can be determined sequentially. Here $\boldsymbol{x}_{i\,\ldots\,j}$ should be read as the subvector of $\boldsymbol{x}$ starting at element i and including all elements up to (including) element j. Furthermore $\min \boldsymbol{z}$ and $\max \boldsymbol{z}$ mean the minimum or maximum element of a vector $\boldsymbol{z}$. The transformed system should be tight in the sense that given a subvector $\boldsymbol{x}_{M-s\,\ldots\,M}$ which satisfies the first $s+1$ inequalities there must exist remaining elements $\boldsymbol{x}_{1\,\ldots\,M-s-1}$ so that $\boldsymbol{x}$ satisfies all inequalities. This is equivalent to demanding that the transformed inequalities must not allow values for an element x_i so that the ranges of allowed values for other elements of $\boldsymbol{x}$ become empty. Obviously the matrices L^i, $\widehat{L}^i$, H^i and $\widehat{H}^i$ depend on A and must be determined.

Multiplying an inequality by a positive, non-zero factor will result in an equivalent system, where equivalent means that it has exactly the same set of solutions as the original system. Thus, by dividing each line i with $A_{i1} \neq 0$ by the factor $|A_{i1}|$ and rearranging, we can bring a

system of the form (4.21) into the equivalent form

$$x_1 + \sum_{j=2}^{M} D_{hj} x_j \geq d_h\,, \quad h \in \{1, \ldots, H\}\,, \tag{4.24a}$$

$$-x_1 + \sum_{j=2}^{M} E_{kj} x_j \geq e_k\,, \quad k \in \{1, \ldots, K\}\,, \tag{4.24b}$$

$$\sum_{j=2}^{M} F_{lj} x_j \geq f_l\,, \quad l \in \{1, \ldots, L\}\,, \tag{4.24c}$$

with $H + K + L = N$. It is clear that adding two inequalities will not reduce the set of solutions, i.e. if $\boldsymbol{x}$ is a solution to the inequalities $\boldsymbol{a}^T\boldsymbol{x} \geq \alpha$ and $\boldsymbol{b}^T\boldsymbol{x} \geq \beta$, then $\boldsymbol{x}$ is also a solution to the inequality $(\boldsymbol{a} + \boldsymbol{b})^T\boldsymbol{x} \geq \alpha + \beta$. Consequently by adding each inequality from (4.24a) to each inequality from (4.24b) and dropping the used inequalities we arrive at the reduced system with x_1 eliminated,

$$\sum_{j=2}^{M} (D_{hj} + E_{kj})\, x_j \geq d_h + e_k\,, \quad h \in \{1, \ldots, H\},\, k \in \{1, \ldots, K\}\,, \tag{4.25a}$$

$$\sum_{j=2}^{M} F_{lj} x_j \geq f_l\,, \qquad l \in \{1, \ldots, L\}\,, \tag{4.25b}$$

which has at least the solutions $\boldsymbol{x}$ of the original system consisting of (4.24). Fourier and Motzkin (G. B. Dantzig and Eaves, 1973) observed that both systems are indeed equivalent. To verify this, we have to show that for each solution $\boldsymbol{x}_{2\cdots M}$ of (4.25), there exists x_1 so that the combined $\boldsymbol{x}$ satisfies (4.24). From (4.24a) and (4.24b) we see that an x_1 satisfying the original system is given by

$$\min_k \left(\sum_{j=2}^{M} E_{kj} x_j - e_k \right) \geq x_1 \geq \max_h \left(-\sum_{j=2}^{M} D_{hj} x_j + d_h \right) \tag{4.26}$$

and rewriting (4.25a) as

$$\sum_{j=2}^{M} E_{kj} x_j - e_k \geq -\sum_{j=2}^{M} D_{hj} x_j + d_h\,, \quad h \in \{1, \ldots, H\},\, k \in \{1, \ldots, K\} \tag{4.27}$$

shows that an x_1 with this property exists if the reduced system is satisfied.

By iteratively applying the reduction method just described, we can sequentially eliminate x_1, x_2 and so on up to x_M, as long as there exists at least one pair of inequalities with opposite

signs for a specific x_i. If this is not the case, then the remaining $x_{i+1\,\ldots\,M}$ are not affected by these inequalities since a value for x_i can always be found after determining $x_{i+1\,\ldots\,M}$ because x_i is bounded from one side only; consequently when x_i occurs with positive or negative sign only, all inequalities containing x_i can be dropped to progress with the elimination. After x_M has been eliminated, what remains is a system of constant inequalities of the form

$$0 \geq f_l\,, \quad l \in \{1, \ldots, L\}\,. \tag{4.28}$$

If these inequalities contain a contradiction, i.e. if any f_l is positive, the original system of inequalities is inconsistent and the set of solutions for $\boldsymbol{x}$ is empty.

This elimination method gives rise to algorithm 10 which has been adapted from (G. Dantzig, 2016; G. B. Dantzig and Thapa, 2006a,b) to work on matrix A only and thus solving the system of inequalities for arbitrary $\boldsymbol{b}$. The algorithm produces matrices L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ for $i \in \{1, \ldots, M\}$ that can be inserted into the inequalities (4.23) to subsequently obtain the ranges for each element of $\boldsymbol{x}$. It also outputs the feasibility matrix F, with the property that if $F\boldsymbol{b} \leq \boldsymbol{0}$, then there exist a solution for a particular $\boldsymbol{b}$.

Algorithm 10: Fourier-Motzkin elimination for a system of linear inequalities $A\,\boldsymbol{x} \geq \boldsymbol{b}$

Input: matrix $A \in \mathbb{R}^{N \times M}$
Output: matrices L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ for $i \in \{1, \ldots, M\}$ for use in (4.23); feasibility matrix F

$B \longleftarrow \mathbb{1}_N$ // initialize B with identity matrix
for $k \in \{1, \ldots, M\}$ **do** // loop over variables to eliminate
  // divide each row i by $|A_{ik}|$
  for $i \in \{1, \ldots N\}$ **do**
    if $A_{ik} \neq 0$ **then**
      $A_{i\star} \longleftarrow \frac{1}{|A_{ik}|} A_{i\star}$
      $B_{i\star} \longleftarrow \frac{1}{|A_{ik}|} B_{i\star}$
  // extract solution matrices
  $\zeta \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = 0\}$; $\phi \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = +1\}$; $\mu \longleftarrow \{i \in \mathbb{Z} \mid A_{ik} = -1\}$
  $S \longleftarrow -$ columns $\{k+1, \ldots, M\}$ of A
  $L^k \longleftarrow$ rows ϕ of B; $H^k \longleftarrow -$ rows μ of B
  $\widehat{L}^k \longleftarrow$ rows ϕ of S; $\widehat{H}^k \longleftarrow -$ rows μ of S
  // eliminate x_k
  if $\phi = \emptyset \wedge \mu = \emptyset$ **then**
    // x_k does not occur, nothing to eliminate
  else if $\phi = \emptyset \vee \mu = \emptyset$ **then**
    // x_k occurs with coefficient $+1$ or -1 only
    $A \longleftarrow$ rows ζ of A; $B \longleftarrow$ rows ζ of B
  else
    // x_k occurs with coefficients $+1$ and -1
    $A' \longleftarrow$ rows ζ of A; $B' \longleftarrow$ rows ζ of B
    for $p \in \phi$ **do**
      for $n \in \mu$ **do**
        $A' \longleftarrow A'$ with row $A_{p\star} + A_{n\star}$ appended
        $B' \longleftarrow B'$ with row $B_{p\star} + B_{n\star}$ appended
    $A \longleftarrow A'$; $B \longleftarrow B'$
$F \longleftarrow B$ // inequalities with no variables left

4.4 Elementwise-Defined Functions and their Derivatives

We introduce the situations that can occur when calculating the derivatives of elementwise-defined tensors using the following set of examples. Then we will describe a general method to derive expressions for the derivatives of elementwise-defined tensors, where the indices of the arguments are an arbitrary linear combination of the indices of the function output. Summations within these expressions are allowed.

Consider the vector-valued function $\boldsymbol{f}^1 : \mathbb{R}^N \to \mathbb{R}^N$, that is defined by specifying how each element of $\boldsymbol{f}^1(\boldsymbol{x})$ depends on the elements of its arguments $\boldsymbol{x}$. For example, a very simple example for such a function is

$$f_i^1(\boldsymbol{x}) = \sin x_i \,.$$

Here it is straightforward to see that its Jacobian is given by

$$\frac{\partial f_i^1}{\partial x_{i'}} = \delta_{i,i'} \, \cos x_{i'}$$

since element i of $\boldsymbol{f}^1$ only depends by element i of its arguments $\boldsymbol{x}$. Hence, the Kronecker delta was introduced in the above expression to make sure that $\partial f_i^1 / \partial x_{i'} = 0$ for $i \neq i'$.

Further assume that $\boldsymbol{f}$ is part of a scalar function l with $l(\boldsymbol{x}) = g(\boldsymbol{f}(\boldsymbol{x}))$ and the derivatives of l w.r.t. the elements of $\boldsymbol{x}$ are to be derived. The derivatives $\partial g / \partial f_i$ are supposed to be known. Let us introduce the notation

$$\mathrm{d}\bullet_\alpha = \frac{\partial l}{\partial \bullet_\alpha}$$

for the derivatives of l w.r.t. an element of a variable or function. In the context of deep learning this is the derivative we are usually interested in, since it provides the gradient of a loss function l and is thus used for minimization of the loss. The explicit computation of the Jacobians $\partial f_i / \partial x_j$ is usually not of interest since it wastes space. We obtain for our function $\boldsymbol{f}^1(\boldsymbol{x})$,

$$\mathrm{d}f_{i'}^1 = \sum_i \frac{\partial g}{\partial f_i^1} \frac{\partial f_i^1}{\partial x_{i'}} = \sum_i \mathrm{d}g_i \, \delta_{i,i'} \, \cos x_i = \mathrm{d}g_{i'} \, \cos x_{i'} \,.$$

Let us now consider a slightly more complicated example given by the function $f^2 : \mathbb{R}^N \times \mathbb{R}^{N \times N} \to \mathbb{R}^{N \times N}$ of two arguments with the elementwise specification

$$f_{ij}^2(\boldsymbol{x}, y) = x_i \, y_{ij} \,.$$

The (extended) Jacobians w.r.t. x and y are given by

$$\frac{\partial f^2_{ij}}{\partial x_{i'}} = \delta_{i,i'}\, y_{i'j}\,, \qquad \frac{\partial f^2_{ij}}{\partial y_{i'j'}} = \delta_{i,i'}\, \delta_{j,j'}\, x_{i'}\,,$$

where the derivative w.r.t. x does not contain a Kronecker delta for index j, since it is not used to index variable x. Consequently application of the chain rule gives the following derivatives of l,

$$\mathrm{d}x_{i'} = \sum_j \mathrm{d}g_{i'j}\, y_{i'j}\,, \qquad \mathrm{d}y_{i'j'} = \mathrm{d}g_{i'j'}\, x_{i'}\,,$$

where the lack of index j on variable x has lead to a summation over this index.

Another situation is demonstrated by the function $f^3 : \mathbb{R}^{N\times N} \to \mathbb{R}^N$ with

$$f^3_i(x) = x^3_{ii}\,.$$

The Jacobian,

$$\frac{\partial f^3_i}{\partial x_{i'j'}} = \delta_{i,i'}\, \delta_{i,j'}\, 3x^2_{i'j'}\,,$$

now contains two Kronecker deltas for the index i to express that $i = i' = j'$ must hold so that the derivative is non-zero. This leads to the derivative of l,

$$\mathrm{d}x_{i'j'} = \delta_{i',j'}\, \mathrm{d}g_{i'}\, 3x^2_{i'j'}\,,$$

which now contains a Kronecker delta itself, since it has not been canceled out by a corresponding summation.

A good example for a function containing a summation over its arguments is the matrix dot product,

$$f^4_{ij}(x, y) = \sum_k x_{ik}\, y_{kj}\,,$$

which has the (extended) Jacobians

$$\frac{\partial f^4_{ij}}{\partial x_{i'k'}} = \delta_{i,i'} \sum_k \delta_{k,k'}\, y_{kj}\,, \qquad \frac{\partial f^4_{ij}}{\partial y_{k'j'}} = \delta_{j,j'} \sum_k \delta_{k,k'}\, x_{ik}\,.$$

Thus the derivatives of l evaluate to

$$\mathrm{d}x_{i'k'} = \sum_i \sum_j \mathrm{d}g_{ij}\, \delta_{i,i'} \sum_k \delta_{k,k'}\, y_{kj} = \sum_j \mathrm{d}g_{i'j}\, y_{k'j}\,,$$
$$\mathrm{d}y_{k'j'} = \sum_i \sum_j \mathrm{d}g_{ij}\, \delta_{j,j'} \sum_k \delta_{k,k'}\, x_{ik} = \sum_i \mathrm{d}g_{ij'}\, x_{ik'}\,.$$

Note that the summation indices of the derivatives have changed over from k to j and i respectively.

Finally consider the situation where the indices of the argument are given by a linear combination of the function indices, as demonstrated by $f^5 : \mathbb{R}^{N\times M} \to \mathbb{R}^{NM}$ with

$$f^5_{ij}(\boldsymbol{x}) = \exp x_{Mi+j}\,.$$

These indexing scheme could, for example, be used to reshape a vector organized in row-major form into a matrix. Its Jacobian is straightforward to express,

$$\frac{\partial f^5_{ij}}{\partial x_{i'}} = \delta_{Mi+j,i'}\, \exp x_{Mi+j}\,,$$

however to efficiently express the derivative of l,

$$\mathrm{d}x_{i'} = \sum_i \sum_j \mathrm{d}g_{ij}\, \delta_{Mi+j,i'}\, \exp x_{Mi+j}\,,$$

the occurring Kronecker delta should be combined with one of the sums, because one of them is redundant. To do so it is necessary to solve the equation $Mi + j = i'$ for j, which is trivial in this example. The solution is given by $j = i' - Mi$ and after substitution this results in

$$\mathrm{d}x_{i'} = \sum_i \mathrm{d}g_{i,i'-Mi}\, \exp x_{i'}\,.$$

We have seen that, depending on the constellation of indices of the arguments of a elementwise-defined function, the derivative will either introduce additional summations, drop existing summations, introduce Kronecker deltas or even require substitution of the solution of a linear equation system into the indices or a combination of these things.

4.4.1 Computing Elementwise Derivative Expressions

We first describe the method without accounting for summations inside the function and reintroduce them later. Generally the problem of computing expressions for elementwise derivatives

can be stated as follows. Let $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_{D_f})$ be a multi-index and let the tensor-valued function $f : \mathbb{R}^{N_1^1 \times \cdots \times N_{D_1}^1} \times \cdots \times \mathbb{R}^{N_1^P \times \cdots \times N_{D_P}^P} \to \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f}$ taking P tensor arguments called $x^1, x^2, \ldots, x^P$ be specified elementwise,

$$f_{\boldsymbol{\alpha}}(x^1, x^2, \ldots, x^P) \triangleq \overline{f}(x^1_{A^1\boldsymbol{\alpha}}, x^2_{A^2\boldsymbol{\alpha}}, \ldots, x^P_{A^P\boldsymbol{\alpha}}), \tag{4.29}$$

where each matrix $A^p : \mathbb{Z}^{D_f} \to \mathbb{Z}^{D_p}$ maps from the indices of f to the indices of its argument x^p. Such a linear transform covers all the cases shown in the introductory examples. If the same argument x^p should appear multiple times with different indices, we shall treat it as different arguments (by renaming the different occurrences) and sum over the resulting expressions for the derivatives after they have been obtained. Note that $\overline{f} : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}$ is a *scalar* function. Furthermore let $g : \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f} \to \mathbb{R}$ be a scalar-valued function and let $l = g \circ f$. Let $\mathrm{d}f \in \mathbb{R}^{N_1^f \times \cdots \times N_{D_f}^f}$ be the tensor of derivatives of l w.r.t. the elements of f, thus by above definition

$$\mathrm{d}f_{\boldsymbol{\alpha}} = \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} = \frac{\partial g}{\partial f_{\boldsymbol{\alpha}}}. \tag{4.30}$$

The objective is to obtain expressions that specify the derivatives of l w.r.t. the elements of each x^p elementwise, i.e.

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \frac{\partial l}{\partial x^p_{\boldsymbol{\beta}^p}} \tag{4.31}$$

where $\boldsymbol{\beta}^p = (\beta^p_1, \beta^p_2, \ldots, \beta^p_{D_p})$ is a multi-index enumerating the elements of x^p.

Applying the chain rule to (4.31) gives

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \frac{\partial l}{\partial x^p_{\boldsymbol{\beta}^p}} = \sum_{\substack{\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \\ A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p}} \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial x^p_{\boldsymbol{\beta}^p}} = \sum_{\boldsymbol{\alpha} \in \Gamma(\boldsymbol{\beta}^p)} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}} \tag{4.32}$$

and since $\overline{f}$ is a scalar function, computing the scalar derivative $\partial\overline{f}/\partial x^p_{A^p\boldsymbol{\alpha}}$ is straightforward using the strategy described in section 4.1. Thus the main challenge is to efficiently evaluate the summation over the set

$$\Gamma(\boldsymbol{\beta}^p) \triangleq \{\boldsymbol{\alpha} \in \mathbb{Z}^{D_f} \mid \mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f \wedge A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p\}, \tag{4.33}$$

i.e. find all *integer* vectors $\boldsymbol{\alpha}$ that fulfill the relation $A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p$ and lie within the range $\mathbf{1} \le \boldsymbol{\alpha} \le \boldsymbol{N}^f$ determined by the shape of f.

An elementary approach is to rewrite eq. (4.32) as

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \sum_{\alpha_1=1}^{N^1} \cdots \sum_{\alpha_{D_f}=1}^{N^{D_f}} \delta_{A^p\boldsymbol{\alpha}-\boldsymbol{\beta}^p}\, \mathrm{d}f_{\boldsymbol{\alpha}}\, \frac{\partial \overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}} \tag{4.34}$$

where the single-argument Kronecker delta is given by $\delta_{\boldsymbol{t}} = 0$ for $\boldsymbol{t} \neq \mathbf{0}$ and $\delta_{\mathbf{0}} = 1$. Thus for each index $\boldsymbol{\alpha}$ of f we test explicitly if it contributes to the derivative of index $\boldsymbol{\beta}^p$ of argument x^p and if so, we include that element in the summation. By evaluating (4.34) for all $\boldsymbol{\beta}^p$ in parallel the cost of iterating over $\boldsymbol{\alpha}$ can be amortized over all elements of $\mathrm{d}x^p$. However, if multiple threads are used to perform this iteration, as it is required to gain acceptable performance on modern GPUs, locking becomes necessary to serialize writes to the same element of $\mathrm{d}x^p$. If A^p has low rank, write collisions on $\mathrm{d}x^p_{A^p\boldsymbol{\alpha}}$ become likely, leading to serialization and thus considerable performance loss.[2] Another drawback of this approach is that even if only a subset of elements of the derivative $\mathrm{d}x^p$ are required, the summation must always be performed over the whole range of $\boldsymbol{\alpha}$. Furthermore, while not being of great importance for minimization of loss functions in machine learning, it is regrettable that no symbolic expression for $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ is obtained using this method.

For these reasons it is advantageous to find a form of the set (4.33) that directly enumerates all $\boldsymbol{\alpha}$ belonging to a particular $\boldsymbol{\beta}^p$. This requires solving $A^p\boldsymbol{\alpha} = \boldsymbol{\beta}^p$ for $\boldsymbol{\alpha}$. In general, a set of linear equations with integer coefficients over integer variables, has either none, one or infinitely many solutions. The set of solutions can be fully described using the pseudo-inverse, cokernel and kernel. Thus, let I be the pseudo-inverse, C the cokernel and K the kernel of the integer matrix A as defined in section 4.3.1. Using these matrices we can rewrite (4.33) as

$$\Gamma(\boldsymbol{\beta}^p) = \{I\boldsymbol{\beta}^p + K\boldsymbol{z} \mid C\boldsymbol{\beta}^p = \mathbf{0} \wedge I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f} \wedge \boldsymbol{z} \in \mathbb{Z}^{\kappa} \wedge \mathbf{1} \leq I\boldsymbol{\beta}^p + K\boldsymbol{z} \leq \boldsymbol{N}^f\}\,, \tag{4.35}$$

where κ is the dimensionality of the kernel of A. The conditions $C\boldsymbol{\beta}^p = \mathbf{0}$ and $I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f}$ determine whether the set is empty or not for a particular $\boldsymbol{\beta}^p$ and since they are independent of $\boldsymbol{z}$, they only need to be checked once for each $\boldsymbol{\beta}^p$. Thus if these conditions do not hold, we can immediately conclude that $\mathrm{d}x^p_{\boldsymbol{\beta}^p} = 0$. Otherwise, in order to further simplify the set specification, we need to find the elements of the set

$$\Sigma(\boldsymbol{\beta}^p) \triangleq \{\boldsymbol{z} \in \mathbb{Z}^{\kappa} \mid \mathbf{1} \leq I\boldsymbol{\beta}^p + K\boldsymbol{z} \leq \boldsymbol{N}^f\} \tag{4.36}$$

[2] The CUDA programming guide (Nvidia, 2017) is vague about the performance penalties associated with atomic addition to the same memory address from within multiple threads. Nonetheless, experiments (Farzad, 2013; yidiyidawu, 2012) show that performance can be degraded by up to a factor of 32 due to locking and resulting serialization.

containing all $\boldsymbol{z}$ that generate values for $\boldsymbol{\alpha}$ within its valid range. Since $\boldsymbol{\alpha}(\boldsymbol{z}) = I\boldsymbol{\beta}^p + K\boldsymbol{z}$ is an affine transformation, the set $\Sigma(\boldsymbol{\beta}^p)$ must be convex. By rewriting the system of inequalities defining the set $\Sigma(\boldsymbol{\beta}^p)$ as

$$K\,\boldsymbol{z} \geq \mathbf{1} - I\boldsymbol{\beta}^p \tag{4.37a}$$

$$-K\,\boldsymbol{z} \geq -\boldsymbol{N}^f + I\boldsymbol{\beta}^p \tag{4.37b}$$

we can apply the Fourier-Motzkin algorithm described in section 4.3.2 to obtain the boundaries of the convex set in closed form. The Fourier-Motzkin algorithm produces matrix L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ so that (4.36) can be written as

$$\begin{aligned}\Sigma(\boldsymbol{\beta}^p) = \{\boldsymbol{z} \in \mathbb{Z}^\kappa \,|\, &\lceil \max(L^\kappa \boldsymbol{b})\rceil \leq z_\kappa \leq \lfloor \min(H^\kappa \boldsymbol{b})\rfloor \wedge \\ &\lceil \max(L^{\kappa-1}\boldsymbol{b} + \widehat{L}^{\kappa-1}\boldsymbol{z}_\kappa)\rceil \leq z_{\kappa-1} \leq \lfloor \min(H^{\kappa-1}\boldsymbol{b} + \widehat{H}^{\kappa-1}\boldsymbol{z}_\kappa)\rfloor \wedge \\ &\cdots \wedge \\ &\lceil \max(L^1\boldsymbol{b} + \widehat{L}^1\boldsymbol{z}_{2\,\ldots\,\kappa})\rceil \leq z_1 \leq \lfloor \min(H^1\boldsymbol{b} + \widehat{H}^1\boldsymbol{z}_{2\,\ldots\,\kappa})\rfloor\}\end{aligned} \tag{4.38}$$

where

$$\boldsymbol{b}(\boldsymbol{\beta}^p) \triangleq \begin{bmatrix} \mathbf{1} - I\boldsymbol{\beta}^p \\ -\boldsymbol{N}^f + I\boldsymbol{\beta}^p \end{bmatrix}$$

and $\lfloor\bullet\rfloor$ and $\lceil\bullet\rceil$ are the floor and ceiling respectively. Since the Fourier-Motzkin algorithm executes independently of the value of $\boldsymbol{b}$, the computationally intensive procedure of computing the matrices L^i, H^i and $\widehat{L}^i$, $\widehat{H}^i$ is only done once for each kernel matrix K. Afterwards, computing the boundaries for a particular index $\boldsymbol{\beta}^p$ requires only four matrix multiplications per dimension and the determination of the minimum and maximum value of a vector.

An example for a one-dimensional kernel, i.e. line, is shown in fig. 4.3. In this case (4.38) consists of only one condition for z_1 and describes the part of the line that is inside the range specified by $\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f$. Another example, this time for a two-dimensional kernel, i.e. plane, is shown in fig. 4.4. Due to the choice of the kernel basis, the range specified by $\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f$ becomes a parallelogram in the domain of the kernel and thus the resulting ranges for z_1 and z_2 are dependent on each other.

Since we have the convex set

$$\begin{aligned}\Gamma(\boldsymbol{\beta}^p) &= \{I\boldsymbol{\beta}^p + K\boldsymbol{z} \mid C\boldsymbol{\beta}^p = \mathbf{0} \wedge I\boldsymbol{\beta}^p \in \mathbb{Z}^{D_f} \wedge \boldsymbol{z} \in \Sigma(\boldsymbol{\beta}^p)\} \\ &= \{I\boldsymbol{\beta}^p + K\boldsymbol{z} \mid C\boldsymbol{\beta}^p = \mathbf{0} \wedge \boldsymbol{z} \in \Sigma(\boldsymbol{\beta}^p)\}\,,\end{aligned} \tag{4.39}$$

where we were able to drop the integer condition on $I\boldsymbol{\beta}^p$, because it is redundant to $\Sigma(\boldsymbol{\beta}^p)$

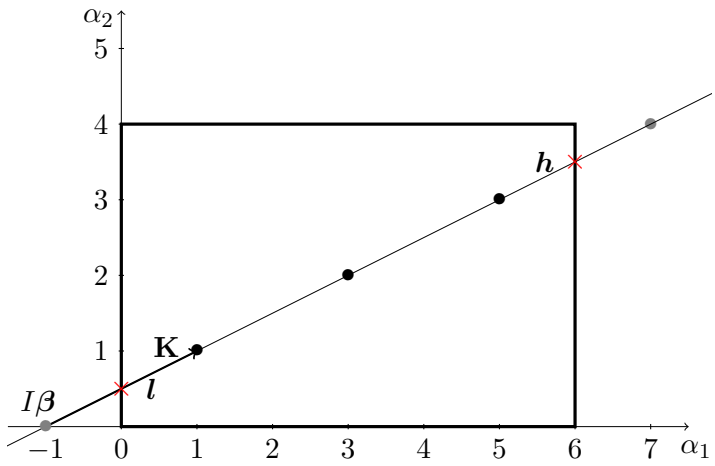


Figure 4.3: A one-dimensional parameter index $\boldsymbol{\beta}$ driven by a two-dimensional function index $\boldsymbol{\alpha}$ shown in $\boldsymbol{\alpha}$-space. The one-dimensional index of x is given by $\boldsymbol{\beta} = A\boldsymbol{\alpha}$ with $A = \begin{pmatrix}1 & -2\end{pmatrix}$. This yields $\boldsymbol{\alpha} = I\boldsymbol{\beta} + K\boldsymbol{z}$ with the pseudo-inverse $I^T = \begin{pmatrix}1 & 0\end{pmatrix}$ and one-dimensional kernel $K^T = \begin{pmatrix}2 & 1\end{pmatrix}$. For $\boldsymbol{\beta} = \begin{pmatrix}-1\end{pmatrix}$ the set of possible values for $\boldsymbol{\alpha}$ lies on the marked line with direction vector given by the kernel K. This set is limited by the requirement that $\boldsymbol{\alpha}$ must be integer, thus only the marked points on the line are valid values for $\boldsymbol{\alpha}$. Furthermore the constraint (4.36) imposed by the range of $\boldsymbol{\alpha}$ requires valid values to lie between the points marked $\boldsymbol{l}$ and $\boldsymbol{h}$. Thus values for z as allowed by (4.38) are $\Sigma(\begin{pmatrix}-1\end{pmatrix}) = \{z \in \mathbb{Z} \mid 1 \leq z \leq 3\}$, corresponding to the three points on the line inside the rectangle.

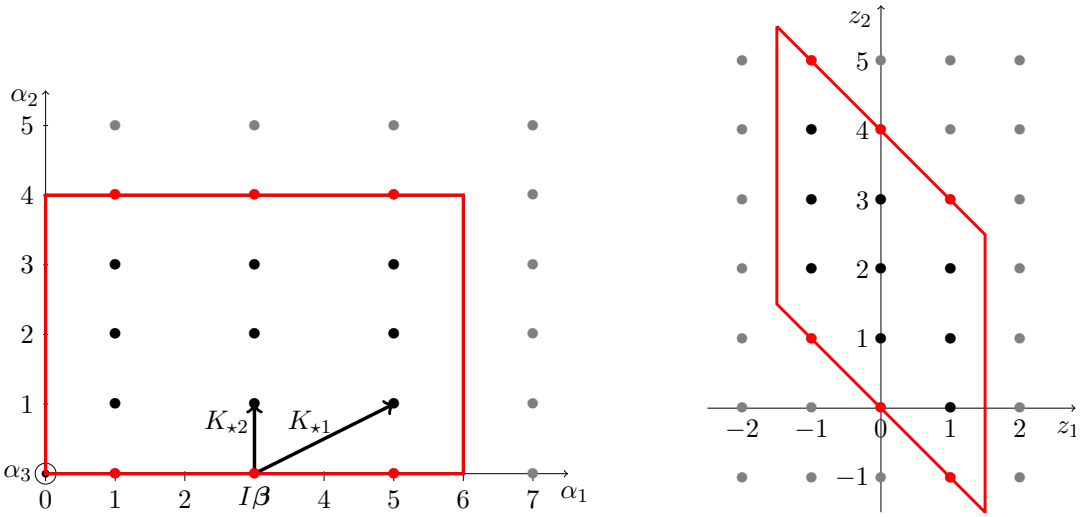
being not empty, we can now expand the sum (4.32) and thus write down an explicit expression for $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$. This gives

$$\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \delta_{C\boldsymbol{\beta}^p} \sum_{z_\kappa=\lceil\max(L^\kappa\boldsymbol{b})\rceil}^{\lfloor\min(H^\kappa\boldsymbol{b})\rfloor} \sum_{z_{\kappa-1}=\lceil\max(L^{\kappa-1}\boldsymbol{b}+\widehat{L}^{\kappa-1}\boldsymbol{z}_\kappa)\rceil}^{\lfloor\min(H^{\kappa-1}\boldsymbol{b}+\widehat{H}^{\kappa-1}\boldsymbol{z}_\kappa)\rfloor} \cdots \sum_{z_1=\lceil\max(L^1\boldsymbol{b}+\widehat{L}^1\boldsymbol{z}_{2\ldots\kappa})\rceil}^{\lfloor\min(H^1\boldsymbol{b}+\widehat{H}^1\boldsymbol{z}_{2\ldots\kappa})\rfloor} \mathrm{d}f_{I\boldsymbol{\beta}^p+K\boldsymbol{z}} \left.\frac{\partial\overline{f}}{\partial x^p_{A^p\boldsymbol{\alpha}}}\right|_{\boldsymbol{\alpha}=I\boldsymbol{\beta}^p+K\boldsymbol{z}}, \tag{4.40}$$

in which no Kronecker delta occurs within the sums and thus all iterations are utilized. The evaluation of the sums can be parallelized without difficulty and no synchronization is necessary for writes to $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ since in this form one thread can be used per element of $\mathrm{d}x^p$.

4.4.2 Handling Expressions Containing Sums

As mentioned earlier we also want to handle expressions that contain summations over one or more indices. For this purpose consider a function containing a summation depending on



(a) in $\boldsymbol{\alpha}$-space (b) in $\boldsymbol{z}$-space

Figure 4.4: A one-dimensional parameter index $\boldsymbol{\beta}$ driven by a three-dimensional function index $\boldsymbol{\alpha}$. (a) This shows $\boldsymbol{\alpha}$-space as a cut through the α_1-α_2 plane, i.e. the α_3-axis is perpendicular to this drawing. The one-dimensional index of x is given by $\boldsymbol{\beta} = A\boldsymbol{\alpha}$ with $A = \begin{pmatrix} 1 & -2 & -2 \end{pmatrix}$. This yields $\boldsymbol{\alpha} = I\boldsymbol{\beta} + K\boldsymbol{z}$ with the pseudo-inverse $I = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$. A possible choice for the two-dimensional kernel is $K = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 2 & 1 \end{pmatrix}$. For $\boldsymbol{\beta} = \begin{pmatrix} 3 \end{pmatrix}$ the set of possible values for $\boldsymbol{\alpha}$ is given by the sum of $I\boldsymbol{\beta}$ and integer linear combinations of the columns of the kernel matrix K. The constraint (4.36) imposed by the range of $\boldsymbol{\alpha}$ requires valid values to lie inside the red rectangle. (b) By mapping this rectangle into the domain of the kernel, i.e. $\boldsymbol{z}$-space, we obtain a parallelogram. Thus values for $\boldsymbol{z}$ as allowed by (4.38) are the integer points that lie within this parallelogram, i.e. $\Sigma(\begin{pmatrix} 3 \end{pmatrix}) = \{z \in \mathbb{Z} \mid -1 \leq z_2 \leq 5 \wedge \max(-1, -z_2) \leq z_1 \leq \min(1, 4 - z_2)\}$ corresponding to the 15 points inside the rectangle in $\boldsymbol{\alpha}$-space. This causes the range of z_1 to become dependent on the value of z_2.

arguments $y^1, \ldots, y^{P'}$. It can be written in the form

$$f_{\boldsymbol{\alpha}}(y^1, \ldots, y^{P'}, x^1, \ldots, x^P) = \overline{f}\left(\overline{s}(y^1, \ldots, y^{P'}), x^1_{A^1\boldsymbol{\alpha}}, \ldots, x^P_{A^P\boldsymbol{\alpha}}\right) \tag{4.41}$$

with

$$\overline{s}(y^1, \ldots, y^{P'}) = \sum_{k \in \Psi} s(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \ldots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}) \tag{4.42}$$

where $s : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}$ and $\widehat{\boldsymbol{\alpha}} \triangleq \begin{bmatrix}\boldsymbol{\alpha} & k\end{bmatrix}$ and $\widehat{A}^{p'} : \mathbb{Z}^{D_f+1} \to \mathbb{Z}^{D_{p'}}$ and $\Psi \subset \mathbb{Z}$ is a convex integer set. Using the chain rule to calculate the derivative of l (defined as before) w.r.t. $y^{p'}_{\boldsymbol{\beta}^{p'}}$ gives

$$\begin{aligned}
\mathrm{d}y^{p'}_{\boldsymbol{\beta}^{p'}} = \frac{\partial l}{\partial y^{p'}_{\boldsymbol{\beta}^{p'}}} &= \sum_{\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f} \frac{\partial l}{\partial f_{\boldsymbol{\alpha}}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial y^{p'}_{\boldsymbol{\beta}^{p'}}} = \sum_{\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial \overline{s}} \sum_{\substack{k \in \Psi \\ \widehat{A}^{p'}\widehat{\boldsymbol{\alpha}} = \boldsymbol{\beta}^{p'}}} \frac{\partial s}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}} \\
&= \sum_{\substack{\mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f \\ k \in \Psi \\ \widehat{A}^{p'}\widehat{\boldsymbol{\alpha}} = \boldsymbol{\beta}^{p'}}} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \overline{f}}{\partial \overline{s}} \frac{\partial s}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}} = \sum_{\widehat{\boldsymbol{\alpha}} \in \widehat{\Gamma}} \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial \widehat{f}}{\partial y^{p'}_{\widehat{A}^{p'}\widehat{\boldsymbol{\alpha}}}}
\end{aligned} \tag{4.43}$$

with the "sum-liberated" scalar function

$$\widehat{f}(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \ldots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}, x^1_{A^1\boldsymbol{\alpha}}, \ldots, x^P_{A^P\boldsymbol{\alpha}}) \triangleq \overline{f}\left(s(y^1_{\widehat{A}^1\widehat{\boldsymbol{\alpha}}}, \ldots, y^{P'}_{\widehat{A}^{P'}\widehat{\boldsymbol{\alpha}}}), x^1_{A^1\boldsymbol{\alpha}}, \ldots, x^P_{A^P\boldsymbol{\alpha}}\right) \tag{4.44}$$

and the "sum-extended" multi-index set

$$\widehat{\Gamma} \triangleq \left\{ \begin{bmatrix}\boldsymbol{\alpha} & k\end{bmatrix} \,\middle|\, \boldsymbol{\alpha} \in \mathbb{Z}^{D_f} \wedge k \in \Psi \wedge \mathbf{1} \leq \boldsymbol{\alpha} \leq \boldsymbol{N}^f \wedge \widehat{A}^{p'} \begin{bmatrix}\boldsymbol{\alpha} & k\end{bmatrix} = \boldsymbol{\beta}^{p'} \right\}. \tag{4.45}$$

Note that (4.43) equals the original expression for the derivative (4.32) but with $\overline{f}$ replaced by $\widehat{f}$, which is the same as $\overline{f}$ but with the sum symbol removed, and Γ replaced by $\widehat{\Gamma}$, which additionally includes the conditions on k from the summation range.

Thus handling summations can be done using the previously described strategy for derivation by extending it as follows. Each sum symbol (!) in the function f to be derived is removed, its summation index is appended to the multi-index $\boldsymbol{\alpha}$ of f and its summation range is included as an additional constraint in the set Γ. This process is iterated for nested sums. When indexing into $\mathrm{d}f$ the additional indices in $\boldsymbol{\alpha}$ introduced by sums are ignored.

4.4.3 Elementwise Derivation Algorithm

Combining the concepts described so far algorithm 11 computes expressions for derivatives $\mathrm{d}x^p_{\boldsymbol{\beta}^p} = \partial l / \partial x^p_{\boldsymbol{\beta}^p}$ of a elementwise defined function f.

Since the summation ranges (4.38) in the produced derivative are of the same form as the index ranges (4.36) of the input function and we have shown in section 4.4.2 how to handle summations in the input function, we can iteratively apply the derivation algorithm on derivative functions to obtain second and higher order derivatives. Therefore the set of elementwise defined functions using linear combination of indices for its arguments is closed under the operation of derivation.

If an explicit expression for the Jacobian $\partial f_{\boldsymbol{\alpha}'} / \partial x^p_{\boldsymbol{\beta}^p}$ is desired, it can be obtained from $\mathrm{d}x^p_{\boldsymbol{\beta}^p}$ by substituting

$$\mathrm{d}f_{\boldsymbol{\alpha}} \triangleq \prod_d \delta_{\alpha_d, \alpha'_d}$$

into it.

Algorithm 11: Elementwise expression derivation

Input: elementwise defined tensor-valued function f taking P tensor arguments $x^1, \ldots, x^P$; expression of derivative $\mathrm{d}f_{\boldsymbol{\alpha}} \triangleq \partial l / \partial f_{\boldsymbol{\alpha}}$

Output: expression of derivatives w.r.t. arguments $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \triangleq \partial l / \partial x^p_{\boldsymbol{\beta}^p}$

for $p \in \{1, \ldots, P\}$ **do** // loop over arguments x^p

  $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \longleftarrow 0$

  for $q \in \{1, \ldots, Q_p\}$ **do** // loop over index expressions for x^p

    // compute derivative expression w.r.t. $x^p_{A^{pq}\boldsymbol{\alpha}}$ using reverse accumulation automatic differentiation (sec. 4.1)

    $\Delta \longleftarrow \mathrm{d}f_{\boldsymbol{\alpha}} \frac{\partial f_{\boldsymbol{\alpha}}}{\partial x^p_{A^{pq}\boldsymbol{\alpha}}}$ // ignore sum symbols within f

    // compute range constraints from shape of f and limits of occurring sums

    $\Omega \longleftarrow \{\text{range constraints on } \boldsymbol{\alpha} \text{ of the form } R\boldsymbol{\alpha} \geq r\}$

    // compute Smith normal form (sec. 4.3.1) to obtain the following

    $I \longleftarrow$ integer pseudo-inverse of A^{pq}

    $K \longleftarrow$ integer kernel of A^{pq}

    $C \longleftarrow$ integer cokernel of A^{pq}

    // rewrite constraints using $\boldsymbol{\beta}^p$ and kernel factors $\boldsymbol{z}$

    $\Omega' \longleftarrow \{R\,K\boldsymbol{z} \geq r - R\,I\boldsymbol{\beta}^p \mid (R\boldsymbol{\alpha} \geq r) \in \Omega\}$

    // solve Ω' for $\boldsymbol{z}$ using Fourier-Motzkin elimination (sec. 4.3.2)

    $\Sigma \longleftarrow \{\text{range constraints } \Omega' \text{ on } \boldsymbol{z} \text{ transformed into form (4.38)}\}$

    // generate derivative expressions

    $\mathrm{d}x^p_{\boldsymbol{\beta}^p} \longleftarrow \mathrm{d}x^p_{\boldsymbol{\beta}^p} + \delta_{C\boldsymbol{\beta}^p} \sum_{\boldsymbol{z} \in \Sigma} \Delta|_{\boldsymbol{\alpha} = I\boldsymbol{\beta}^p + K\boldsymbol{z}}$ // use form (4.40) for sum

4.5 Example and Numeric Verification

In our implementation and thus in this example we use zero-based indexing, i.e. a vector $x \in \mathbb{R}^N$ has indices $\{0, \ldots, N-1\}$, as it is usual in modern programming languages. Given the function

$$f_{ij}(a, b, c, \boldsymbol{d}) = \exp\left[-\sum_{k=0}^{4}\left(\left(a_{ik} + b_{jk}\right)^2 c_{ii} + d_{i+k}^3\right)\right]$$

where the shapes of the arguments are $a \in \mathbb{R}^{3\times5}$, $b \in \mathbb{R}^{4\times5}$, $c \in \mathbb{R}^{3\times3}$ and $d \in \mathbb{R}^8$ and the shape of the function is $f \in \mathbb{R}^{3\times4}$ the derivation algorithm produces the following output:

```
Input: f[i; j] = exp (-sum{k}_0^4 (((a[i; k] + b[j; k]) ** 2 * c[i; i] + d[i + k] ** 3)))
Derivative of f wrt. a: da[da_0; da_1] = sum{da_z0}_0^3 (((-(df[da_0; da_z0] * exp (-sum{k}_0
    ^4 (((a[da_0; k] + b[da_z0; k]) ** 2 * c[da_0; da_0] + d[da_0 + k] ** 3))))) * c[da_0;
    da_0] * 2 * (a[da_0; da_1] + b[da_z0; da_1]) ** (2 - 1)))
Derivative of f wrt. b: db[db_0; db_1] = sum{db_z0}_0^2 (((-(df[db_z0; db_0] * exp (-sum{k}_0
    ^4 (((a[db_z0; k] + b[db_0; k]) ** 2 * c[db_z0; db_z0] + d[db_z0 + k] ** 3))))) * c[db_z0;
     db_z0] * 2 * (a[db_z0; db_1] + b[db_0; db_1]) ** (2 - 1)))
Derivative of f wrt. c: dc[dc_0; dc_1] = if {dc_0 + -dc_1 = 0} then (sum{dc_z1}_0^4 (sum{dc_z0}
    _0^3 (((a[dc_1; dc_z1] + b[dc_z0; dc_z1]) ** 2 * (-(df[dc_1; dc_z0] * exp (-sum{k}_0^4 ((( 
    a[dc_1; k] + b[dc_z0; k]) ** 2 * c[dc_1; dc_1] + d[dc_1 + k] ** 3)))))))) else (0)
Derivative of f wrt. d: dd[dd_0] = sum{dd_z1}_(max [0; -2 + dd_0])^(min [4; dd_0]) (sum{dd_z0}
    _0^3 (((-(df[dd_0 + -dd_z1; dd_z0] * exp (-sum{k}_0^4 (((a[dd_0 + -dd_z1; k] + b[dd_z0; k
    ]) ** 2 * c[dd_0 + -dd_z1; dd_0 + -dd_z1] + d[dd_0 + -dd_z1 + k] ** 3))))) * 3 * d[dd_0]
    ** (3 - 1))))
```

The operator ** denotes exponentiation in this output. The Kronecker delta has been encoded as a "*if* x *then* y *else* z" expression for more efficiency.

Internally these expressions are represented as graphs, thus subexpressions occurring multiple times are only stored and evaluated once and no expression blowup as with symbolic differentiation occurs. To cleanup the generated expressions from the automatic differentiation algorithm an expression optimization step, which pre-evaluates constant parts of the expressions, should be incorporated. However, since this is not part of the core derivation problem, it has not been performed for this demonstration.

These derivative expressions have been verified by using random numeric values for the arguments and comparing the resulting values for the Jacobians with results from numeric differentiation.

4.6 Discussion

We have presented a method to compute symbolic expressions for derivatives of any order of elementwise defined tensor-valued functions. These functions may contain summations and the indices of its arguments can be an arbitrary linear combination of the function indices. The output of our algorithm is an explicit symbolic expression for each element of the derivative.

Thus the resulting expressions are very well suited for massively parallel evaluation in a lock- and synchronization-free computational kernel, which runs on a massively parallel GPU and computes one element of the derivative per thread. No temporary memory is necessary for the evaluation of the derivatives.

The derivatives themselves may contain additional summations over indices which have become free in the derivative. The output of the algorithm specifies the ranges of these sums as a maximum or minimum over a set of linear combinations of the derivative indices; therefore computing the numerical range at evaluation time costs only two matrix multiplications per loop run (not iteration).

The method presented in this chapter is employed together with a code generator for elementwise defined functions within an efficient implementation of the models that will be introduced in chapter 5.

Chapter 5

Gaussian Process Neurons

The objective of this chapter is to introduce a neural activation function that can be learned completely from training data alongside the weights of the neural network. The number of constraints on the form of the learnable functions should be kept as low as possible to allow the highest amount of flexibility. However, we want the neural network to be trainable using stochastic gradient descent, thus we require the activation function to be at least continuously differentiable. Furthermore, it is a fundamental principle of statistics that by increasing the flexibility of a model the risk of overfitting is also increased. To keep that risk in check the model needs to regularized. Here we apply the Bayesian approach to regularization by putting a prior over the space of activation functions. We choose a GP with a zero mean and SE covariance function for that prior, since it encourages smooth functions of small magnitude. This probabilistic treatment transforms a neuron into a probabilistic unit, which we call Gaussian process neuron (GPN). Consequently a neural network built from GPNs becomes a probabilistic graphical model, which can intrinsically handle training and test data afflicted with uncertainties and estimate the confidence of its own predictions.

This chapter is structured as follows. First, we introduce the basic GPN unit. Then we show how integrating it into a feed-forward neural network transforms the network into a probabilistic graphical model. Since a GP is a non-parametric model it introduces dependencies between samples into the model. Consequently, even after learning the network weights, the predictions on a test input directly depend on the training samples and inference has to be performed using Monte Carlo methods. Since this is impractical, we introduce the parametric GPN, an auxiliary model that applies methods from sparse GP regression to represent its activation function using a set of trainable parameters. The resulting model approximation can be trained using stochastic gradient descent by sampling the gradient.

Then we will combine ideas from fast Dropout and sparse variational GPs to make a *non-parametric* GPN network trainable using variational Bayesian inference. The result is a fully

deterministic loss function that eliminates the need to sample the gradient and thus makes training vastly more efficient. Although the model is treated fully probabilistically, the loss function retains the functional structure of a neural network, making it directly applicable to network architectures such as RNNs and CNNs.

Since literature shows that monotonicity has proven favorable for the generalization ability of a neural network in some cases, we present methods to constrain GPNs to monotonic activation functions. Preliminary benchmark and experimental results on regression and classification tasks are reported at the end of the chapter.

A chart showing the family of GPN models, their relationships and inference methods is presented in fig. 6.1. Furthermore, the relation to deep Gaussian processes, which is a model that likewise combines GPs in a probabilistic graphical model, is established in section 6.1.

Contributions to this Chapter

The GPN model and methods for its efficient training were proposed and designed by me. Implementation was performed by Marcus Basalla and me. Benchmarking and experiments were done by Marcus Basalla and published in (Basalla, 2017).

5.1 The Gaussian Process Neuron

A Gaussian process neuron (GPN) is a probabilistic unit that receives multiple inputs and computes an output distribution conditioned on the values of its inputs. Given multiple input samples the output samples of a GPN become correlated, i.e. the output distribution is *not* iid. over the samples. A probabilistic graphical model corresponding to an instance of a GPN for a fixed number of inputs and samples is shown in fig. 5.1.

The input to a GPN is a set of random variables denoted by Y_{sm}, $s \in \{1, \ldots, S\}$, $m \in \{1, \ldots, M\}$, where s is the sample and m is the index of the input. The random variables A_s, $s \in \{1, \ldots, S\}$, are called activations and they are given by the dot product of the inputs with the weight vector $\boldsymbol{w} \in \mathbb{R}^M$. The weight vector is not probabilistic. Furthermore, we do not specify a bias term explicitly, since it can be absorbed into the weight matrix and input distribution, if necessary. The conditional distribution of A_s can be written[1] as

$$\mathrm{P}(A_s \,|\, Y_{s\star}) = \delta(A_s - \boldsymbol{w}^T Y_{s\star}) \,, \qquad s \in \{1, \ldots, S\} \,, \tag{5.1}$$

[1]The star ($\star$) performs tensor slicing as described in section 2.1.

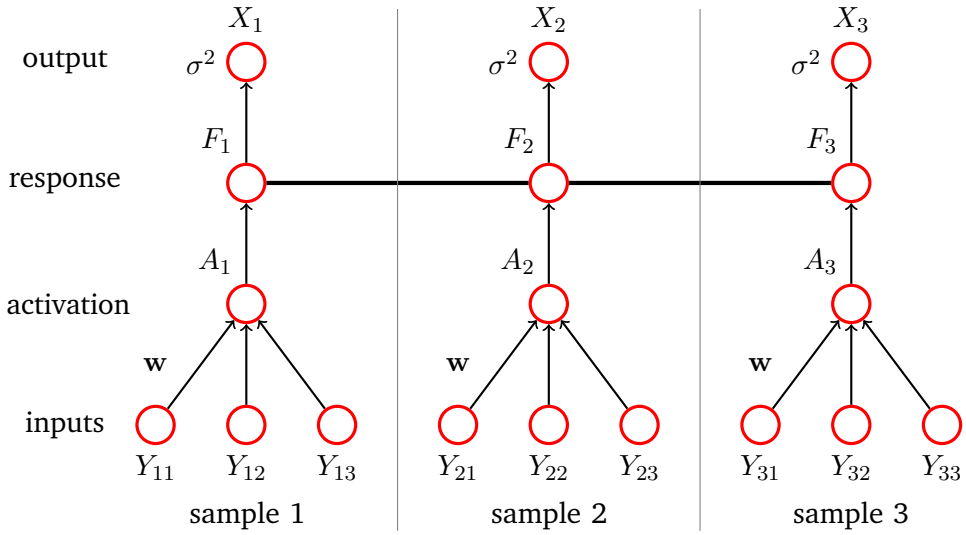


Figure 5.1: A GPN with three inputs is shown for three samples as a probabilistic graphical model. The inputs are represented by the random variables $Y_{sm}, s \in \{1,2,3\}, m \in \{1,2,3\}$. The activations for each sample are represented by the random variables $A_s, s \in \{1,2,3\}$, and depend deterministically on the inputs. The responses $F_\star$ are a GP over the samples conditioned on the activations; in the figure the GP is represented by a Markov random field shown as an undirected connection between all samples. The outputs are represented by the random variables X_s, $s \in \{1,2,3\}$.

where $\delta(t)$ is the delta distribution, defined by

$$\int_A^B \delta(t)\,\mathrm{d}t \triangleq \begin{cases} 1 & \text{if } A \leq 0 \leq B \\ 0 & \text{otherwise} \end{cases} . \tag{5.2}$$

The random variables F_s, $s \in \{1,\ldots,S\}$, represent the responses of the GPN to the activations. The GPN does not assume a fixed activation function, instead it imposes a GP prior on the activation function thus allowing it to be inferred from the data. Therefore, the response of sample s is given by $F_s = F(A_s)$ where the distribution over the activation functions $F(A)$ is given by

$$F \mid A \sim \mathcal{GP}(0, k(A, A'))\,, \tag{5.3}$$

where $k : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a valid covariance function for a one-dimensional GP, cf. section 2.4. Note, that the GP introduces correlations between the responses of different samples; thus the responses of a GPN are *not* independent. The GPN uses the SE covariance covariance function,

$$k(a, a') \triangleq \exp\bigl(-(a-a')^2\bigr)\,, \tag{5.4}$$

where we have omitted the lengthscale parameter, as it is redundant to a rescaling of the weights $\boldsymbol{w}$ in (5.1).

To make the conditional dependencies introduced by the GP more clear, it is helpful to explicitly consider a set of samples for eq. (5.3). Per definition of a GP for a finite number of samples, that equation is equivalent to

$$F_\star \,|\, A_\star \sim \mathcal{N}(\mathbf{0}, K)\,, \tag{5.5}$$

where $F_\star$ and $A_\star$ denote vectors of random variables, i.e., $F_\star = (F_1, F_2, \ldots, F_S)^T$ and $A_\star = (A_1, A_2, \ldots, A_S)^T$ respectively. The covariance matrix $K \in \mathbb{R}^{S\times S}$ is given by

$$K_{ss'} \triangleq k(A_s, A_{s'})\,. \tag{5.6}$$

Thus, as shown in fig. 5.1, the responses F_s, $s \in \{1, \ldots, S\}$, form a Markov random field with full connectivity between all samples. *All* samples of *each* GPN share the same Gaussian process and hence the same activation function.[2]

Each output is represented by the random variable X_s, $s \in \{1, \ldots, S\}$, which is a noisy version of the corresponding response F_s. Its conditional distribution is given by

$$X_s \,|\, F_s \sim \mathcal{N}(F_s, \sigma^2)\,, \qquad s \in \{1, \ldots, S\}\,, \tag{5.7}$$

where $\sigma \geq 0$ specifies the standard deviation of the superimposed Gaussian noise. This can be used to introduce additional noise into the output of a GPN. If this is not desired, σ is set to zero and the output equals the response. While the superimposed noise is iid. for each sample, the outputs X are not iid. due to the propagation of the correlation of the responses F between the samples. This concludes the definition of a Gaussian process neuron.

Let us briefly compare a GPN unit to a conventional neuron in an artificial neural network, as introduced in section 2.5. A conventional neuron computes its output x using the deterministic function $x = \sigma(\boldsymbol{w}^T\boldsymbol{y})$, where $\boldsymbol{y}$ is the vector of the neuron's inputs and $\boldsymbol{w}$ are its weights. The term $a \triangleq \boldsymbol{w}^T\boldsymbol{y}$ is called activation and the (non-linear) function $\sigma(a)$ is called activation function. Both the GPN and the conventional neuron compute their activations in the same way, with the difference that, being a probabilistic model, the GPN deals with an activation distribution $P(A_s)$. Note that, the weights $\boldsymbol{w}$ are fully deterministic in the GPN model and the uncertainty in A_s arises fully from the uncertainty in the inputs Y_{sm}. The GPN replaces the fixed activation function $\sigma(a)$ of the conventional neuron by a distribution over functions

[2] If $F_\star \,|\, A_\star$ did not form a Markov random field over the samples, then each sample would use an *independent* and random activation function and no generalization to new samples would be possible.

$\mathrm{P}(F \mid A)$. This distribution is defined by a GP over the activations. Because the SE covariance function is used, each activation function sampled from $\mathrm{P}(F \mid A)$ is infinitely differentiable and thus smooth. Since a GP is non-parametric, it introduces dependencies between the samples, i.e. each GPN uses *one* GP to represent the activation function of *all* samples. Consequently, a GPN has components of a parametric model, e.g. the weights, and components of a non-parametric model, e.g. the distribution of activation functions.

5.1.1 Marginal Distribution

The GPN has been specified in terms of conditional probabilities for the activations, responses and outputs. In order to obtain a more convenient conditional distribution for the outputs X given the inputs Y, we marginalize over the random variables internal to the GPN, i.e. the activations A and responses F. Using (5.1) and (5.5) we obtain

$$\begin{aligned}
\mathrm{P}(F_\star \mid Y_\star) &= \int \mathrm{P}(F_\star \mid A_\star) \left(\prod_{s=1}^{S} \mathrm{P}(A_s \mid Y_{s\star}) \right) \mathrm{d}A_\star \\
&= \int \mathcal{N}(F_\star \mid \mathbf{0}, K) \left(\prod_{s=1}^{S} \delta(A_s - \boldsymbol{w}^T Y_{s\star}) \right) \mathrm{d}A_\star \\
&= \mathcal{N}(F_\star \mid \mathbf{0}, \widetilde{K}) \,, \qquad (5.8)
\end{aligned}$$

where $\widetilde{K}_{ss'} \triangleq \widetilde{k}(Y_{s\star}, Y_{s'\star})$ and the covariance function $\widetilde{k}(\boldsymbol{y}, \boldsymbol{y'})$ is given by

$$\widetilde{k}(\boldsymbol{y}, \boldsymbol{y'}) \triangleq k(\boldsymbol{w}^T \boldsymbol{y}, \boldsymbol{w}^T \boldsymbol{y'}) = \exp\left(-[\boldsymbol{w}^T \boldsymbol{y} - \boldsymbol{w}^T \boldsymbol{y'}]^2\right) \,. \qquad (5.9)$$

This function can be interpreted as follows. From the geometric definition of the dot product, $\boldsymbol{w}^T \boldsymbol{y} = |\boldsymbol{w}|\,|\boldsymbol{y}| \cos\theta$, where $|\bullet|$ is the 2-norm and θ is the angle between $\boldsymbol{w}$ and $\boldsymbol{y}$, we see that $\boldsymbol{w}^T \boldsymbol{y}$ in (5.9) is the length of the projection of $\boldsymbol{y}$ onto $\boldsymbol{w}$ scaled by $|\boldsymbol{w}|$. Consequently the GP defined by (5.8) encourages similar response values F_s to samples that have inputs with comparable projection lengths onto $\boldsymbol{w}$. The length $1/|\boldsymbol{w}|$ can be interpreted as the lengthscale of the SE covariance function and thus the magnitude of the weight vector determines how much samples of the activation function vary around the zero mean function.

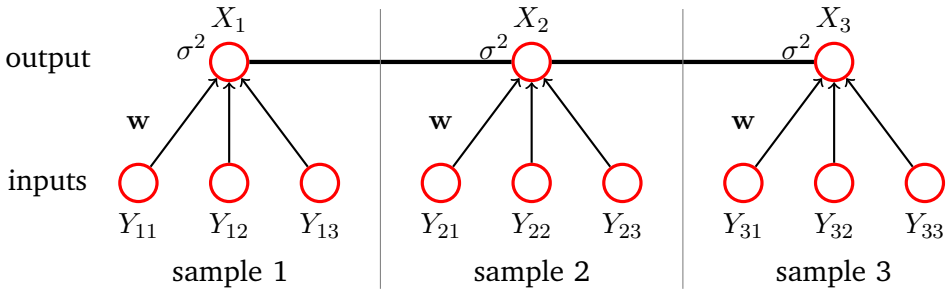


Figure 5.2: A probabilistic graphical model for the conditional distribution (5.10) of a GPN having three inputs is shown for three samples. The thick line between the output samples $X_\star$ indicates, that they are all dependent and form a (conditional) Markov random field. The parameters of a GPN are its weights $\boldsymbol{w}$ and variance σ^2.

We perform another marginalization using (5.7) to obtain the output distribution

$$\begin{aligned} \mathrm{P}(X_\star \,|\, Y_\star) &= \int \left(\prod_{s=1}^{S} \mathrm{P}(X_s \,|\, F_s) \right) \mathrm{P}(F_\star \,|\, Y_\star) \,\mathrm{d}F_\star \\ &= \int \left(\prod_{s=1}^{S} \mathcal{N}(X_s \,|\, F_s, \sigma^2) \right) \mathcal{N}(F_\star \,|\, \mathbf{0}, \widetilde{K}) \,\mathrm{d}F_\star \\ &= \mathcal{N}(X_\star \,|\, \mathbf{0}, \widetilde{K} + \sigma^2 \mathbb{1}) \,, \end{aligned} \tag{5.10}$$

with $\widetilde{K}$ defined as above and $\mathbb{1}$ being the identity matrix. The outputs of a GPN given its inputs are thus normally distributed *over the samples* with zero mean and an input-dependent covariance matrix. The equivalent graphical model is shown in fig. 5.2.

5.1.2 Building Layers

A set of GPNs that share the same inputs is called a GPN layer. This definition is equivalent to the definition of a layer of neurons in a conventional artificial neural networks. While combing conventional neurons into a layer is straightforward, in the case of GPNs, care must be taken because of the additional inter-sample dependencies. GPN layers can be stacked to form a multi-layer model similar to a feed-forward neural network.

Let the GPN layer l receive S samples of M-dimensional input $Y^l = X^{l-1}$ from the previous layer $l-1$ and let it produce N-dimensional output X^l. The input dimension is indexed using $m \in \{1, \ldots, N_{l-1}\}$ and the output dimension is indexed by $n \in \{1, \ldots, N_l\}$. As before the samples are enumerated by $s \in \{1, \ldots, S\}$. An instance of a GPN layer is shown as a graphical model in fig. 5.3a. Let the input that the layer receives be denoted by $Y^l_{sm} = X^{l-1}_{sm}$, $s \in \{1, \ldots, S\}$, $m \in \{1, \ldots, M\}$. Note, that we make no assumption on the distribution of the inputs

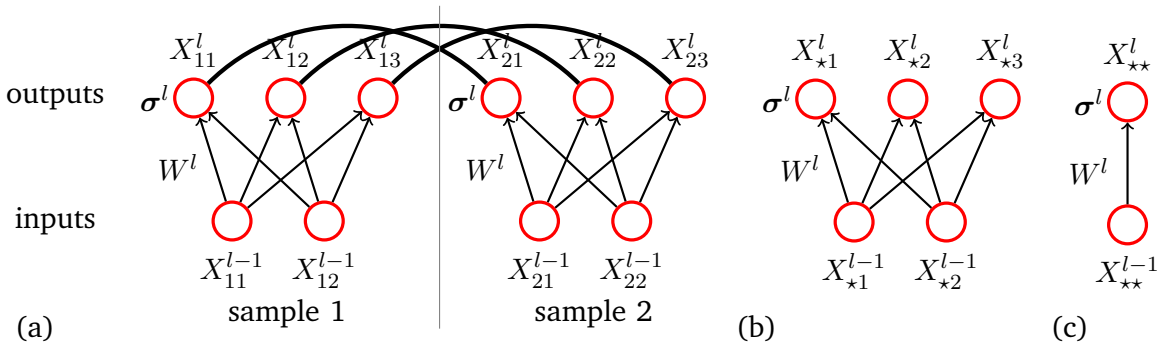


Figure 5.3: The conditional distribution of a layer of three GPNs with two inputs is shown for two samples as a probabilistic graphical model. Each GPN output $X^l_{\star n}$, $n \in \{1, 2, 3\}$, forms a Markov random field (represented by the thick undirected connection) that depends on all inputs $X^{l-1}_{\star\star}$ of all samples. The GPN layer is parameterized by the weight matrix W^l and the standard deviation vector $\boldsymbol{\sigma}^l$. (a) All GPNs and samples are shown as separate nodes. (b) All samples are consolidated into one node for each GPN. (c) All GPNs of a layer and all samples are consolidated into one node.

and they may be correlated within or between samples or both. The outputs are represented by the random variables X^l_{sn}, $s \in \{1, \ldots, S\}$, $n \in \{1, \ldots, N\}$ where the distribution of each GPN $X^l_{\star n}$, $n \in \{1, \ldots, N\}$, is given by (5.10). Thus, by combining the individual weights $\boldsymbol{w}$ of the GPNs into the weight matrix $W^l \in \mathbb{R}^{N \times M}$ and the standard deviations σ into the vector $\boldsymbol{\sigma}^l \in \mathbb{R}^N$, we have

$$\mathrm{P}(X^l_{\star\star} \,|\, X^{l-1}_{\star\star}) = \prod_{n=1}^{N} \mathcal{N}(X^l_{\star n} \,|\, \mathbf{0}, \widetilde{K}^l_n + (\sigma^l_n)^2 \mathbb{1}) \quad \text{with } \widetilde{K}^l_{n,ss'} \triangleq k(W^l_{n\star} X^{l-1}_{s\star}, W^l_{n\star} X^{l-1}_{s'\star}) \,. \tag{5.11}$$

For clarity of visualization the graphical model of a GPN layer can be condensed: In fig. 5.3a each node represents a particular input/output dimension and sample. In fig. 5.3b each node represents all samples, thus the Markov random is not visible in the graphical model. This is the representation most similar to a layer of a conventional neural network. In fig. 5.3c the graphical model is reduced to one input node and one output node, both representing all input/output dimensions and samples.

5.2 Probabilistic Feed-Forward Networks

Similar to conventional feed-forward ANNs, layers of GPNs can be stacked to form probabilistic models for regression and classification tasks. The resulting models are a hybrid of parametric and non-parametric models, because their predictions depend on learned parameters (weights),

but also directly on the training samples due to the inter-samples dependencies of GPs.

5.2.1 Regression

Given a set of $\widehat{S}$ training inputs $\widehat{X}^0 = \{\widehat{\boldsymbol{x}}_1^0, \ldots, \widehat{\boldsymbol{x}}_{\widehat{S}}^0\}$ with $\widehat{\boldsymbol{x}}_s^0 \in \mathbb{R}^{N_0}$ and corresponding targets $\widehat{Z} = \{\widehat{\boldsymbol{z}}_1, \ldots, \widehat{\boldsymbol{z}}_{\widehat{S}}\}$ with $\widehat{\boldsymbol{z}}_s \in \mathbb{R}^{N_Z}$, the regression problem is to find a function $\boldsymbol{f} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_Z}$ that assigns a prediction $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}^0)$ to an input $\boldsymbol{x}^0$. The prediction function $\boldsymbol{f}$ should minimize the loss $\mathscr{L}(\widehat{X}^0, \widehat{Z})$ with

$$\mathscr{L}(X, Z) \triangleq \frac{1}{S} \sum_{s=1}^{S} L(\boldsymbol{f}(\boldsymbol{x}_s^0), \boldsymbol{z}_s)\,, \tag{5.12}$$

where the loss measure $L : \mathbb{R}^{N_Z} \times \mathbb{R}^{N_Z} \to \mathbb{R}$ assigns a penalty $L(\boldsymbol{y}, \boldsymbol{z})$ based on the difference between the prediction $\boldsymbol{f}(\boldsymbol{x}^0)$ and the target value $\boldsymbol{z}$. A good prediction function must generalize well to previously unseen test inputs $\widetilde{X}^0 = \{\widetilde{\boldsymbol{x}}_1^0, \ldots, \widetilde{\boldsymbol{x}}_{\widetilde{S}}^0\}$ by correctly predicting their target values $\widetilde{Z} = \{\widetilde{\boldsymbol{z}}_1, \ldots, \widetilde{\boldsymbol{z}}_{\widetilde{S}}\}$ and thus having a small loss $\mathscr{L}(\widetilde{X}^0, \widetilde{Z})$ on the test set as well. A common loss measure for regression problems is the squared L^2-norm given by

$$L(\boldsymbol{y}, \boldsymbol{z}) = |\boldsymbol{y} - \boldsymbol{z}|^2 = \sum_d (y_d - z_d)^2\,. \tag{5.13}$$

It can be shown that this loss measure is proportional to the negative log-likelihood of $\boldsymbol{z}$ under a normal distribution with mean $\boldsymbol{y}$ and unit variance, i.e.

$$L(\boldsymbol{y}, \boldsymbol{z}) \propto -\log \mathcal{N}(\boldsymbol{z} \,|\, \boldsymbol{y}, \mathbb{1})\,. \tag{5.14}$$

Consequently, minimizing the loss given by eqs. (5.12) and (5.13) corresponds to finding the maximum likelihood solution of a probabilistic model that assumes a normal distribution for $\mathrm{P}(\boldsymbol{y} \,|\, \boldsymbol{x}^0)$.

Here, we use a stack of GPN layers to model the predictive distribution $\mathrm{P}(X^L \,|\, X^0)$. Like in a conventional feed-forward artificial neural networks the output of a GPN layer is fed as the input into the next GPN layer. The first layer receives the sample inputs X^0 and the last layer outputs the predictions X^L. A graphical model corresponding to a feed-forward GPN network is shown in fig. 5.4.

The random variable $\widehat{X}_{sn}^l$ denotes the n-th GPN in layer l corresponding to training sample s. Test samples are represented by the random variables $\widetilde{X}_{sn}^l$. The layer $l = 0$ corresponds to the inputs and its values are observed for both the training and test samples. The top-most layer $l = L$ represents the outputs. Since its values are observed for the training samples, we have $\widehat{X}_{s\star}^l = \widehat{\boldsymbol{z}}_s$, $s \in \{1, \ldots, \widehat{S}\}$. For defining the distributions occurring in the model, it is convenient

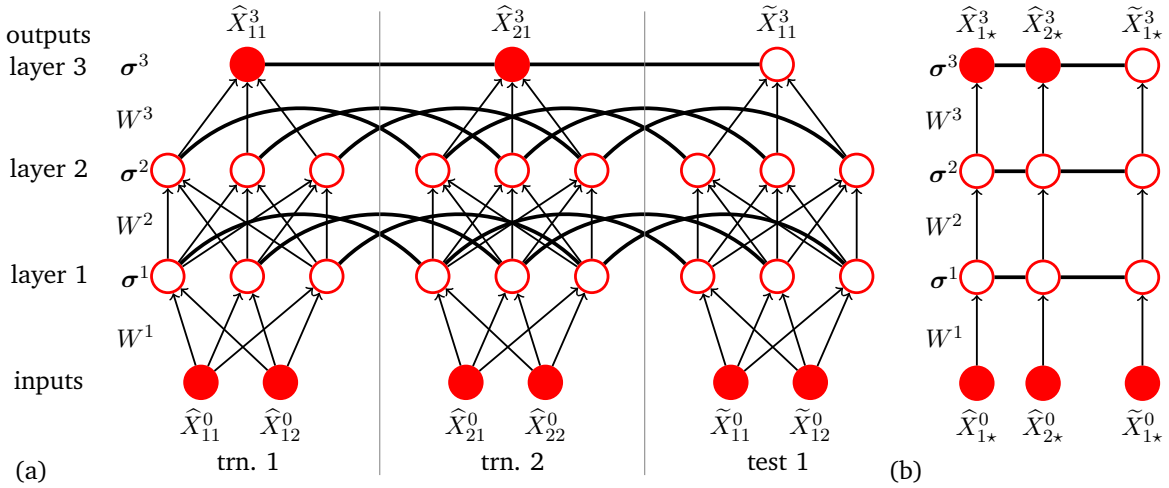


Figure 5.4: A feed-forward network of GPNs for a regression task with two-dimensional input and one-dimensional output is shown as a probabilistic graphical model for two training samples and one test sample. The network consists of an input layer and three layers of GPNs with the top-most layer representing the predictions of the model. All inputs and the training targets are observed (represented by filled circles). For performing regression the distribution over the test output $\widetilde{X}^3_{11}$ must be inferred. Panel (a) shows one random variable X^l_{sn} per GPN and sample. For a more compact representation, in panel (b) each random variable represents *all* GPNs $X^l_{s\star}$ of a layer and sample.

to concatenate the training and test samples of all inputs and GPNs, i.e.

$$X^l_{\star n} = \begin{bmatrix} \widehat{X}^l_{\star n} \\ \widetilde{X}^l_{\star n} \end{bmatrix}, \quad l \in \{0, \ldots, L\},\, n \in \{1, \ldots, N_l\}\,. \tag{5.15}$$

The joint distribution of the GPN feed-forward network for regression is given by

$$\mathrm{P}(X^1_{\star\star}, \ldots, X^L_{\star\star} \mid X^0_{\star\star}) = \prod_{l=1}^{L} \mathrm{P}(X^l_{\star\star} \mid X^{l-1}_{\star\star}) \tag{5.16}$$

where the conditional layer probabilities $\mathrm{P}(X^l_{\star\star} \mid X^{l-1}_{\star\star})$ are given by (5.11).

5.2.2 Training and Inference

We have shown how to build a regression model using GPN layers. It is also possible to build a classification model using GPNs, by using the same structure as for regression, feeding the network output distribution through the softmax function (2.92) and interpreting the result as class probabilities of a categorical distribution (2.93).

As we see GPN models for regression and classification are similar to conventional ANN mod-

els. Both contain parameters (weights) that must be fitted on the training set. However, the GPN models also introduce dependencies between samples, which does not occur in conventional ANNs where the predictions for all samples are independent and identical distributed. Consequently, GPN models as presented above are a hybrid between parametric and non-parametric models and their predictions on test samples depend directly on the training samples. Later we will show how a GPN model can be approximated by a fully parameterized model thus dropping the direct dependency on the training samples and making its predictions independent and identical distributed.

For now, however, we stay with the exact model and show how to obtain predictions on a test set. Naturally, predictions on the test set follow the distribution $\mathrm{P}_\theta(\widetilde{X}^L \mid \widetilde{X}^0, \widehat{X}^0, \widehat{X}^L)$, where we wrote P_θ to express that this distribution depends on the model parameters $\theta \triangleq \{\theta^1, \ldots, \theta^L\}$ consisting of the layer parameters $\theta^l \triangleq \{W^l, \boldsymbol{\sigma}^l\}$. To obtain these predictions, two steps are necessary.

First, the parameters θ of the model need to be estimated from the training data. A possible way to handle these parameters is to treat them as additional random variables and marginalize them out. To do so it would become necessary to introduce a prior for each of these parameters. While this is sound, it has the drawback that by making the parameters uncertain, additional stochasticity that is redundant to that of the probabilistic activation function of each GPN is introduced. Also, it would introduce complexity into the model which we wish to avoid at this point. Thus we keep all parameters θ deterministic and find a point estimate for their best values by maximizing their likelihood $P_\theta(\widehat{X}^L \mid \widehat{X}^0)$ under the training set. As we will see, this likelihood is intractable, since it involves marginalization over the inner layers $l \in \{1, \ldots, L-1\}$, which cannot be done analytically due to the occurrence of integrals that cannot be computed. However, it is possible to approximate the likelihood function and its derivatives w.r.t. θ by MCMC methods.

Second, the predictive distribution $\mathrm{P}_\theta(\widetilde{X}^L \mid \widetilde{X}^0, \widehat{X}^0, \widehat{X}^L)$ needs to be calculated. Unfortunately, it is also intractable due to the same reason that the likelihood is intractable. Thus, to obtain predictions on the test set, we have to employ MCMC methods here as well. Particularly, we will sample training values $\widehat{X}^l$ of the *inner* GPN layers from $\mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \mid \widehat{X}^0, \widehat{X}^L)$ and equipped with these samples, we can sample the test values $\widetilde{X}^l$ of each layer conditioned on training values $\widehat{X}^l$ from the distributions $\mathrm{P}_\theta(\widetilde{X}^l \mid \widetilde{X}^{l-1}, \widehat{X}^l, \widehat{X}^{l-1})$. We will show that this method produces unbiased samples from the predictive distribution.

Both steps and the employed sampling methods are explained in detail in the following sections.

Fitting the Model Parameters

To find a good point estimate for the model parameters θ, we calculate their likelihood under the training data and maximize it w.r.t. the parameters. For the regression task the likelihood of the parameters on the training set is given by

$$\begin{aligned}\mathcal{L}(\theta) \triangleq \mathrm{P}_\theta(\widehat{X}^L \mid \widehat{X}^0) &= \int \cdots \int \mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^L \mid \widehat{X}^0)\, \mathrm{d}\widehat{X}^1 \cdots \mathrm{d}\widehat{X}^{L-1} \\ &= \int \cdots \int \left(\prod_{l=1}^{L} \mathrm{P}_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1}) \right) \mathrm{d}\widehat{X}^1 \cdots \mathrm{d}\widehat{X}^{L-1}\,. \end{aligned} \tag{5.17}$$

with $\mathrm{P}_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1})$ given by (5.11). Applying the logarithm does not result in a simplified expression for the log-likelihood since the marginalization over the latent variables prevents the likelihood to factorize into a product over the training samples. The marginalizations in eq. (5.17) cannot be performed analytically, since $\widehat{X}^{l-1}$ enters the PDF through the highly non-linear inverse of the covariance matrix given by (5.11).

Thus an analytic maximization of the likelihood is impossible and we will use an iterative optimization procedure, for example gradient descent introduced in section 2.3.1, to maximize the likelihood. For that, it is necessary to compute the gradients of $\mathcal{L}$ w.r.t. the parameters θ. Using the form (2.28) for the PDF of the multivariate normal distribution, we obtain for the derivative w.r.t. a parameter $\varphi^l \in \{W^l, \boldsymbol{\sigma}^l\}$ of the l-th layer:

$$\frac{\partial \mathcal{L}}{\partial \varphi^l} = \int \cdots \int \left(\prod_{l'=1}^{L} \mathrm{P}_{\theta^{l'}}(\widehat{X}^{l'} \mid \widehat{X}^{l'-1}) \right) \rho'_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1})\, \mathrm{d}\widehat{X}^1 \cdots \mathrm{d}\widehat{X}^{L-1} \tag{5.18}$$

where $\rho_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1}) \triangleq \log \mathrm{P}_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1})$ and $\rho'_{\theta^l} \triangleq \partial \rho_{\theta^l} / \partial \varphi^l$. Consequently the derivatives can be represented as expectations under the joint distribution of the feed-forward GPN network,

$$\frac{\partial \mathcal{L}}{\partial \varphi^l} = \mathrm{E}_{\mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \mid \widehat{X}^0)} \left[\rho'_{\theta^l}(\widehat{X}^l \mid \widehat{X}^{l-1})\, \mathrm{P}_{\theta^L}(\widehat{X}^L \mid \widehat{X}^{L-1}) \right]. \tag{5.19}$$

This expectation can be approximated by sampling from $\mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \mid \widehat{X}^0)$ using MCMC methods. Since by definition of the GPN feed-forward network (5.16) the GPN layers form a Markov chain, unbiased samples can be obtained by sequentially sampling from the conditional normal densities $\mathrm{P}(\widehat{X}^1 \mid \widehat{X}^0)$ followed by $\mathrm{P}(\widehat{X}^2 \mid \widehat{X}^1)$ and so on up to $\mathrm{P}(\widehat{X}^{L-1} \mid \widehat{X}^{L-2})$. Sampling these conditionals is straightforward since their distributions are multivariate normals. Note, that this uses the whole training set to estimate the gradient w.r.t. the parameters and splitting the training set into mini-batches is not possible at this point, as this would neglect the dependencies the model introduces between the samples.

Although layer-wise sampling of $\widehat{X}^l$ to evaluate (5.19) correctly generates unbiased samples of the derivatives, it has the crucial drawback that for the majority of samples the factor $\mathrm{P}_{\theta^L}(\widehat{X}^L \,|\, \widehat{X}^{L-1})$ is nearly zero, since the sampled values for the inner layers are only conditioned on the inputs $\widehat{X}^0$ and not on the target values $\widehat{X}^L$. Consequently, while this training procedure is sound, convergence to meaningful parameter values would be very slow.

The described problem can be avoided by observing that the joint density of the model can be decomposed into $\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \,|\, \widehat{X}^0) = \mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L)\, \mathrm{P}(\widehat{X}^L \,|\, \widehat{X}^0)$. Using this alternative decomposition of the joint density, the derivative (5.18) can be rewritten as

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \varphi^l} &= \int \cdots \int \mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^L \,|\, \widehat{X}^0)\, \rho'_{\theta^l}(\widehat{X}^l \,|\, \widehat{X}^{l-1})\, \mathrm{d}\widehat{X}^1 \cdots \mathrm{d}\widehat{X}^{L-1} \\
&= \mathrm{P}_\theta(\widehat{X}^L \,|\, \widehat{X}^0) \int \cdots \int \mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L)\, \rho'_{\theta^l}(\widehat{X}^l \,|\, \widehat{X}^{l-1})\, \mathrm{d}\widehat{X}^1 \cdots \mathrm{d}\widehat{X}^{L-1} \\
&= \mathcal{L}(\theta)\, \mathrm{E}_{\mathrm{P}_\theta(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L)} \left[\rho'_{\theta^l}(\widehat{X}^l \,|\, \widehat{X}^{l-1}) \right] . \qquad (5.20)
\end{aligned}$$

As stated before, analytic calculation of the likelihood $\mathcal{L}(\theta)$ is intractable. However, calculating its actual value is unnecessary, since $\mathcal{L}(\theta)$ has the same value, no matter what parameter φ^l the derivative is taken with respect to. Thus, by setting $\mathcal{L}(\theta) \triangleq 1$ while evaluating (5.20) for all parameters θ, we obtain a scaled version of the true gradient $\nabla_\theta \mathcal{L}$. Since $\mathcal{L}(\theta)$ is a probability density, the scaling factor is always non-negative and the gradient points into the right direction. Maximization of $\mathcal{L}$ w.r.t. θ can then be performed using the gradient ascent algorithm. It does not matter that the obtained gradient is scaled, since only the direction is important, cf. section 2.3.1.

Contrary to (5.19) the expectation in (5.20) samples from the distribution $\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L)$, thus avoiding the problem of vanishing probabilities. However, due to the additional dependency on the training targets $\widehat{X}^L$, this distribution *cannot* be sampled from using a Markov chain over the layers. A method to sample from the GPN network, taking into account the observed targets, will be presented in next section.

It remains to calculate the actual derivatives of the log conditional densities $\rho_{\theta^l}(\widehat{X}^l \,|\, \widehat{X}^{l-1})$ using (5.11). They are given by

$$\rho'_{\theta^l} \triangleq \frac{\partial \rho_{\theta^l}}{\partial \varphi^l} = \frac{1}{2} \sum_{n=1}^{N_l} \left[(\widehat{X}^l_{\star n})^T (\widetilde{K}^l_n)^{-1} \frac{\partial \widetilde{K}^l_n}{\partial \varphi} \widetilde{K}^l_n \widehat{X}^l_{\star n} - \mathrm{tr}\left((\widetilde{K}^l_n)^{-1} \frac{\partial \widetilde{K}^l_n}{\partial \varphi} \right) \right] , \qquad (5.21)$$

where the derivatives of the covariance matrix w.r.t. the weights and variances are given by

$$\frac{\partial \widetilde{K}^l_{n,ss'}}{\partial W^l_{n\hat{m}}} = -2\,(\widehat{X}^{l-1}_{s\hat{m}} - \widehat{X}^{l-1}_{s'\hat{m}})\,S^l_{n,ss'}\exp\Big(-(S^l_{n,ss'})^2\Big) \quad \text{with } S^l_{n,ss'} \triangleq \sum_{m=1}^{N_{l-1}} W^l_{nm}\,(\widehat{X}^{l-1}_{sm} - \widehat{X}^{l-1}_{s'm}) \tag{5.22a}$$

$$\frac{\partial \widetilde{K}^l_{n,ss'}}{\partial \sigma^l_n} = 2\,\sigma^l_n\,\delta_{ss'}\,. \tag{5.22b}$$

Note, that we do not optimize the parameters by finding some point estimate of values $\widetilde{X}^1, \ldots, \widetilde{X}^{L-1}$ for the inner layers first and then optimizing the likelihood assuming fixed values for the latent variables. Instead, we use the distribution over the inner GPN layers for calculation of the likelihood, which leads to more robust estimates of the GPN parameters and an inherent resiliency to overfitting due to probabilistic nature of the GPN model. Before a complete training algorithm can be specified, we need to develop a method to sample from $\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L)$ which will be done in the following section.

Sampling from GPN Feed-Forward Networks

Consider the distribution of a GPN feed-forward network as given by (5.16). If only the inputs X^0 are observed, then, as stated before, sampling from the distribution is straightforward. Since the distribution of each layer is defined conditioned on the values of the previous layer by (5.11), we can sequentially sample from the conditional distributions $\mathrm{P}(X^l \,|\, X^{l-1})$ for $l \in \{1, 2, \ldots, L\}$. Each conditional distribution is a multivariate normal distribution, thus efficient samplers are readily available.

However, in the previous section the need arose to sample from a GPN feed-forward network having observed the inputs and the targets, i.e. we need to sample from the distribution

$$\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L) = \frac{1}{Z}\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \,|\, \widehat{X}^0) \tag{5.23}$$

with the partition function $Z = \mathrm{P}(\widehat{X}^L \,|\, \widehat{X}^0)$ being independent of the inner layers $\widehat{X}^l$, $l \in \{1, \ldots, L-1\}$. The sequential sampling procedure is no longer applicable, since no analytically tractable density function exists for the conditional $\mathrm{P}(X^l \,|\, X^{l-1}, X^L)$. Under these circumstances we have to use a more elaborate sampling method. We consider the following, well-known MCMC sampling methods introduced in detail in section 2.2.13.

Gibbs sampling splits the random variables into multiple partitions and alternatively samples from each partition conditioned on the values of the other partitions. It can be shown

that this procedure draws sample from the joint distribution of all random variables if run for enough steps. Observed variables can be introduced be keeping their values fixed at the observed value. In the case of a GPN feed-forward network it is natural to partition the random variables per layer, i.e. each X^l for $l \in \{1, \ldots, L\}$ is a separate partition. Then Gibbs sampling consists of sampling from each conditional $\mathrm{P}(X^l \mid X^{l-1}, X^{l+1})$ in a round-robin sequence over the layers $l \in \{1, 2, \ldots, L\}$. Unfortunately, in the conditional density function $\mathrm{P}(X^l \mid X^{l-1}, X^{l+1}) \propto \mathrm{P}(X^{l-1}, X^l, X^{l+1})$ the random variable X^l occurs inside the covariance matrix of the PDF of the normal distribution, thus introducing highly non-linear functional dependencies for which no efficient sampling method is available.

Metropolis-Hastings moves within the state space of a probabilistic model by drawing candidate moves from a proposal distribution and accepting or rejecting them based on the change of joint probability that would occur if the move was executed. As with Gibbs sampling, observed random variables can be introduced by keeping their values fixed. The algorithm requires two things: a function that is proportional to the density to sample from and a distribution that proposes new potential states given the current state of the random variables. Thus, it makes it possible to work with unnormalized probability densities, allowing us to sample from (5.23) without calculating the partition function Z. However, to work efficiently the Metropolis-Hastings algorithm requires a proposal distribution that is well-tailored to the joint distribution of the model.

Hamiltonian Monte-Carlo sampling describes a method to calculate good proposal distributions for the Metropolis-Hastings algorithm. In addition to a function that is proportional to the model's PDF, it requires the derivatives of this function w.r.t. all random variables. The gradient of the probability density is used to propose states of higher probability compared to the current state and thus the proposed state have a high acceptance probability, making HMC work efficiently even in a high-dimensional state space. Since the derivatives of (5.23) can be calculated in closed form, HMC is a good choice for sampling from a GPN feed-forward network and we will now describe its application.

The HMC potential energy corresponding to the joint distribution $\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \mid \widehat{X}^0)$ given by eqs. (5.11) and (5.16) is

$$\begin{aligned} U(X^1, \ldots, X^L) &= -\log \mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \mid \widehat{X}^0) = -\sum_{l=1}^{L} \log \mathrm{P}(X^l \mid X^{l-1}) \\ &\propto -\sum_{l=1}^{L} \sum_{n=1}^{N_l} \mathcal{LN}(X^l_{\star n} \mid \mathbf{0}, \widetilde{K}^l_n) \end{aligned} \tag{5.24}$$

where $\mathcal{LN}$ is the unnormalized log-density of the multivariate normal distribution given by

$$\mathcal{LN}(\boldsymbol{x} \,|\, \boldsymbol{\mu}, \Sigma) \triangleq -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})\,. \tag{5.25}$$

The HMC algorithm requires the derivatives of the potential energy w.r.t. the layer values X^l_{sn}. For computation of these derivatives, let us split up the potential energy layer-wise, so that we obtain

$$U(X^1, \ldots, X^L) = \sum_{l=1}^{L} U^l(X^l, X^{l-1}) \tag{5.26}$$

with

$$U^l(X^l, X^{l-1}) \triangleq -\sum_{n=1}^{N_l} \mathcal{LN}(X^l_{\star n} \,|\, \mathbf{0}, \widetilde{K}^l_n)\,. \tag{5.27}$$

The layer-wise gradients are straightforward to compute. For each layer l, we obtain

$$\frac{\partial U^l}{\partial X^l_{\star n}} = (\widetilde{K}^l_n)^{-1} X^l_{\star n} \tag{5.28a}$$

$$\frac{\partial U^l}{\partial X^{l-1}_{\hat{s}\hat{m}}} = \frac{1}{2}(X^l_{\star n})^T (\widetilde{K}^l_n)^{-1} \frac{\partial \widetilde{K}^l_n}{\partial X^{l-1}_{\hat{s}\hat{m}}} (\widetilde{K}^l_n)^{-1} X^l_{\star n} \tag{5.28b}$$

where we have defined

$$\frac{\partial \widetilde{K}^l_{n,ss'}}{\partial X^{l-1}_{\hat{s}\hat{m}}} \triangleq \widehat{K}^l_{n\hat{m},ss'} (\delta_{s\hat{s}} - \delta_{s'\hat{s}}) \tag{5.29a}$$

$$\widehat{K}^l_{n\hat{m},ss'} \triangleq -2\, W^l_{n\hat{m}} \exp\left[-\left(\sum_{m=1}^{N_{l-1}} W^l_{nm} (X^{l-1}_{sm} - X^{l-1}_{s'm}) \right)^2 \right] \sum_{m=1}^{N_{l-1}} W^l_{nm} (X^{l-1}_{sm} - X^{l-1}_{s'm})\,. \tag{5.29b}$$

Evaluating the Kronecker deltas in (5.29a) allows to rewrite (5.28b) into

$$\frac{\partial U^l}{\partial X^{l-1}_{\hat{s}\hat{m}}} = -\sum_{n=1}^{N_{l-1}} \sum_{s=1}^{S} \hat{K}^l_{n\hat{m},\hat{s}s}\, z^l_{n,\hat{s}}\, z^l_{n,s} \quad \text{with } \boldsymbol{z}^l_n \triangleq (\widetilde{K}^l_n)^{-1} X^l_{\star n}\,. \tag{5.30}$$

This completes the calculation of the derivatives and we can now apply HMC sampling with the leapfrog method as described in section 2.2.13. If the target values X^L are observed, as in (5.23), they are kept fixed at their observed values during HMC sampling.

In conclusion, we can now evaluate the expectation in (5.20) and optimize the likelihood function $\mathcal{L}(\theta)$ iteratively towards a maximum-likelihood estimate of the model parameters θ. The complete training procedure is described in algorithm 12. For subsequent evaluations of (5.20) it is beneficial to preserve the state of the random variables from evaluation to evalu-

Algorithm 12: Parameter estimation for a GPN feed-forward regression network

Input: training inputs $\widehat{X}^0$ and corresponding targets $\widehat{X}^L$
Output: local parameter optimum θ maximizing $\mathcal{L}(\theta)$
Parameters: learning rate η; number of gradient est. samples R; init. variance σ_{init}

`// random initialization of parameters`
$\forall l, n, \hat{m}$: sample $W^l_{n\hat{m}} \sim \mathcal{N}(0, \sigma^2_{\text{init}})$
$\forall l, n$: sample $\sigma^l_n \sim \mathcal{U}(0, \sigma_{\text{init}})$
while $\mathcal{L}(\theta)$ *increases* **do** `// training loop`
  `// calculate scaled gradient estimate using` R `gradient samples`
  $\forall l, n, \hat{m}$: $\Delta W^l_{n\hat{m}} \longleftarrow 0$
  $\forall l, n$: $\Delta \sigma^l_n \longleftarrow 0$
  for $r \in \{1, 2, \ldots, R\}$ **do**
    Sample $\{\widehat{X}^1, \ldots, \widehat{X}^{L-1}\}$ with HMC using the potential energy (5.24) and its derivatives given by eqs. (5.28a) and (5.30) while keeping $\widehat{X}^L$ fixed to target.
    $\forall l, n, \hat{m}$: $\Delta W^l_{n\hat{m}} \longleftarrow \Delta W^l_{n\hat{m}} + \dfrac{\partial \rho_{\theta^l}}{\partial W^l_{n\hat{m}}}$ `// eqs. (5.21) and (5.22a)`
    $\forall l, n$: $\Delta \sigma^l_n \longleftarrow \Delta \sigma^l_n + \dfrac{\partial \rho_{\theta^l}}{\partial \sigma^l_n}$ `// eqs. (5.21) and (5.22b)`
  `// perform parameter updates using gradient ascent`
  $\omega \longleftarrow |\Delta W|^2 + |\Delta \sigma|^2$ `// use Frobenius norm`
  $\forall l, n, \hat{m}$: $W^l_{n\hat{m}} \longleftarrow W^l_{n\hat{m}} + \dfrac{\eta}{\sqrt{\omega}} \Delta W^l_{n\hat{m}}$
  $\forall l, n$: $\sigma^l_n \longleftarrow \sigma^l_n + \dfrac{\eta}{\sqrt{\omega}} \Delta \sigma^l_n$

ation, since the state from the previous evaluation still represents an area of high probability even after a small change of the parameters and thus the burn-in period of the HMC algorithm can be reduced significantly.

Care must be taken with the learning rate η. In standard gradient ascent the magnitude of the gradient vector goes to zero as a local optimum is approached and thus training proceeds using smaller steps near the optimum. However, since our gradient is scaled by an arbitrary factor, this property does not hold. Thus we have to explicitly monitor the trend of the likelihood $\mathcal{L}(\theta)$ and decrease the learning rate η when the likelihood starts to oscillate near the optimum.

Obtaining Predictions on the Test Set

After fitting the model parameters θ we want to obtain predictions $\widetilde{X}^L$ for previously unseen test inputs $\widetilde{X}^0$ from the GPN feed-forward network. This means sampling from the distribution

$$\mathrm{P}_\theta(\widetilde{X}^L \,|\, \widetilde{X}^0, \widehat{X}^0, \widehat{X}^L) = \frac{\mathrm{P}_\theta(\widehat{X}^0, \widetilde{X}^0, \widehat{X}^L, \widetilde{X}^L)}{\mathrm{P}_\theta(\widehat{X}^0, \widetilde{X}^0, \widehat{X}^L)} \,,$$

which cannot be calculated analytically due to required marginalizations of (5.16) over the inner layer values $\widehat{X}^l$ and $\widetilde{X}^l$, $l \in \{1, \ldots, L-1\}$, that are intractable. The situation is shown in fig. 5.5a for a GPN feed-forward network with three layers. The left side shows the random variables associated with the training set with the input and outputs being observed. The right side shows the random variables associated with the test set with only the input being observed and the outputs to be inferred. Training and test samples share the same Markov random field within each layer. Note that for clarity each node in this figure represents all GPNs and all samples of the respective partition within a layer.

We will now show how predictions can be obtained without having access to an explicit form of the predictive distribution. By expanding (5.11) over the training and test samples explicitly, we get the conditional GPN layer distribution in the form

$$\mathrm{P}(\widehat{X}^l, \widetilde{X}^l \,|\, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) = \prod_{n=1}^{N_l} \mathrm{P}(\widehat{X}^l_{\star n}, \widetilde{X}^l_{\star n} \,|\, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) \tag{5.31}$$

with the GPN layer normal distribution being defined jointly over training and test samples,

$$\mathrm{P}(\widehat{X}^l_{\star n}, \widetilde{X}^l_{\star n} \,|\, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) = \mathcal{N}\left(\begin{bmatrix} \widehat{X}^l_{\star n} \\ \widetilde{X}^l_{\star n} \end{bmatrix} \,\middle|\, \mathbf{0}, \begin{bmatrix} \Sigma^l_{n,11} & \Sigma^l_{n,12} \\ \Sigma^l_{n,21} & \Sigma^l_{n,22} \end{bmatrix} \right) \,, \tag{5.32}$$

with the blocks of the covariance matrix given by

$$\Sigma^l_{n,11,\hat{s}\hat{s}'} \triangleq k(W^l_{n\star}\widehat{X}^{l-1}_{\hat{s}\star}, W^l_{n\star}\widehat{X}^{l-1}_{\hat{s}'\star}) + (\sigma^l_n)^2\delta_{\hat{s}\hat{s}'} \tag{5.33a}$$

$$\Sigma^l_{n,12,\hat{s}\widetilde{s}} \triangleq k(W^l_{n\star}\widehat{X}^{l-1}_{\hat{s}\star}, W^l_{n\star}\widetilde{X}^{l-1}_{\widetilde{s}\star}) \tag{5.33b}$$

$$\Sigma^l_{n,21,\widetilde{s}\hat{s}} \triangleq k(W^l_{n\star}\widetilde{X}^{l-1}_{\widetilde{s}\star}, W^l_{n\star}\widehat{X}^{l-1}_{\hat{s}\star}) \tag{5.33c}$$

$$\Sigma^l_{n,22,\widetilde{s}\widetilde{s}'} \triangleq k(W^l_{n\star}\widetilde{X}^{l-1}_{\widetilde{s}\star}, W^l_{n\star}\widetilde{X}^{l-1}_{\widetilde{s}'\star}) + (\sigma^l_n)^2\delta_{\widetilde{s}\widetilde{s}'} \,. \tag{5.33d}$$

We decompose the GPN distribution (5.32) into

$$\mathrm{P}(\widehat{X}^l_{\star n}, \widetilde{X}^l_{\star n} \,|\, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) = \mathrm{P}(\widehat{X}^l_{\star n} \,|\, \widehat{X}^{l-1}) \, \mathrm{P}(\widetilde{X}^l_{\star n} \,|\, \widehat{X}^l_{\star n}, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) \tag{5.34}$$

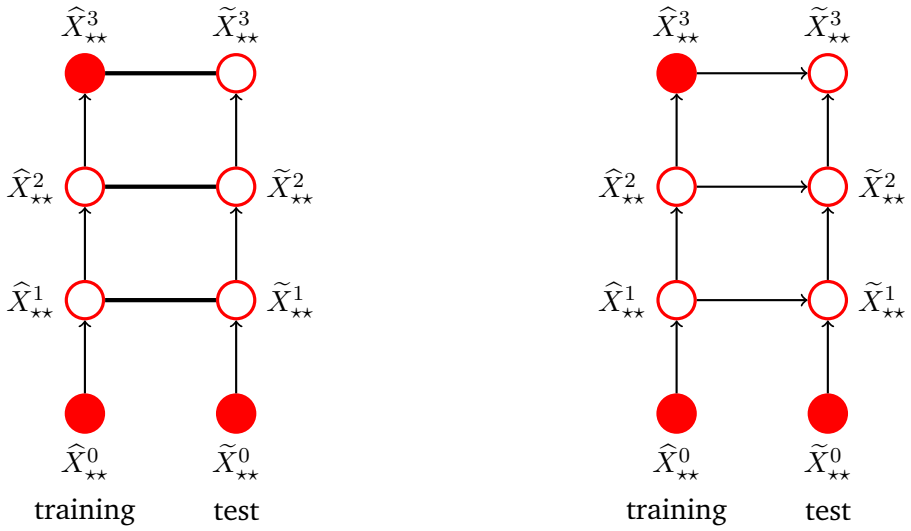


Figure 5.5: Obtaining predictions from a GPN feed-forward network. Observed values are shown as filled nodes. (a) The graphical model for a feed-forward GPN with separate nodes for training and test samples. The layer-wise conditional probabilities are specified jointly over the training and test set. (b) The undirected connections between training and test samples have been replaced by directed connections by using a different factorization of the joint probability distribution of the model shown in (a). Thus (a) and (b) are representing the same probability distribution. Since all nodes in (b) representing training samples are d-separated from the observed test inputs $\widetilde{X}^0$, the test part (on the right side) can be ignored while sampling the training part $\widehat{X}^l$, $l \in \{1, \ldots, L-1\}$ (on the left side) and then the test part can be inferred conditioned on sampled values of the training part.

and using the marginalization and conditioning properties of the multivariate normal distribution, cf. section 2.2.9, we obtain the following factors,

$$\mathrm{P}(\widehat{X}^l_{\star n} \,|\, \widehat{X}^{l-1}) = \mathcal{N}(\widehat{X}^l_{\star n} \,|\, \mathbf{0}, \Sigma^l_{n,11}) \,, \tag{5.35a}$$

$$\mathrm{P}(\widetilde{X}^l_{\star n} \,|\, \widehat{X}^l_{\star n}, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) = \mathcal{N}(\widetilde{X}^l_{\star n} \,|\, \Sigma^l_{n,21}(\Sigma^l_{n,11})^{-1}\widehat{X}^l_{\star n},\, \Sigma^l_{n,22} - \Sigma^l_{n,21}(\Sigma^l_{n,11})^{-1}\Sigma^l_{n,12}) \,. \tag{5.35b}$$

This allows us to rewrite the joint distribution of training and test samples of the GPN feed-forward network (5.16) as

$$\mathrm{P}(\widehat{X}^1, \widetilde{X}^1, \ldots, \widehat{X}^L, \widetilde{X}^L \,|\, \widehat{X}^0, \widetilde{X}^0) = \mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \,|\, \widehat{X}^0)\, \mathrm{P}(\widetilde{X}^1, \ldots, \widetilde{X}^L \,|\, \widetilde{X}^0, \widehat{X}^0, \ldots, \widehat{X}^L) \tag{5.36}$$

Algorithm 13: Obtaining predictions from a GPN feed-forward regression network

Input: test inputs $\widetilde{X}^0$; training inputs $\widehat{X}^0$ and corresponding targets $\widehat{X}^L$; fitted model parameters θ
Output: test predictions $\widetilde{X}^L$

1 Sample $\{\widehat{X}^1, \ldots, \widehat{X}^{L-1}\}$ with HMC using the potential energy (5.24) and its derivatives given by eqs. (5.28a) and (5.30) while keeping $\widehat{X}^L$ fixed to targets.

2 **for** $l \in \{1, 2, \ldots, L\}$ **do**

3 Sample $\widetilde{X}^l$ from $\mathrm{P}(\widetilde{X}^l \,|\, \widehat{X}^l, \widehat{X}^{l-1}, \widetilde{X}^{l-1})$ given by eq. (5.35b).

with the factors

$$\mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^L \,|\, \widehat{X}^0) = \prod_{l=1}^{L} \prod_{n=1}^{N_l} \mathrm{P}(\widehat{X}^l_{\star n} \,|\, \widehat{X}^{l-1}) \tag{5.37a}$$

$$\mathrm{P}(\widetilde{X}^1, \ldots, \widetilde{X}^L \,|\, \widetilde{X}^0, \widehat{X}^0, \ldots, \widehat{X}^L) = \prod_{l=1}^{L} \prod_{n=1}^{N_l} \mathrm{P}(\widetilde{X}^l_{\star n} \,|\, \widehat{X}^l_{\star n}, \widehat{X}^{l-1}, \widetilde{X}^{l-1}) \tag{5.37b}$$

and thus we obtain the probabilistic graphical model shown in fig. 5.5b. Both fig. 5.5a and fig. 5.5b represent the same joint distribution but with different expansions of the distribution. In fig. 5.5b the alternative decomposition (5.34) replaced the undirected connection between training and test samples in each layer with a directed connection from training to test samples.

From (5.36) it follows that the GPN feed-forward network distribution with the training targets $\widehat{X}^L$ observed is given by

$$\begin{aligned}\mathrm{P}(\widehat{X}^1, \widetilde{X}^1, \ldots, \widehat{X}^{L-1}, \widetilde{X}^{L-1}, \widetilde{X}^L \,|\, \widehat{X}^0, \widetilde{X}^0, \widehat{X}^L) = \mathrm{P}(\widehat{X}^1, \ldots, \widehat{X}^{L-1} \,|\, \widehat{X}^0, \widehat{X}^L) \cdot \\ \mathrm{P}(\widetilde{X}^1, \ldots, \widetilde{X}^L \,|\, \widetilde{X}^0, \widehat{X}^0, \ldots, \widehat{X}^L)\,.\end{aligned} \tag{5.38}$$

From fig. 5.5b we see that all nodes representing training samples are d-separated (section 2.2.10) from the observed test input $\widetilde{X}^0$ and thus they are unaffected by its observation. Hence, drawing samples from the distribution (5.38) is straightforward as described in algorithm 13. Thus, finally we can infer model predictions $\widetilde{X}^L$ given the test inputs $\widetilde{X}^0$. To calculate confidence intervals of the predictions, multiple predictive samples must be obtained and the mean and variance evaluated numerically.

5.2.3 Complexity of Non-Parametric Training and Inference

The main cost of training a GPN network is determined by HMC sampling (in line 7 of algorithm 12) to get values for the inner layers $\widehat{X}^1, \ldots, \widehat{X}^L$ given training inputs and targets. Each step of HMC sampling computes derivatives w.r.t. all GPNs in the network and thus has cost similar to one round of back-propagation in a conventional neural network. Consequently the cost factor of training a GPN network compared to the cost of training a conventional neural network is determined by how many HMC steps are necessary between drawing gradient samples. Provided that the learning rate η is reasonably small, it can be assumed that the change of the parameters is small enough, so that the final state of the Markov chain from the previous training step is a good initial state for the Markov chain of the next training step and hence no burn-in period is required. Thus the number of required steps is solely driven by the need to avoid autocorrelation between samples of the gradient.

Contrary to a conventional neural network with a fixed activation function, the training set is also required to obtain predictions from a GPN network, even after the parameters θ have been fitted. This is due to the correlation between samples inherent to the model. Also, training can only be performed in batches. Mini-batch training, in which the gradient is estimated on alternating parts of the training set and used for intermediate updates of the parameters, is not possible. Thus, in its current form, a GPN network cannot scale to datasets that are beyond the size of available memory.

While the flexibility of a data-adaptable activation function for each GPN increases the expressive power of the model compared to a conventional neural network, the drawbacks listed above make the model impractical and inefficient to use in its current form. Especially the loss of the fully parametric nature of a neural network and the associated benefit of being able to scale to datasets of arbitrary size, limits the applicability of the model. Thus, we will introduce an auxiliary parametric approximation of the GPN model that avoids the drawbacks listed above and retains the advantages of a fully parametric model, while allowing the activation functions to adapt to the training data. We will later show how the non-parametric GPN can be recovered from the auxiliary parametric model by use of an appropriate prior. This will allow us to perform Bayesian inference using an approximative variational posterior without the need for MCMC sampling methods and their associated costs.

5.3 The Parametric Gaussian Process Neuron

Let us revisit the definition of a GPN given in section 5.1. According to (5.3) the responses are given by a GP conditioned on the activations and thus correlations between samples are introduced, as can be seen from (5.5). As demonstrated in the previous sections, these correlations force us to perform HMC sampling for training and inference, and require the preservation of the training set for inference. While it is desirable to break up the inter-sample dependencies, the activation function still has to be consistent across all samples.

A possibility to convey consistent information across all samples is to introduce additional random variables that are sampled independent and condition the response of each sample on them. In the context of GPs this method was first used to find a sparse approximation of the training data and is described in (Quiñonero-Candela and Rasmussen, 2005). Three random vectors $\boldsymbol{V} \in \mathbb{R}^R$, $\boldsymbol{U} \in \mathbb{R}^R$ and $\boldsymbol{S} \in (\mathbb{R}^+)^R$ are introduced per GPN. Their purpose is to explicitly parameterize R points of the activation function representing the mode of the GP distribution. For each virtual observation point $r \in \{1, \ldots, R\}$ this parameterization consists of an inducing point V_r, corresponding to the activation of the observation, the target U_r, corresponding to the response given that activation, and the variance S_r around the response. Thus we assume that we are making observations of the activation function $f(a)$. These observations are of the form

$$f(V_r) = U_r + \epsilon \quad \text{with } \epsilon \sim \mathcal{N}(0, S_r)\,, \quad r \in \{1, \ldots, R\}\,. \tag{5.39}$$

We introduce these observations by replacing the GP prior $\mathrm{P}(F \mid A)$, i.e. eqs. (5.3) and (5.5), with the conditional distribution

$$\mathrm{P}(F_\star \mid A_\star, V_\star, U_\star, S_\star) = \mathcal{N}(F_\star \mid \mu^F_\star, \Sigma^F_{\star\star}) \tag{5.40}$$

where the mean and covariance are those obtained by using the virtual observations $V_\star$ and $U_\star$ as "training" points for a GP regression evaluated at the "test" points $A_\star$. By calculating the conditional of the multivariate normal distribution (5.5), cf. section 2.2.9, we obtain

$$\mu^F_\star = K(A_\star, V_\star)\,[K(V_\star, V_\star) + \mathrm{diag}(S_\star)]^{-1}\,U_\star \tag{5.41a}$$

$$\Sigma^F_{\star\star} = K(A_\star, A_\star) - K(A_\star, V_\star)\,[K(V_\star, V_\star) + \mathrm{diag}(S_\star)]^{-1}\,K(V_\star, A_\star)\,. \tag{5.41b}$$

where the covariance matrices are defined by

$$[K(V_\star, V_\star)]_{rr'} \triangleq k(V_r, V_{r'})\,, \tag{5.42a}$$
$$[K(A_\star, A_\star)]_{ss'} \triangleq k(A_s, A_{s'})\,, \tag{5.42b}$$
$$[K(A_\star, V_\star)]_{sr} = [K(V_\star, A_\star)]_{rs} \triangleq k(A_s, V_r)\,. \tag{5.42c}$$

using the covariance function $k(a, a')$ given by (5.4).

5.3.1 Marginal Distribution and Layers

Leaving the rest of the GPN model unchanged we can perform marginalization over A and F to obtain a collapsed distribution for the outputs given the inputs $\mathrm{P}(X \mid Y)$ of the parametric model. By following the same procedure as in section 5.1.1 we obtain

$$\mathrm{P}(X_\star \mid Y_{\star\star}, V_\star, U_\star, S_\star) = \mathcal{N}(X_\star \mid \mu^X, \Sigma^X) \tag{5.43}$$

with

$$\mu^X = K(Y_{\star\star}\boldsymbol{w}, V_\star)\,[K(V_\star, V_\star) + \mathrm{diag}(S_\star)]^{-1}\,U_\star \tag{5.44}$$
$$\Sigma^X = K(Y_{\star\star}\boldsymbol{w}, Y_{\star\star}\boldsymbol{w}) - K(Y_{\star\star}\boldsymbol{w}, V_\star)\,[K(V_\star, V_\star) + \mathrm{diag}(S_\star)]^{-1} K(V_\star, Y_{\star\star}\boldsymbol{w}) + \sigma^2\,, \tag{5.45}$$

where $\boldsymbol{w}$ is the weight vector and σ^2 is the output variance of the GPN as before. Since the inducing points V, targets U and variances S will always assumed to be observed in the following sections, we stop listing them explicitly from here on to maintain the clarity of notation. We call this model a *parametric Gaussian process neuron*.

As before when stacking layers, we set $Y^l = X^{l-1}$ for $l \geq 1$. Thus, for layer l of GPNs as described in section 5.1.2 the parametric GPN distribution becomes

$$\mathrm{P}(X^l_{\star n} \mid X^{l-1}_{\star\star}) = \mathcal{N}(X^l_{\star n} \mid \mu^{X^l}_{\star n}, \Sigma^{X^l}_{\star\star n}) \tag{5.46}$$

with

$$\mu^{X^l}_{\star n} = K(X^{l-1}_{\star\star}W^l_{n\star}, V^l_{\star n})\,[K(V^l_{\star n}, V^l_{\star n}) + \mathrm{diag}(S^l_{\star n})]^{-1}\,U^l_{\star n} \tag{5.47a}$$
$$\Sigma^{X^l}_{\star\star n} = K(X^{l-1}_{\star\star}W^l_{n\star}, X^{l-1}_{\star\star}W^l_{n\star}) - K(X^{l-1}_{\star\star}W^l_{n\star}, V^l_{\star n})\,[K(V^l_{\star n}, V^l_{\star n}) + \mathrm{diag}(S^l_{\star n})]^{-1} K(V^l_{\star n}, X^{l-1}_{\star\star}W^l_{n\star}) + (\sigma^l_n)^2\,\mathbb{1}\,. \tag{5.47b}$$

where V^l_{rn}, U^l_{rn} and S^l_{rn} respectively denotes the r-th virtual observation point, target and

symbol	*purpose*
model hyper-parameters:	
N_l	number of GPNs in layer l
random variables:	
Y^l_{sm}	input dimension m of sample s to GPN layer l
A^l_{sn}	activation of GPN n in layer l corresponding to sample s
F^l_{sn}	response of GPN n in layer l corresponding to sample s
X^l_{sn}	output of GPN n in layer l corresponding to sample s
model parameters:	
W^l_{nm}	weight from input m to GPN n in layer l
$(\sigma^l_n)^2$	output variance of GPN n in layer l
additional parameters for parametric GPN:	
V^l_{rn}	virtual observation point r of GPN n in layer l
U^l_{rn}	virtual observation target r of GPN n in layer l
S^l_{rn}	virtual observation variance r of GPN n in layer l

Table 5.1: Overview of the notation used for GPNs. When GPN layers are stacked a superscripted l is used to denote the layer index and the output of one layer is the input to the next, thus $Y^l = X^{l-1}$.

variance of GPN n in layer l. Thus we obtained a *parametric GPN layer*.

An overview of the notation used is provided in table 5.1. As before we will use superscripts to denote the layer index when stacking GPN layers to build feed-forward networks. The parameters of a layer now include the virtual observations besides the weights, thus $\theta^l \triangleq \{W^l, \boldsymbol{\sigma}^l, V^l, U^l, S^l\}$ are the parameters of each layer that need to be estimated during training.

5.3.2 Drawing Samples

Let us set aside the question on how to efficiently estimate the parameters of the model and examine how sampling is performed in a parametric GPN feed-forward network beforehand. Thus, for now we assume that the values for the parameters θ have been learned in some way and we want to obtain predictions from the model. Since the GPN layers form a Markov chain, sampling is straightforward. Given the inputs X^0, the mean vectors and covariance matrices of all GPNs in the first layer are calculated using (5.47) and a sample is drawn from the resulting multivariate normal distribution for X^1. This process is repeated for X^l, $l \in \{2, \ldots, L\}$, until reaching the top layer. The resulting value for X^L is an unbiased sample from the distribution $\mathrm{P}(X^L \mid X^0)$.

Alternatively, we can understand this procedure as drawing samples in the space of acti-

vation functions. Revisiting the response distribution (5.40), we note that it gives raise to a distribution over functions $f(a)$, which can be specified in terms of a GP as

$$f(a) \,|\, V^l_{\star n}, U^l_{\star n}, S^l_{\star n} \sim \mathcal{GP}(m_p(a), k_p(a, a')) \tag{5.48}$$

with mean and covariance functions

$$m_p(a) = K(a, V^l_{\star n}) \left[K(V^l_{\star n}, V^l_{\star n}) + \operatorname{diag}(S^l_{\star n})\right]^{-1} U^l_{\star n} \,, \tag{5.49a}$$

$$k_p(a, a') = k(a, a') - K(a, V^l_{\star n}) \left[K(V^l_{\star n}, V^l_{\star n}) + \operatorname{diag}(S^l_{\star n})\right]^{-1} K(V^l_{\star n}, a') \,. \tag{5.49b}$$

As a consequence the sampling procedure described above can be interpreted as follows. First, for each GPN in the feed-forward network a *function* $f(a)$ is drawn from the GP (5.48). This is depicted for four GPNs in fig. 5.6. Then, each GPN is treated as if it was a conventional neuron using the sampled function as its activation function. Thus the input samples are propagated through the network as if it was a conventional feed-forward neural network. To obtain a new batch of samples a new set of activation functions is drawn and the procedure repeats.

5.3.3 Loss Functions and Training Objectives

The standard method for training a *deterministic* regression or classification model is to minimize the so-called loss on the training set. The loss is defined as a function of the model parameters θ,

$$\mathscr{L}(\theta) = \frac{1}{S} \sum_{s=1}^{S} L(\boldsymbol{y}_\theta(X_{s\star}), T_{s\star}) \,, \tag{5.50}$$

where $X_{s\star}$ is a training sample with corresponding target $T_{s\star}$ and $\boldsymbol{y}_\theta(\boldsymbol{x})$ is the prediction of the model given input $\boldsymbol{x}$. The loss measure $L : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ assigns a loss value to each sample based on the difference between the model's prediction and the ground truth. Many loss measures are imaginable. For classification tasks the cross entropy loss as given by (2.95) is a common choice for the loss measure. It uses the softmax function (2.92) to transform the outputs of the model into class probabilities of a categorical distribution and calculates the log probability of the targets under this distribution.

Since a GPN feed-forward network provides a predictive distribution $\mathrm{P}(X^L \,|\, X^0)$ instead of a point estimate, training is performed by minimizing the expectation of the loss, i.e. the objective is to minimize

$$\mathscr{L}(\theta) = \mathrm{E}_{\mathrm{P}(X^L \,|\, X^0)} \left[\frac{1}{S} \sum_{s=1}^{S} L(X^L_{s\star}, T_{s\star}) \right] = \frac{1}{S} \sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X^L_{s\star} \,|\, X^0)} \left[L(X^L_{s\star}, T_{s\star}) \right] \,. \tag{5.51}$$

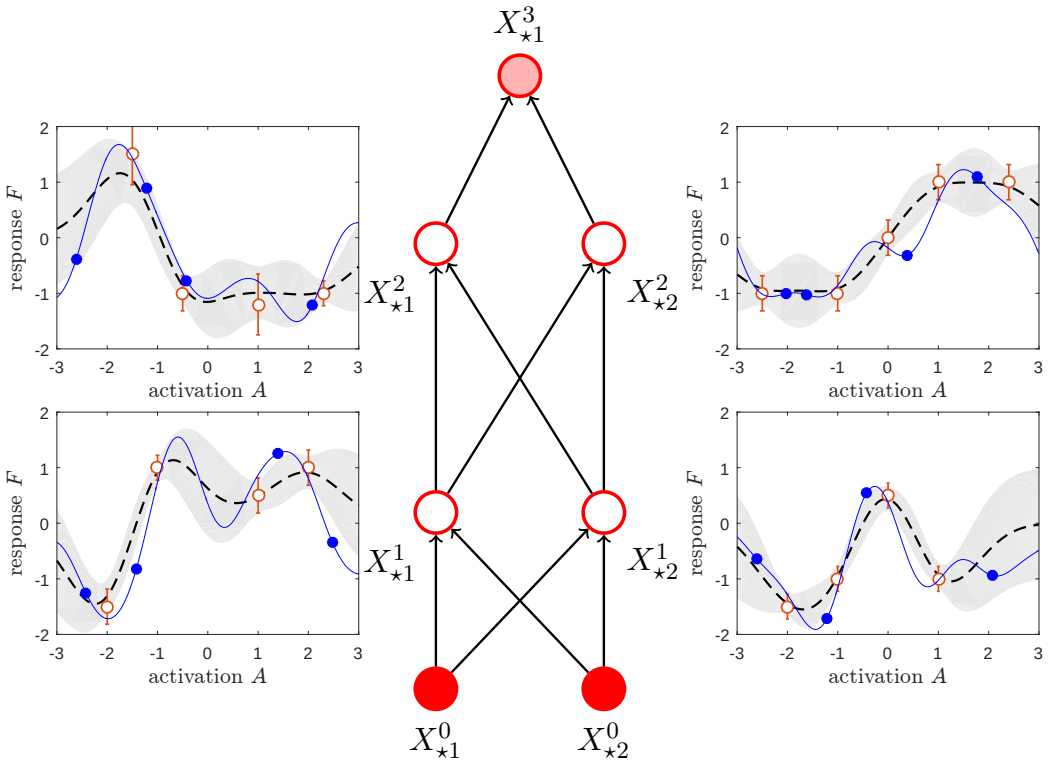


Figure 5.6: Obtaining samples from a feed-forward parametric GPN network. The inducing points (V, U) of each GPN are shown as red circles with their variances S depicted by error bars. The resulting mean function is denoted by the dashed black line and the standard deviation is represented by the gray shaded area. For each GPN an activation function has been drawn from this distribution over functions and is shown as the blue line. The filled blue circles represent data samples that are propagated through the network using the sampled activation functions.

Due to the marginalization property of the multivariate normal distribution and because $\mu^{X^l}_{s\star}$ and $\Sigma^{X^l}_{ss\star}$ depend only on X^{l-1} via $X^{l-1}_{s\star}$ as can be seen from (5.47), we observe that $\mathrm{P}(X^l_{s\star} \mid X^{l-1}_{\star\star}) = \mathrm{P}(X^l_{s\star} \mid X^{l-1}_{s\star})$ for all l, and thus the objective becomes

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X^L_{s\star} \mid X^0_{s\star})}\left[L(X^L_{s\star}, T_{s\star})\right] . \tag{5.52}$$

Thus for a classification task we have the objective

$$\mathscr{L}_{\text{class}}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X^L_{s\star} \mid X^0_{s\star})}\left[T_{s\star} \cdot \log \operatorname{softmax}(X^L_{s\star})\right] \tag{5.53}$$

where $T_{s\star}$ uses a one-hot encoding for the target classes and $\cdot$ denotes the scalar product

between two vectors.

For a regression task, we use the negative delta distribution (5.2) as the loss measure

$$L_{\text{reg}}(\boldsymbol{y}, \boldsymbol{t}) = -\delta(\boldsymbol{y} - \boldsymbol{t})\,. \tag{5.54}$$

Thus the loss is "minus infinity" if the prediction is exactly correct, otherwise it is zero. This seems unreasonable at first since this measure is not differentiable, but analytically evaluating parts of the expectation will result in a plausible loss function as we will show now. Inserting this loss into (5.52) and expanding the expectation over the last layer leads to

$$\begin{aligned}
\mathscr{L}_{\text{reg}}(\theta) &= -\frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X_{s\star}^{L} \,|\, X_{s\star}^{0})}\left[\delta(X_{s\star}^{L} - T_{s\star})\right] \\
&= -\frac{1}{S}\sum_{s=1}^{S} \iint \mathrm{P}(X_{s\star}^{L-1} \,|\, X_{s\star}^{0})\, \mathrm{P}(X_{s\star}^{L} \,|\, X_{s\star}^{L-1})\, \delta(X_{s\star}^{L} - T_{s\star})\, \mathrm{d}X_{s\star}^{L-1}\, \mathrm{d}X_{s\star}^{L} \\
&= -\frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X_{s\star}^{L-1} \,|\, X_{s\star}^{0})}\left[\mathrm{P}(X_{s\star}^{L} = T_{s\star} \,|\, X_{s\star}^{L-1})\right]\,.
\end{aligned} \tag{5.55}$$

We can further take the logarithm and apply Jensen's inequality to obtain the standard negative log-likelihood objective function,

$$\mathscr{L}_{\text{ll}}(\theta) = -\frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(X_{s\star}^{L-1} \,|\, X_{s\star}^{0})}\left[\log \mathrm{P}(X_{s\star}^{L} = T_{s\star} \,|\, X_{s\star}^{L-1})\right] \geq -\log(-\mathscr{L}_{\text{reg}}(\theta))\,, \tag{5.56}$$

which is an upper bound of the logarithmic regression loss. Note that by minimizing $\mathscr{L}_{\text{ll}}(\theta)$ the predicted variance of the outputs X^{L-1} is taken into account. A prediction that is far off from the ground truth will be penalized stronger if the GPN network simultaneously predicts a low variance, i.e. high confidence, at the same time. Consequently during training the model not only learns to predict the targets but also to self-estimate the confidence of its predictions.

5.3.4 Reparameterization and Stochastic Training

Analytically maximizing the objective function (5.52) would involve marginalization over the states of the intermediate layers $l \in \{1, \ldots, L-1\}$. Unfortunately this is intractable, because each layer marginal $P(X_{s\star}^{l} \,|\, X_{s\star}^{0})$ for $l \geq 2$ is a distribution of arbitrary form since the values of the previous layer appear non-linearly through the covariance function in its conditional mean and covariance. Instead we will approximate the expectations in eqs. (5.52) and (5.56) by sampling from $\mathrm{P}(X_{s\star}^{L} \,|\, X_{s\star}^{0})$ or $\mathrm{P}(X_{s\star}^{L-1} \,|\, X_{s\star}^{0})$ respectively.

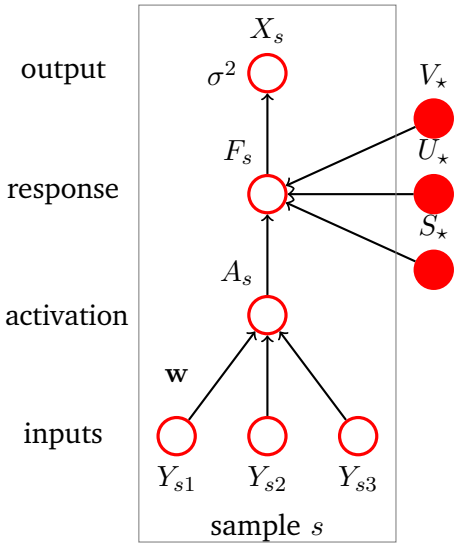


Figure 5.7: Parametric approximation of a GPN with factorization over samples for training. The box indicates that the inside structure is replicated for each sample s. The response of every sample depends on a set of inducing points $\boldsymbol{V}$, targets $\boldsymbol{U}$ and variances $\boldsymbol{S}$ that is shared between *all* samples. This parameterization allows for efficient training and inference.

The structure of this expectation leads to all samples being independent given the model parameters. Consequently, each sample can be propagated independently through the network. For this purpose the layer conditionals further simplify into *univariate* normal distributions for each sample s and GPN n in layer l,

$$\mathrm{P}(X^l_{sn} \,|\, X^{l-1}_{s\star}) = \mathcal{N}(X^l_{sn} \,|\, \mu^{X^l}_{sn}, \Sigma^{X^l}_{ssn}) \tag{5.57}$$

with $\mu^{X^l}_{sn}$ and $\Sigma^{X^l}_{ssn}$ given by (5.47). Note that this significantly reduces the computational complexity, since only the diagonals of the matrices $\Sigma^{X^l}_{\star\star n}$ are required from now on. This factorization corresponds to the probabilistic graphical model shown in fig. 5.7. The Markov random field between the samples in the responses of the full GPN model (fig. 5.1) has been replaced by directed connections from the inducing points to each sample, with the virtual observations $\{V^l_{rn}, U^l_{rn}, S^l_{rn}\}$ being shared between all samples.

For optimization we need the derivative of $\mathscr{L}(\theta)$ w.r.t. θ, but it cannot be calculated directly because the parameters θ appear in the distribution $\mathrm{P}(X^L_{s\star} \,|\, X^0_{s\star})$ the expectation is taken over. To solve this issue we reparameterize the model using the reparameterization trick (D. P. Kingma et al., 2013) as follows. For each random variable X^l_{sn} representing a GPN we introduce

an auxiliary random variable χ^l_{sn} with standard normal distribution,

$$\chi^l_{sn} \sim \mathcal{N}(0,1)\,, \quad l \in \{1,\ldots,L\},\, s \in \{1,\ldots,S\},\, n \in \{1,\ldots,N_l\}\,, \tag{5.58}$$

and replace each random variable X^l_{sn} by

$$X^l_{sn}(X^{l-1}_{s\star}, \chi^l_{sn}) \triangleq \mu^{X^l}_{sn} + \Sigma^{X^l}_{ssn}\chi^l_{sn} \tag{5.59}$$

where the dependency on $X^{l-1}_{s\star}$ is mediated through $\mu^{X^l}_{sn}$ and $\Sigma^{X^l}_{ssn}$. This is equivalent to defining the probability

$$\mathrm{P}(X^l_{sn} \,|\, X^{l-1}_{s\star}, \chi^l_{sn}) = \delta(X^l_{sn} - \mu^{X^l}_{sn} - \Sigma^{X^l}_{ssn}\chi^l_{sn}) \tag{5.60}$$

using the delta distribution. The affine transformation property of the normal distribution (2.36) enables us to immediately verify that the distribution of

$$\mathrm{P}_{\theta^l}(X^l_{sn} \,|\, X^{l-1}_{s\star}) = \int_{-\infty}^{\infty} \mathrm{P}(X^l_{sn} \,|\, X^{l-1}_{s\star}, \chi^l_{sn})\, \mathrm{P}(\chi^l_{sn})\, \mathrm{d}\chi^l_{sn} \tag{5.61}$$

is unchanged, i.e. we recover the original $\mathrm{P}(X^l_{sn} \,|\, X^{l-1}_{s\star})$ given by (5.57). We insert (5.61) into (5.52) and obtain the loss of the reparameterized model,

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \int \cdots \int \left(\prod_{l=1}^{L} \mathrm{P}_{\theta^l}(\widehat{X}^l_{s\star} \,|\, \widehat{X}^{l-1}_{s\star}, \chi^l_{s\star})\, \mathrm{P}(\chi^l_{s\star}) \right) L(X^L_{s\star}, T_{s\star}) \cdot \\ \mathrm{d}\widehat{X}^1_{s\star}\, \mathrm{d}\chi^1_{s\star} \cdots \mathrm{d}\widehat{X}^L_{s\star}\, \mathrm{d}\chi^L_{s\star}\,, \tag{5.62}$$

where we have defined

$$\mathrm{P}(\chi^l_{s\star}) \triangleq \prod_{n=1}^{N_l} \mathrm{P}(\chi^l_{sn})\,, \qquad \mathrm{d}\chi^l_{s\star} \triangleq \mathrm{d}\chi^l_{s1}\, \mathrm{d}\chi^l_{s2} \cdots \mathrm{d}\chi^l_{sN_l}\,.$$

The integrals over $\widehat{X}^1_{s\star}, \ldots, \widehat{X}^L_{s\star}$ can now be evaluated, leading to

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \int \cdots \int L\Big(X^L_{s\star}\Big(X^{L-1}_{s\star}\big(\cdots X^1_{s\star}(X^0_{s\star}, \chi^1_{s\star}) \cdots, \chi^{L-1}_{s\star}\big), \chi^L_{s\star}\Big), T_{s\star}\Big) \cdot \\ \mathrm{P}(\chi^1_{s\star})\mathrm{d}\chi^1_{s\star} \cdots \mathrm{P}(\chi^L_{s\star})\mathrm{d}\chi^L_{s\star}\,.$$

Thus the reparameterized objective function is

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(\chi^1_{s\star},\ldots,\chi^L_{s\star})}[\widehat{L}_\chi(\theta, X^0_{s\star}, T_{s\star})]\,, \tag{5.63}$$

with the reparameterized loss measure given by

$$\widehat{L}_\chi(\theta, X^0_{s\star}, T_{s\star}) \triangleq L\Big(X^L_{s\star}\Big(X^{L-1}_{s\star}\big(\cdots X^1_{s\star}(X^0_{s\star}, \chi^1_{s\star})\cdots, \chi^{L-1}_{s\star}\big), \chi^L_{s\star}\Big), T_{s\star}\Big)\,. \tag{5.64}$$

Note that this is a fully deterministic function given χ.

By performing the same reparameterization for the regression log-likelihood (5.56) we obtain

$$\mathscr{L}_{\mathrm{ll}}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(\chi^1_{s\star},\ldots,\chi^{L-1}_{s\star})}[\widehat{L}_{\mathrm{ll},\chi}(\theta, X^0_{s\star}, T_{s\star})] \tag{5.65}$$

with the reparameterized regression loss measure

$$\begin{aligned}\widehat{L}_{\mathrm{ll},\chi}(\theta, X^0_{s\star}, T_{s\star}) \triangleq -\log \mathrm{P}_\theta\Big(X^L_{s\star} = T_{s\star}\,\Big| \\ X^{L-1}_{s\star} = X^{L-1}_{s\star}\Big(X^{L-2}_{s\star}\big(\cdots X^1_{s\star}(X^0_{s\star}, \chi^1_{s\star})\cdots, \chi^{L-2}_{s\star}\big), \chi^{L-1}_{s\star}\Big)\Big)\,.\end{aligned} \tag{5.66}$$

Note that in both cases only the expectant depends on the model parameters θ and the distribution the expectation is taken over is independent of them. Since the expectation is a linear operator, we can now calculate the gradients of the expected loss w.r.t. a parameter $\varphi \in \theta$. It is given by

$$\frac{\partial \mathscr{L}}{\partial \varphi} = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\mathrm{P}(\chi^\star_{s\star})}\left[\frac{\partial \widehat{L}_\chi}{\partial \varphi}(\theta, X^0_{s\star}, T_{s\star})\right]\,. \tag{5.67}$$

Since $\widehat{\mathcal{L}}_\chi(\theta, X^0_{s\star}, X^L_{s\star})$ and $\widehat{\mathcal{L}}_{\mathrm{ll},\chi}(\theta, X^0_{s\star}, X^L_{s\star})$ are deterministic functions provided that $\chi^\star_{s\star}$ is given, all their derivatives can be calculated by iterated applications of the chain rule. Furthermore, $\mathrm{P}(\chi^\star_{s\star})$ is the product of standard, univariate normal distributions and thus sampling values for all χ^l_{sn} is trivial and can be performed fully parallel.

A training algorithm for parametric GPN feed-forward networks using mini-batches is shown in algorithm 14. The only difference to training a conventional neural network is in line 9 where the values for χ are sampled. During the forward and back propagation passes in lines 10 and 12 χ is treated as a constant and the loss and gradient calculations are done using the standard forward- and back-propagation algorithms. The formulas for the derivatives are not given here; instead we rely on automatic differentiation in our implementation to calculate them. Further

Algorithm 14: Training a parametric GPN feed-forward network using sampling

Input: training inputs $\widehat{X}^0$ and corresponding targets $\widehat{T}^L$
Output: local parameter optimum θ
Parameters: learning rate η_p; loss update rate η_l; mini-batch size S_b; initial variance σ_{init}; initial ranges D_V, D_U

`// random initialization of parameters`
$\forall l, n, m$: sample $W^l_{nm} \sim \mathcal{N}(0, \sigma^2_{\text{init}})$
$\forall l, n$: sample $\sigma^l_n \sim \mathcal{U}(0, \sigma_{\text{init}})$
$\forall l, r, n$: sample $V^l_{rn} \sim \mathcal{U}(-D_V, D_V)$
$\forall l, r, n$: sample $U^l_{rn} \sim \mathcal{U}(-D_U, D_U)$
$\forall l, r, n$: sample $S^l_{rn} \sim \mathcal{U}(0, \sigma_{\text{init}})$

`// training loop using mini-batches`
$\mathscr{L}_{\text{est}} \longleftarrow 0$
while $\mathscr{L}_{\text{est}}$ *decreases* **do**
  Draw S_b samples from the training set and call the inputs $\overline{X}^0_{s\star}$ and targets $\overline{T}_{s\star}$ with $s \in \{1, \ldots, S_b\}$.
  $\forall l, s, n$: sample $\chi^l_{sn} \sim \mathcal{N}(0, 1)$
  $\mathscr{L}_{\text{est}} \longleftarrow (1 - \eta_l)\mathscr{L}_{\text{est}} + \frac{\eta_l}{S_b} \sum_{s=1}^{S} \widehat{L}_\chi(\theta, \overline{X}^0_{s\star}, \overline{T}_{s\star})$ `// update loss estimate`
  for $\varphi \in \{W^l_{nm}, \sigma^l_n, V^l_{rn}, U^l_{rn}, S^l_{rn}\}$ **do**
    $\Delta\varphi \longleftarrow \frac{1}{S_b} \sum_{s=1}^{S_b} \frac{\partial \widehat{L}_\chi}{\partial \varphi}(\theta, \overline{X}^0_{s\star}, \overline{T}_{s\star})$ `// use backpropagation for deriv.`
  for $\varphi \in \{W^l_{nm}, \sigma^l_n, V^l_{rn}, U^l_{rn}, S^l_{rn}\}$ **do**
    $\varphi \longleftarrow \varphi - \eta_p \Delta\varphi$ `// perform parameter updates`

details about how this is performed are given in chapter 4.

5.3.5 Discussion

We have presented an auxiliary parametric approximation to the GPN model, which provides several benefits compared to the original model. Since the model is fully parametric, after training all information about the training samples is stored in the parameters of the model and hence it is not necessary to keep training samples for prediction. The number of parameters depends on the number of GPNs and how many virtual observations are used per GPN, but it is independent of the number of training samples. Furthermore training can be performed in mini-batches; thus portions of the training set can be presented sequentially to the model during training. These properties taken together allow parametric GPN networks to scale to datasets of

arbitrary size, just like conventional neural networks do. The parametric approximation makes no assumptions on the marginal distributions $\mathrm{P}(X^l_{s\star} \mid X^0_{s\star})$, $l \in \{1, \ldots, L\}$, and thus, although all conditional distributions are multivariate normals, a parametric GPN feed-forward network can learn to represent arbitrary data distributions $\mathrm{P}(X^L_{s\star} \mid X^0_{s\star})$.

By sampling from the distribution of activation functions during training we are introducing a stochastic element into the training procedure. Consequently parametric GPNs further up in a feed-forward network see a noisy version of the processed inputs and learn to perform predictions even under presence of this noise, making the model resilient against overfitting on the training data. This is similar to the regularization technique Dropout (Srivastava et al., 2014), where the output of neurons is randomly set to zero with a given probability. Compared to Dropout however, the amount of noise induced in a parametric GPN is not constant but data-dependent. This follows from the variance of the probabilistic activation function being depended on the inputs to the parametric GPN, i.e. close to an inducing point the activation function has low variance, but far away from any inducing point it becomes almost completely random.

While stochastic training has been proven to find parameter optima that generalize better to unseen data samples, it also slows training down considerably because it leads to noisy estimates of the gradient, which, in turn, require the learning rate to be kept at least a magnitude lower compared to noise-free algorithms to prevent the trajectory in parameter space from becoming unstable. This is also a drawback of the Dropout technique mentioned above. To increase the speed of training and make it comparable with that of a conventional neural network, we will therefore demonstrate how to analytically propagate distributions through a parametric GPN network in section 5.5.

On the negative side the maximum likelihood estimation of the parameters performed in this section increases the risk of overfitting. Section 5.6 will show how to prevent this by performing Bayesian inference using appropriate priors.

5.4 Monotonic Activation Functions

Many commonly used activation functions, such as the sigmoid, hyperbolic tangent and (soft) rectifier are non-decreasing, i.e. their first derivative is non-negative for all values of the function. The idea behind using non-decreasing activation functions comes from treating a neuron as a thresholding unit that becomes active when the weighted sum of its inputs reaches a predefined threshold. This behavior would be best captured by using the step function as the activation function; however since its derivative is almost everywhere zero, it is not a suitable choice for a model that needs to be differentiable for training. Thus a softer version of the step function like the logistic function and its relatives, which all have in common that they are increasing functions, are used. Furthermore, it can be shown analytically (Wu, 2009) that the loss function of a neural network with a single layer is convex and thus only has one minimum, when a monotonic activation function is used. Taking into account these arguments, it might be desirable to enforce that activation functions modeled by a GPN are also monotonic.

As it can be seen from fig. 5.8a monotonicity cannot be imposed by simply requiring the targets of the inducing points to be increasing, since the zero mean function of the GP pulls the mean back to zero in areas without virtual observations. Moreover samples from a GP with an increasing mean function, fig. 5.8b, can be non-monotonous due the variability of the sampled function between the virtual observations. Hence, there exist several methods of varying complexity that achieve different levels of monotonicity of the activation function. We start by describing how the ensure a monotonous mean function with slight effort and continue by developing a more elaborate method that ensures that nearly every activation function sample is monotonous.

5.4.1 Increasing Activation Function Mean

A positive mean function can be enforced by calculating the derivative of the activation function at a finite number of derivative check points and adding a penalty term for negative derivative values at these points to the loss function of the model. Since a GP is differentiable the mean of the derivative can be calculated analytically from the inducing points and targets using (2.84) without the need for numerical differentiation. Let $\boldsymbol{c} \in \mathbb{R}^d$ be a vector of D derivative check points; for example, a set of evenly spaced points between the left-most and right-most inducing point of the GPN. For each parametric GPN with index n within a layer we obtain for the mean of the derivative

$$\mu_{\star n}^{X'} = K'(\boldsymbol{c}, V_{\star n}) \left[K(V_{\star n}, V_{\star n}) + \operatorname{diag}(S_{\star n})\right]^{-1} U_{\star n} \tag{5.68}$$

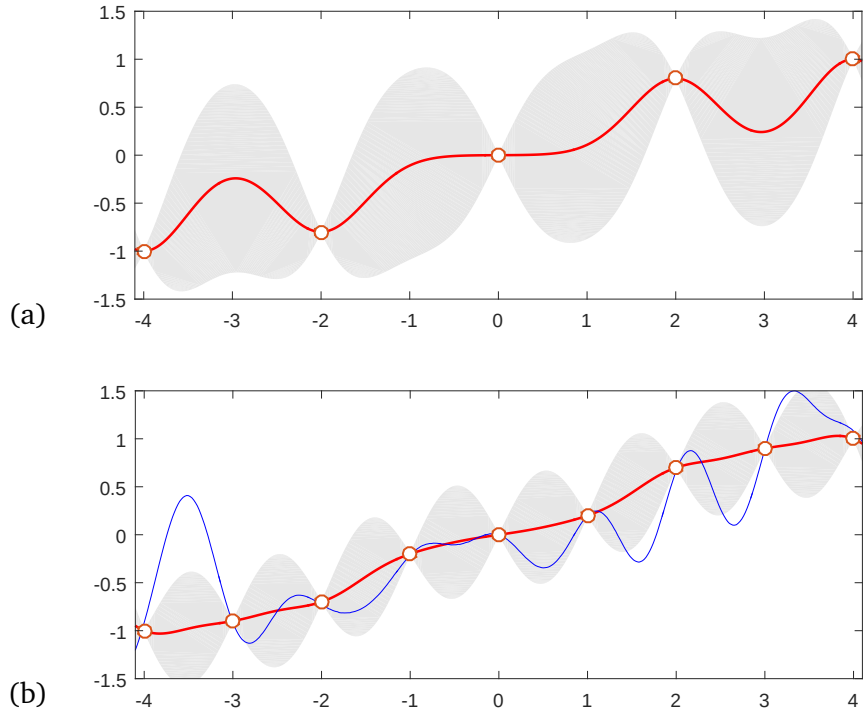


Figure 5.8: (a) Increasing targets (red circles) for inducing points do not necessarily lead to an increasing mean function (red line). (b) Furthermore, even with an increasing mean function a sample (blue line) from the GP can still be non-monotonous.

where the derivative covariance matrix is given by

$$[K'(\boldsymbol{c}, V_{\star n})]_{dr} \triangleq -2\exp\left(-(c_d - V_{rn})^2\right)\,(c_d - V_{rn})\,. \tag{5.69}$$

The penalty term is then given by

$$\mathscr{L}_{\mathfrak{m}}(\theta) = \frac{\alpha_{\mathfrak{m}}}{N\,D}\sum_{n=1}^{N}\sum_{d=1}^{D}\sigma(-\beta_{\mathfrak{m}}\,\mu_{dn}^{X'}) \tag{5.70}$$

where $\sigma(t)$ is the logistic function and $\alpha_{\mathfrak{m}} \gg 1$ and $\beta_{\mathfrak{m}} \gg 1$ are constants that control the strength of the penalty. Thus a value of α_m is added to the loss for each derivative check point with a negative derivative value. Adding $\mathscr{L}_{\mathfrak{m}}$ to the loss of the model results ensures that the means of the activation functions of all parametric GPNs are increasing, if the derivative check points are placed densely enough.

5.4.2 Increasing Activation Function Samples

As seen from fig. 5.8b even a GP with an increasing mean can give rise to non-increasing function samples. In fact, due to the stochastic nature of a GP it is impossible to guarantee that each function sample will be increasing, unless the observations are so closely packed the the GP effectively becomes deterministic. However, it is possible to greatly reduce the probability of obtaining non-increasing function samples by introducing *virtual observations of the derivative* into the GP at locations where it shows a significant probability of becoming negative.

A method for choosing this virtual derivative observations has been suggested by Riihimäki et al. (2010). The authors place a monotonicity prior on the GP and use the expectation propagation algorithm (Minka, 2001) to infer the derivative observations in a Bayesian framework. While the proposed technique works well, the use of expectation propagation makes it computationally expensive and particularly problematic to include in a model that is trained by gradient descent, such as a parametric GPN network. As an alternative we develop a method based on co-optimizing a secondary loss that penalizes non-increasing function samples w.r.t. the *variances* of the virtual derivative observations.

In addition to the derivate check points $\boldsymbol{c}$ we introduce R' virtual derivative observations per parametric GPN. The locations of the derivative observations are denoted by $V'_{r'n}$ and the derivative targets are $U_{r'n}$ with $r' \in \{1, \dots R'\}$. Each derivative observation is associated with a variance $S'_{r'n} \geq 0$ that controls the influence of that virtual observation on the activation function of the GPN. The locations $V'_{r'n}$ of the derivative observations can be chosen arbitrarily; however it is most efficient to evenly place the derivative observation locations between the left- and right-most inducing point of a GPN. The corresponding targets $U'_{r'n}$ are set to be the slope between the targets of the neighboring inducing points; i.e. let r^- be the index of the inducing point V_{r^-n} that is nearest to the left of $V'_{r'n}$ and let r^+ be the index of the inducing point V_{r^+n} that is nearest to the right of $V'_{r'n}$, then we set

$$U'_{r'n} = \frac{U_{r^+n} - U_{r^-n}}{V_{r^+n} - V_{r^-n}}\,. \tag{5.71}$$

The derivative targets $U'_{r'n}$ are updated accordingly when the targets U change. Once initialized the locations $V'_{r'n}$ remain fixed during training of the model. Note that here we assume that the inducing points V of a parametric GPN do not move much during training. This is an observation we have made during empirical evaluations of parametric GPN models.

The variances $S'_{r'n}$ are variable and used to control the influence of the derivative observations. For $S'_{r'n} \gg 1$ the derivative observation r' becomes insignificant and the GP behaves as if it was not present. As $S'_{r'n}$ gets smaller the derivative observation becomes more important

and for $S'_{r'n} = 0$ all function samples f obtained from the GP will have $f'(V'_{r'n}) = U'_{r'n}$. The idea is thus to use a low derivative variance $S'_{r'n}$ in regions where the GP has a high risk of producing non-increasing functions and otherwise let $S'_{r'n} \to \infty$ to keep the GP as unconstrained as possible. We do so by introducing a secondary loss function that penalizes function samples that are non-increasing at the check points $\boldsymbol{c}$. It is given by

$$\mathscr{L}_{\mathfrak{s}} = \frac{\alpha_{\mathfrak{s}}}{N\,D} \sum_{n=1}^{N} \sum_{d=1}^{D} \sigma\big(-\beta_{\mathfrak{s}}\, X'_{dn}\big) \tag{5.72}$$

where σ is the logistic function and $\alpha_{\mathfrak{s}}$ and $\beta_{\mathfrak{s}}$ serve the same function as $\alpha_{\mathfrak{m}}$ and $\beta_{\mathfrak{m}}$ respectively. The random variable X'_{dn} is distributed according to

$$X'_{dn} \sim \mathcal{N}(\mu^{X'}_{dn}, (\sigma^{X'}_{dn})^2) \tag{5.73}$$

with its mean and variance given by

$$\mu^{X'}_{dn} = K^*_d\, \widetilde{K}^{-1}\, \widetilde{U}\,, \tag{5.74a}$$

$$(\sigma^{X'}_{dn})^2 = K''(c_d, c_d) - K^*_d\, \widetilde{K}^{-1}\, (K^*_d)^T\,. \tag{5.74b}$$

where the covariance matrices and targets are

$$K^*_d \triangleq \begin{bmatrix} K'(c_d, V_{\star n}) & K''(c_d, V'_{\star n}) \end{bmatrix}\,, \tag{5.75a}$$

$$\widetilde{K} \triangleq \begin{bmatrix} K(V_{\star n}, V_{\star n}) + \operatorname{diag}(S_{\star n}) & K'(V'_{\star n}, V_{\star n})^T \\ K'(V'_{\star n}, V_{\star n}) & K''(V'_{\star n}, V'_{\star n}) + \operatorname{diag}(S'_{\star n}) \end{bmatrix}\,, \tag{5.75b}$$

$$\widetilde{U} \triangleq \begin{bmatrix} U_{\star n} \\ U'_{\star n} \end{bmatrix}\,. \tag{5.75c}$$

The blocks of the covariance matrices are given by the corresponding derivatives (2.83) and thus evaluate to

$$[K'(\boldsymbol{x}', \boldsymbol{y})]_{r't} = -2 \exp\big(-(x'_{r'} - y_t)^2\big)\ (x'_{r'} - y_t)\,, \tag{5.76a}$$

$$[K''(\boldsymbol{x}', \boldsymbol{y}')]_{r't'} = \exp\big(-(x'_{r'} - y'_{t'})^2\big)\ \big(2 - 4(x'_{r'} - y'_{t'})^2\big)\,. \tag{5.76b}$$

The expectation of the monotonicity loss (5.72) can either be evaluated by sampling from (5.73) or, more efficiently, by using the unscented transform. Since the activation function and its derivative are one-dimensional, the unscented transform (section 2.2.11) can be easily applied without the need to perform a Cholesky decomposition. The approximate expectation

of the monotonicity loss is thus given by

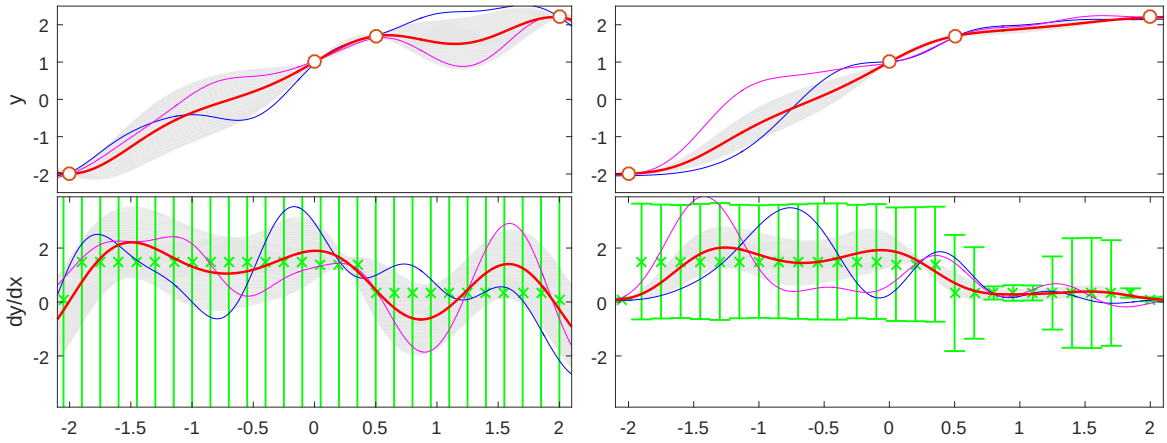
$$\begin{aligned}\mathrm{E}_{\mathrm{P}(X')}[\mathscr{L}_{\mathfrak{s}}] \approx \frac{\alpha_{\mathfrak{s}}}{N\,D}\sum_{n=1}^{N}\sum_{d=1}^{D}\frac{1}{1+\kappa}\Bigg[&\kappa\,\sigma\Big(-\beta_{\mathfrak{s}}\,\mu_{dn}^{X'}\Big)\\ &+\frac{1}{2}\sigma\Big(-\beta_{\mathfrak{s}}\mu_{dn}^{X'}-\beta_{\mathfrak{s}}\sqrt{(1+\kappa)(\sigma_{dn}^{X'})^2}\Big)\\ &+\frac{1}{2}\sigma\Big(-\beta_{\mathfrak{s}}\mu_{dn}^{X'}+\beta_{\mathfrak{s}}\sqrt{(1+\kappa)(\sigma_{dn}^{X'})^2}\Big)\Bigg]\end{aligned} \tag{5.77}$$

Since the virtual derivative observations should only influence the GPN when necessary to avoid a non-increasing activation function, the monotonicity loss shall include a term that drives the precision $1/S'_{r'n}$ to zero. This can be achieved by

$$\mathscr{L}_{\mathfrak{p}} = \frac{\alpha_{\mathfrak{p}}}{N\,R'}\sum_{n=1}^{N}\sum_{r'=1}^{R'}\frac{1}{S'_{r'n}} \tag{5.78}$$

with $\alpha_{\mathfrak{p}} \ll \alpha_{\mathfrak{s}}$ since (5.77) quickly goes to zero when the derivative is positive, due to the use of the logistic function. The monotonicity loss (5.77) and derivative precision regularization (5.78) are added to the model loss and minimized w.r.t. the variances $S'_{r'n}$.

Let us demonstrate the working principle of this method on a GPN with fixed inducing points U, V and S. An example is shown in fig. 5.9. Without virtual derivative observations (derivative variance S' is infinite, fig. 5.9a) the lower panel shows that the derivative of the GP has areas where it becomes negative with high probability. After the sum of the losses (5.77) and (5.78) is optimized w.r.t. S' (fig. 5.9b) the virtual derivative observations make it unlikely for the GP to produce a non-increasing function. The variance of these observations remains as high as possible, thus we obtain a GP the derivative of which is constrained only as much as necessary.



(a) initialization (b) after minimization of penalty

Figure 5.9: Encouraging a GP to produce increasing function samples by optimizing the sum of the monotonicity losses (5.77) and (5.78) w.r.t. the variances of virtual derivative observations. The upper panel shows the GP with mean function (red) and two samples (blue and magenta); observations are shown as red circles. The lower panel shows the derivative of the GP with virtual derivative observations indicated by green crosses with lines showing their standard deviation. (a) The variance of the virtual derivative observations is initialized with infinity and thus they do not affect the GP. The target derivative value is set to the slope between the two neighboring inducing points. (b) After optimizing the monotonicity loss, the virtual derivative observations have low variance where the GP would have high risk of producing function with negative derivative, for example around $x \approx 1$. At regions where the GP produces increasing function anyway, here between -1.5 and 0, the variance of the virtual derivative observations remains high.

5.5 Central Limit Activations

The reason for approximating the loss and its derivatives by sampling in section 5.3.4 is that a parametric GPN is able to completely change the shape of a distribution propagated through it. This can be seen as follows. Given enough virtual observations the activation function of an parametric GPN can approximate an arbitrary function. Thus, taking for instance the step function as the activation function of the GPN, we see immediately that a normal input distribution will result in an output distribution that is not normal. Given the appropriate choice of inducing points and targets, it is even possible for a GPN to transform a, say, uniform distribution into a normal distribution or vice versa. Intuitively speaking, the incoming normal distribution passes through the activation function, which is smooth but arbitrary, and thus the distribution will be deformed arbitrarily. Hence, in general it is not possible to find a class of distributions that is closed under propagation through a GPN. However, in practice a multivariate normal is still a very good approximation of the occurring distributions, as we will show in this section.

Consider the excerpt of a GPN feed-forward network shown in fig. 5.10. Since the inputs are observed, the distribution of each GPN in layer 1 is a univariate normal distribution with all GPNs being pairwise independent. Thus their joint distribution is a multivariate normal with diagonal covariance,

$$X^1_{s\star} \sim \mathcal{N}(\mu^{X^1}_{s\star}, \operatorname{diag}(\sigma^{X^1}_{s\star}))\,. \tag{5.79}$$

and consequently the distribution over the activations of the GPNs in layer 2 is given by

$$A^2_{s\star} \sim \mathcal{N}(\mu^{A^2_{s\star}}, \Sigma^{A^2_{s\star}})$$

with mean and covariance given by

$$\begin{aligned}\mu^{A^2_{s\star}} &= W^2 \mu^{X^1_{s\star}} \\ \Sigma^{A^2_{s\star}} &= W^2 \operatorname{diag}(\sigma^{X^1}_{s\star})\,(W^2)^T\,.\end{aligned}$$

We now have to discriminate between two cases.

If the variances in layer 1 are small, $\sigma^{X^1}_{sn} \ll 1$, and the weights are equally distributed around zero[3], $\langle W^2_{nm}\rangle_{nm} \approx 0$, we can assume that the variances of the activations in layer 2 will also be reasonably small, $\Sigma^{A^2_{s\star}}_{ii} \ll 1$, so that a first-order Taylor expansion[4] of the activation

[3] $\langle\bullet\rangle_{nm}$ denotes the average over the elements of a matrix as defined in eq. (2.1).

[4] For the sake of this argument we ignore the additional variance imposed by the GP representing the activation function.

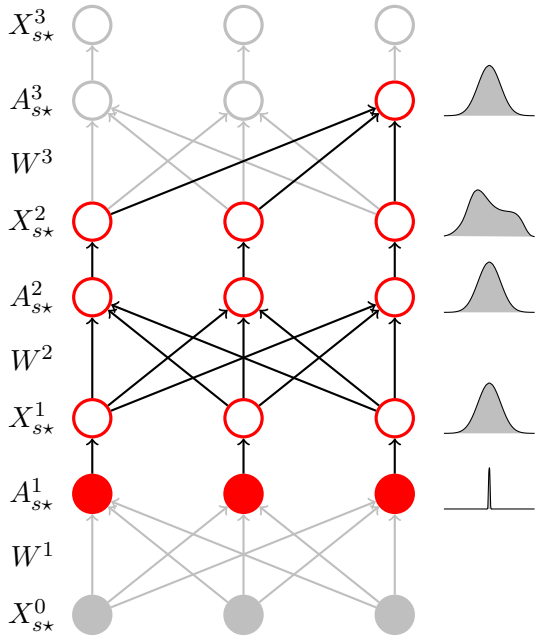


Figure 5.10: The principle of the normal approximation in a feed-forward GPN network. Propagating the distribution of the activations $A^2_{s\star}$ through the activation functions results in arbitrary distributions $X^2_{s\star}$. However, given enough GPNs in layer 1 and 2, the responses X^2_{sn}, $n \in \{1, 2, \dots N_2\}$, are only weakly correlated and thus the distribution of the activations $A^3_{s\star}$ will again resemble a normal distribution due to the central limit theorem.

function of each GPN around the mean,

$$F_n(A) = F(\mu^{A^2_{sn}}) + \left.\frac{\mathrm{d}F}{\mathrm{d}A}\right|_{A=\mu^{A^2_{sn}}} (A - \mu^{A^2_{sn}}) + O\Big((A - \mu^{A^2_{sn}})^2\Big) \,,$$

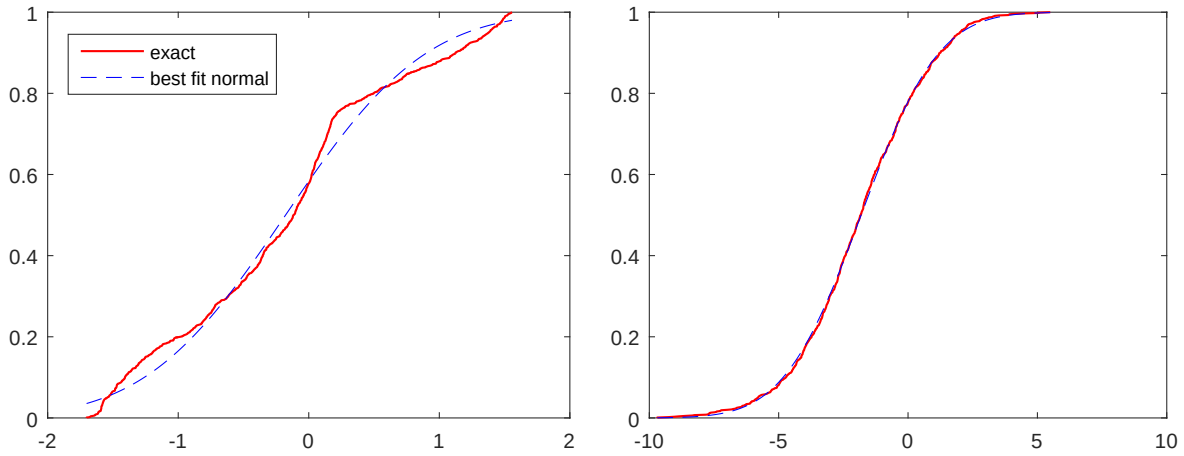
can be used to propagate the distribution of A^2_{sn} through the activation function $F_n(A)$. Hence, it follows that in this case the distribution of X^2_{sn} must also be normal, since applying an affine transformation to a normal distribution results in another normal distribution.

If the variances $\Sigma^{A^2_{s\star}}_{ii}$ are not sufficiently small, the response distribution X^2_{sn} is arbitrary. However, if the activation functions are sufficiently uncorrelated, the covariance statistics remain approximately unchanged; thus it will hold that

$$\langle \mathrm{Cov}(X^2_{sn}, X^2_{sm}) \rangle_{n \neq m} \approx \langle \mathrm{Cov}(A^2_{sn}, A^2_{sm}) \rangle_{n \neq m} \,.$$

Assuming random layer 2 weights W^2, we see that by writing the covariance matrix as

$$\Sigma^{A^2_{s\star}}_{nm} = |V_{n\star}|\,|V_{m\star}| \cos \measuredangle(V_{n\star}, V_{m\star}) \qquad \text{with } V \triangleq W^2 \,\mathrm{diag}(\sigma^{X^1}_{s\star})^{\frac{1}{2}} \,,$$



(a) layer 1 and 2 both have 3 GPNs (b) layer 1 and 2 both have 10 GPNs

Figure 5.11: The figure shows the CDF of the distribution of activations of a GPN in layer 3 of the GPN feed-forward network shown in fig. 5.10. The red line shows the empirical distribution obtained by propagating 1000 draws from $X^1_{s\star}$ through layer 2 and applying the weights W^3. The dashed blue line is the CDF of a best-fit normal distribution on this data. Given random weights and activation functions, 10 GPNs seem to be enough for the activations $A^3_{s\star}$ to resemble a Gaussian.

the expected value of the off-diagonal elements is determined by the cosine of the angle between elements of V. Since the weights are assumed to be iid.random, this implies

$$\langle \cos \measuredangle(V_{n\star}, V_{m\star}) \rangle_{n \neq m} \approx 0\,. \tag{5.80}$$

This makes the central limit theorem for dependent variables, cf. section 2.2.5, applicable on the layer 3 activations

$$A^3_{si} = \sum_n W^3_{in} X^2_{sn}\,,$$

since (5.80) ensures that the variables X^2_{sn} are weakly correlated at most and thus the prerequisite for the central limit theorem that

$$\tau^2_{N_2} = \text{const} + \frac{1}{N_2} \sum_{n=1}^{N_2} \sum_{\substack{m=1 \\ m \neq n}}^{N_2} \text{Cov}(W^3_{in} X^2_{sn}, W^3_{im} X^2_{sm})$$

stays finite for $N_2 \to \infty$ is fulfilled. Consequently even for large variances $\Sigma^{A^2_{s\star}}_{ii}$ the central limit theorem suggests that the activations $A^3_{s\star}$ of the following layer will be normally distributed due to summation over a large number of weakly correlated variables.

We can also test the claim of having normally distributed activations in each layer experi-

mentally. For that purpose we instantiate three-layer GPN feed-forward networks (see fig. 5.10) with different numbers of GPNs in layer 1 and 2. Their weights are sampled from a standard normal distribution and the activation function of each GPN is randomly sampled from a zero-mean GP with the SE covariance function with unit lengthscale. A random input vector is drawn and the parameters of the distribution for $X^1_{s\star}$ given by (5.79) are calculated. Then 1000 samples are drawn from $X^1_{s\star}$ and propagated through each network until reaching $A^3_{s\star}$. The resulting empirical CDFs of $A^3_{s\star}$ for two networks are shown in fig. 5.11 together with the best-fit normal CDFs. For a very small network with only 3 GPNs on both layers it is apparent that the activations are not normally distributed. As the number of GPNs increases the distribution becomes more normal and having 10 GPNs in both layers is enough for the distribution to resemble a Gaussian very closely.

A similar analysis has been performed by (Wang et al., 2013) on a conventional neural network. In their work the authors show empirically that the activations of each neuron remain normally distributed even *after* the network has been trained to convergence. We can therefore deduce that for the weights in a trained network, eq. (5.80) still holds and thus our argument remains valid during and after training.

In conclusion, we have motivated analytically and shown empirically that assuming a multivariate normal distribution over $A^l_{s\star}$ for all layers l is a reasonable approximation for non-degenerate feed-forward networks of parametric GPNs. The best choice for the parameters of the approximating normal is to use the same mean and covariance as the true distribution, i.e.

$$\widetilde{\mathrm{P}}(A^l_{s\star}) = \mathcal{N}(A^l_{s\star} \,|\, \widetilde{\mu}^{A^l}_{s\star}, \widetilde{\Sigma}^{A^l}_{s\star\star}) \tag{5.81}$$

with

$$\widetilde{\mu}^{A^l}_{sn} \triangleq \mathrm{E}_{\mathrm{P}(A^l_{s\star} \,|\, A^{l-1}_{s\star})\widetilde{\mathrm{P}}(A^{l-1}_{s\star})}[A^l_{sn}] \tag{5.82}$$

$$\widetilde{\Sigma}^{A^l}_{snm} \triangleq \mathrm{Cov}_{\mathrm{P}(A^l_{s\star} \,|\, A^{l-1}_{s\star})\widetilde{\mathrm{P}}(A^{l-1}_{s\star})}(A^l_{sn}, A^l_{sm})\,, \tag{5.83}$$

and we must now find how to propagate these statistics from layer to layer. It turns out that it can be done exactly using closed-form equations that will be derived in the following section.

5.5.1 Propagation of Mean and Covariance

Since $A^l_{s\star} = W^l X^{l-1}_{s\star}$ we have for each layer

$$\widetilde{\mu}^{A^l}_{s\star} = W^l \, \widetilde{\mu}^{X^{l-1}}_{s\star} \tag{5.84a}$$
$$\widetilde{\Sigma}^{A^l}_{s\star\star} = W^l \, \widetilde{\Sigma}^{X^{l-1}}_{s\star\star} \, (W^l)^T \,, \tag{5.84b}$$

where $\widetilde{\mu}^{X^{l-1}}$ and $\widetilde{\Sigma}^{X^{l-1}}$ are the mean and variance of X^{l-1}, which is not normally distributed. We now have to calculate $\widetilde{\mu}^{X^l}$ and $\widetilde{\Sigma}^{X^l}$ given $\widetilde{\mu}^{A^l}$ and $\widetilde{\Sigma}^{A^l}$.

The first layer, $l = 1$, already follows a normal distribution, since its inputs and thus activations are deterministic. Thus we set

$$\widetilde{\mu}^{X^1}_{sn} = \mu^{X^1}_{sn} \tag{5.85a}$$
$$\widetilde{\Sigma}^{X^1}_{snm} = \Sigma^{X^1}_{ssn} \, \delta_{nm} \tag{5.85b}$$

with $\mu^{X^1}_{sn}$ and $\Sigma^{X^1}_{ssn}$ given by (5.47). Note that in the normal approximation $\widetilde{\Sigma}$ describes the covariance between different GPNs within a layer while Σ from (5.47b) captures the covariance between different samples for the same parametric GPN.

For the following layers, $l > 1$, we proceed as follows. To calculate the mean vector we apply the law of total expectation and get

$$\widetilde{\mu}^{X^l}_{sn} = \mathrm{E}[X^l_{sn}] = \mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{s\star})}\left[\mathrm{E}_{\mathrm{P}(X^l_{s\star} \,|\, A^l_{s\star})}[X^l_{sn}]\right] = \mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{s\star})}[\mu^{X^l}_{sn}] \,. \tag{5.86}$$

By inserting $\mu^{X^l}_{sn}$ from (5.47a) we further obtain

$$\widetilde{\mu}^{X^l}_{sn} = \left(\mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{s\star})}[\alpha^l_{\star n}]\right)^T \kappa^l_{\star\star n} \, U^l_{\star n} \tag{5.87}$$

with

$$\alpha^l_{rn} \triangleq k(A^l_{sn}, V^l_{rn}) \tag{5.88}$$
$$\kappa^l_{\star\star n} \triangleq [K(V^l_{\star n}, V^l_{\star n}) + \mathrm{diag}(S^l_{\star n})]^{-1} \,. \tag{5.89}$$

The expectation becomes

$$\mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{sn})}[\alpha^l_{rn}] = \int k(A^l_{sn}, V^l_{rn}) \, \mathcal{N}(A^l_{sn} \,|\, \widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn}) \, \mathrm{d}A^l_{sn} \,. \tag{5.90}$$

Noting that the SE covariance function can be written as the unnormalized PDF of a Gaussian,

$$k(a, a') = \sqrt{\pi}\,\mathcal{N}\left(a \,\middle|\, a', \frac{1}{2}\right),$$

we can further calculate

$$\mathrm{E}[\alpha_{rn}^l] = \sqrt{\pi} \int \mathcal{N}(A_{sn}^l \,|\, V_{rn}^l, 1/2)\,\mathcal{N}(A_{sn}^l \,|\, \widetilde{\mu}_{sn}^{A^l}, \widetilde{\Sigma}_{snn}^{A^l})\,\mathrm{d}A_{sn}^l \tag{5.91}$$

and by applying the product formula for two Gaussian PDFs (2.23) we obtain another Gaussian PDF,

$$\sqrt{\pi}\,\mathcal{N}(A_{sn}^l \,|\, V_{rn}^l, 1/2)\,\mathcal{N}(A_{sn}^l \,|\, \widetilde{\mu}_{sn}^{A^l}, \widetilde{\Sigma}_{snn}^{A^l}) = \widehat{\mathcal{S}}\,\mathcal{N}(A_{sn}^l \,|\, \widehat{\mu}, \widehat{\sigma}^2) \tag{5.92}$$

with

$$\frac{1}{\widehat{\sigma}^2} = 2 + \frac{1}{\widetilde{\Sigma}_{snn}^{A^l}} \tag{5.93a}$$

$$\widehat{\mu} = \left(2V_{rn}^l + \frac{\widetilde{\mu}_{sn}^{A^l}}{\widetilde{\Sigma}_{snn}^{A^l}}\right)\widehat{\sigma}^2 \tag{5.93b}$$

$$\widehat{\mathcal{S}} = \sqrt{\frac{1}{1 + 2\widetilde{\Sigma}_{snn}^{A^l}}}\,\exp\left(-\frac{(\widetilde{\mu}_{sn}^{A^l} - V_{rn}^l)^2}{1 + 2\widetilde{\Sigma}_{snn}^{A^l}}\right) \tag{5.93c}$$

After substitution into (5.91) only the factor $\widehat{\mathcal{S}}$ remains since the integral over any PDF is one, thus we obtain

$$\psi_{rn}^l \triangleq \mathrm{E}[\alpha_{rn}^l] = \widehat{\mathcal{S}} \tag{5.94}$$

This concludes the calculation of the mean of X^l.

For the covariance matrix we obtain by using the definition of the covariance and expanding the occurring expectations as above

$$\widetilde{\Sigma}_{snm}^{X^l} = \mathrm{Cov}(X_{sn}^l, X_{sm}^l) = \mathrm{E}_{\widetilde{\mathrm{P}}(A_{s\star}^l)}\left[\mathrm{E}_{\mathrm{P}(X_{s\star}^l \,|\, A_{s\star}^l)}[X_{sn}^l\, X_{sm}^l]\right] - \widetilde{\mu}_{sn}^{X^l}\,\widetilde{\mu}_{sm}^{X^l}\,. \tag{5.95}$$

We must now differentiate between on-diagonal and off-diagonal elements of the covariance matrix. For diagonal elements, $n = m$, we note that $\mathrm{E}[X^2] = \mathrm{Var}(X) + \mathrm{E}[X]^2$ and obtain

$$\widetilde{\Sigma}_{snn}^{X^l} = \mathrm{E}_{\widetilde{\mathrm{P}}(A_{s\star}^{l-1})}\left[\Sigma_{ssn}^{X^l} + (\mu_{sn}^{X^l})^2\right] - (\widetilde{\mu}_{sn}^{X^l})^2\,. \tag{5.96}$$

The first term of the sum in the expectation evaluates to

$$\mathrm{E}_{\widetilde{\mathrm{P}}(A_{s\star}^l)}\left[\Sigma_{ssn}^{X^l}\right] = 1 - \mathrm{E}\left[(\alpha_{s\star}^l)^T\,\kappa_{\star\star n}^l\,\alpha_{s\star}^l\right] + (\sigma_n^l)^2\,. \tag{5.97}$$

which can further be expanded into

$$\mathrm{E}\Big[(\alpha^l_{s\star})^T\,\kappa^l_{\star\star n}\,\alpha^l_{s\star}\Big] = \sum_r \sum_t \kappa^l_{rtn}\,\Omega^l_{rtn}\,.$$

with $\Omega^l_{rtn} \triangleq \mathrm{E}[\alpha^l_{rn}\,\alpha^l_{tn}]$. Rewriting α^l_{rn}, α^l_{tn} as Gaussian PDFs and combining them with the normal distribution over A^l_{sn} using the product formula for Gaussian PDFs (2.25) gives after simplification

$$\sqrt{\pi}\mathcal{N}(A^l_{sn}\,|\,V^l_{rn}, 1/2)\,\sqrt{\pi}\mathcal{N}(A^l_{sn}\,|\,V^l_{tn}, 1/2)\,\mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn}) = \widetilde{\mathcal{S}}\,\mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}, \widetilde{\sigma}^2) \tag{5.98}$$

with

$$\widetilde{\sigma}^2 = \frac{\widetilde{\Sigma}^{A^l}_{snn}}{1 + 4\widetilde{\Sigma}^{A^l}_{snn}} \tag{5.99a}$$

$$\widetilde{\mu} = \frac{\widetilde{\Sigma}^{A^l}_{snn}}{1 + 4\widetilde{\Sigma}^{A^l}_{snn}} \left(2V^l_{rn} + 2V^l_{tn} + \frac{\widetilde{\mu}^{A^l}_{sn}}{\widetilde{\Sigma}^{A^l}_{snn}}\right) \tag{5.99b}$$

$$\widetilde{\mathcal{S}} = \sqrt{\frac{1}{1 + 4\widetilde{\Sigma}^{A^l}_{snn}}}\,\exp\left(-\frac{2\left(\widetilde{\mu}^{A^l}_{sn} - \frac{V^l_{rn}+V^l_{tn}}{2}\right)^2}{1 + 4\widetilde{\Sigma}^{A^l}_{snn}} - \frac{(V^l_{rn} - V^l_{tn})^2}{2}\right). \tag{5.99c}$$

Thus we have $\Omega^l_{rtn} = \widetilde{\mathcal{S}}$ due to integrating over a normalized PDF. The second term in the expectation evaluates to

$$\mathrm{E}\Big[(\mu^{X^l}_{sn})^2\Big] = \mathrm{E}\Big[(\beta^l_{\star n})^T\,\alpha^l_{\star n}\,(\alpha^l_{\star n})^T\,\beta^l_{\star n}\Big] = \sum_r \sum_t \beta^l_{rn}\beta^l_{tn}\,\Omega^l_{rtn} \tag{5.100}$$

where we have defined $\beta^l_{\star n} \triangleq \kappa^l_{\star\star n}\,U^l_{\star n}$. Inserting the derived terms into (5.96) and simplifying gives

$$\widetilde{\Sigma}^{X^l}_{snn} = 1 - \mathrm{tr}\Big[\Big(\kappa^l_{\star\star n} - \beta^l_{\star n}\,(\beta^l_{\star n})^T\Big)\,\Omega^l_{\star\star n}\Big] - \mathrm{tr}\Big(\psi^l_{\star n}(\psi^l_{\star n})^T\,\beta^l_{\star n}(\beta^l_{\star n})^T\Big) + (\sigma^l_n)^2\,. \tag{5.101}$$

An alternative derivation of the mean and variance giving the same result can be found in (Quiñonero-Candela, Girard, et al., 2002).

For off-diagonal elements, $n \neq m$, we observe that X^l_{sn} and X^l_{sm} are *conditionally independent* given $A^l_{s\star}$ because the activation functions of neurons n and m are represented by two different GPs. The correlation between GPNs within a layer l is induced by being conditioned

on the same set of inputs $X_{s\star}^{l-1}$. Exploiting this conditional independence gives

$$\begin{aligned}\widetilde{\Sigma}_{snm}^{X^l} &= \mathrm{E}\Big[\mu_{sn}^{X^l}\,\mu_{sm}^{X^l}\Big] - \widetilde{\mu}_{sn}^{X^l}\,\widetilde{\mu}_{sm}^{X^l} = \mathrm{E}\Big[(\beta_{\star n}^l)^T\,\alpha_{\star n}^l\,(\alpha_{\star m}^l)^T\,\beta_{\star m}^l\Big] - \widetilde{\mu}_{sn}^{X^l}\,\widetilde{\mu}_{sm}^{X^l} \\ &= \sum_r \sum_t \beta_{rn}^l \beta_{tm}^l\,\Lambda_{rtnm}^l - \widetilde{\mu}_{sn}^{X^l}\,\widetilde{\mu}_{sm}^{X^l} \end{aligned} \tag{5.102}$$

To evaluate $\Lambda_{rtnm}^l \triangleq \mathrm{E}[\alpha_{rn}^l \alpha_{tm}^l]$ we observe that

$$\begin{aligned}\alpha_{rn}^l \alpha_{tm}^l &= \exp\Big(-(A_{sn}^l - V_{rn}^l)^2 - (A_{sm}^l - V_{tm}^l)^2\Big) = \exp\left(-\frac{1}{2}(\boldsymbol{A}-\boldsymbol{V})^T 2(\boldsymbol{A}-\boldsymbol{V})\right) \\ &= 4\pi\,\mathcal{N}(\boldsymbol{A}\,|\,\boldsymbol{V}, L)\end{aligned} \tag{5.103}$$

where we have defined

$$\boldsymbol{A} \triangleq \begin{pmatrix} A_{sn}^l \\ A_{sm}^l \end{pmatrix}, \qquad \boldsymbol{V} \triangleq \begin{pmatrix} V_{rn}^l \\ A_{tm}^l \end{pmatrix}, \qquad L \triangleq \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}.$$

We now have

$$\Lambda_{rtnm}^l = 4\pi \iint \mathcal{N}(\boldsymbol{A}\,|\,\boldsymbol{V}, L)\,\mathcal{N}(\boldsymbol{A}\,|\,\widehat{\boldsymbol{\mu}}, \widehat{\Sigma})\,\mathrm{d}\boldsymbol{A}$$

with

$$\widehat{\boldsymbol{\mu}} = \begin{pmatrix} \widetilde{\mu}_{sn}^{A^l} \\ \widetilde{\mu}_{sm}^{A^l} \end{pmatrix}, \qquad \widehat{\Sigma} = \begin{pmatrix} \widetilde{\Sigma}_{snn}^{A^l} & \widetilde{\Sigma}_{snm}^{A^l} \\ \widetilde{\Sigma}_{smn}^{A^l} & \widetilde{\Sigma}_{smm}^{A^l} \end{pmatrix}$$

due to the marginalization property of the multivariate normal distribution over $A_{s\star\star}^l$. Applying the product formula for the PDFs of multivariate normal distributions (2.37) we obtain

$$\Lambda_{rtnm}^l = 4\pi \exp(\mathcal{S}_1 + \mathcal{S}_2 - \mathcal{S}_3) \tag{5.104}$$

with

$$\begin{aligned}\mathcal{S}_1 &= -\log 2\pi - \frac{1}{2}\log|L| - \frac{1}{2}\boldsymbol{V}^T L^{-1}\boldsymbol{V} \\ \mathcal{S}_2 &= -\log 2\pi - \frac{1}{2}\log\left|\widehat{\Sigma}\right| - \frac{1}{2}\widehat{\boldsymbol{\mu}}^T\widehat{\Sigma}^{-1}\widehat{\boldsymbol{\mu}} \\ \mathcal{S}_3 &= -\log 2\pi + \frac{1}{2}\log\left|L^{-1} + \widehat{\Sigma}^{-1}\right| - \frac{1}{2}\left(L^{-1}\boldsymbol{V} + \widehat{\Sigma}^{-1}\widehat{\boldsymbol{\mu}}\right)^T\left(L^{-1} + \widehat{\Sigma}^{-1}\right)^{-1}\left(L^{-1}\boldsymbol{V} + \widehat{\Sigma}^{-1}\widehat{\boldsymbol{\mu}}\right).\end{aligned}$$

Using the general matrix inversion identity

$$(A+B)^{-1} = A^{-1}\left(A^{-1}+B^{-1}\right)^{-1}B^{-1} = B^{-1}\left(A^{-1}+B^{-1}\right)^{-1}A^{-1}$$

the expression for $\mathcal{S}_3$ simplifies to

$$\begin{aligned}\mathcal{S}_3 = &-\log 2\pi + \frac{1}{2}\log\left|L^{-1} + \widehat{\Sigma}^{-1}\right| - \boldsymbol{V}^T\left(L^{-1} + \widehat{\Sigma}^{-1}\right)^{-1}\widehat{\boldsymbol{\mu}} - \\ &\frac{1}{2}\boldsymbol{V}^T\left(L^{-1} + \widehat{\Sigma}^{-1}\right)^{-1}\widehat{\Sigma}\,L^{-1}\,\boldsymbol{V} - \frac{1}{2}\widehat{\boldsymbol{\mu}}^T\left(L^{-1} + \widehat{\Sigma}^{-1}\right)^{-1} L\,\widetilde{\Sigma}^{-1}\,\widetilde{\boldsymbol{\mu}}\,.\end{aligned}$$

All matrices that occur in $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ are of size 2×2, thus their inverses and determinants can be calculated analytically in closed form using

$$\begin{aligned}\det M &= M_{11}\,M_{22} - M_{12}\,M_{21}\,,\\ M^{-1} &= \frac{1}{\det M}\begin{pmatrix} M_{22} & -M_{12} \\ -M_{21} & M_{11}\end{pmatrix}.\end{aligned}$$

Applying these formulas and substituting $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ into (5.104) gives after simplification

$$\Lambda^l_{rtnm} = \frac{\exp(\mathscr{A}^l_{rtnm}/\mathscr{B}^l_{rtnm})}{\sqrt{(1 + 2\widetilde{\Sigma}^{A^{l-1}}_{snn})\,(1 + 2\widetilde{\Sigma}^{A^{l-1}}_{smm}) - 4\widetilde{\Sigma}^{A^{l-1}}_{snm}}} \tag{5.105}$$

with

$$\begin{aligned}\mathscr{A}^l_{rtnm} = &(V^l_{tm} - \widetilde{\mu}^{A^l}_{sm})^2\,(1 + 2\widetilde{\Sigma}^{A^l}_{snn}) + (V^l_{rn} - \widetilde{\mu}^{A^l}_{sn})^2\,(1 + 2\widetilde{\Sigma}^{A^l}_{smm}) + \\ &4\,(V^l_{rn} - \widetilde{\mu}^{A^l}_{sn})\,(\widetilde{\mu}^{A^l}_{sm} - V^l_{tm})\,\widetilde{\Sigma}^{A^l}_{snm} \qquad (5.106a)\\ \mathscr{B}^l_{rtnm} = &(1 + 2\widetilde{\Sigma}^{A^l}_{nn})\,(1 + 2\widetilde{\Sigma}^{A^l}_{mm}) - 4(\widetilde{\Sigma}^{A^l}_{nm})^2 \qquad (5.106b)\end{aligned}$$

This concludes the calculation of the covariance matrix of X^l.

Having made the approximations shown in this section, we are able to analytically propagate the mean and covariance of all data points through all layers of a feed-forward GPN stack.

Including Virtual Derivative Observations

If virtual derivative observations are used to ensure that the activation function of the GPN is increasing, cf. section 5.4, the propagation of the mean and covariance must be adapted to accommodate these observations. To propagate the mean using virtual derivative observations, eq. (5.87) is replaced by

$$\widetilde{\mu}^{X^l}_{sn} = \left(\mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{s\star})}[\widetilde{\alpha}^l_{\star n}]\right)^T \widetilde{\kappa}^l_{\star\star n}\,\widetilde{U}^l_{\star n} \tag{5.107}$$

where we have $\widetilde{\kappa}^l_{\star\star n} \triangleq \widetilde{K}^{-1}$ with $\widetilde{K}$ and $\widetilde{U}$ given by (5.75b) and (5.75c). Furthermore,

$$\widetilde{\alpha}^l_{\star n} \triangleq \begin{bmatrix} \alpha^l_{\star n} \\ \alpha'^l_{\star n} \end{bmatrix}$$

with α^l_{rn} as previously defined in (5.88) and $\alpha'^l_{r'n} = K'(V'^l_{r'n}, A^l_{sn})$ given by (5.76a). The corresponding expectation value $\mathrm{E}[\alpha'^l_{r'n}]$ is straightforward to calculate,

$$\begin{aligned} \mathrm{E}[\alpha'^l_{r'n}] &= -2\left(V'^l_{r'n} - A^l_{sn}\right)\sqrt{\pi}\int \mathcal{N}(A^l_{sn}\,|\,V'^l_{r'n}, 1/2)\,\mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn})\,\mathrm{d}A^l_{sn} \\ &= -2\,\widehat{\mathcal{S}}\left(V'^l_{r'n} - \widehat{\mu}\right) = -2\exp\left(-\frac{(V'^l_{r'n} - \widetilde{\mu}^{A^l}_{sn})^2}{1+\widetilde{\Sigma}^{A^l}_{snn}}\right)\frac{V'^l_{r'n} - \widetilde{\mu}^{A^l}_{sn}}{(1+2\widetilde{\Sigma}^{A^l}_{snn})^{3/2}}\,, \end{aligned} \tag{5.108}$$

where $\widehat{\mathcal{S}}$ and $\widehat{\mu}$ are given by (5.93c) and (5.93b) respectively.

For the variance eq. (5.97) must be replaced by

$$\mathrm{E}_{\widetilde{\mathrm{P}}(A^l_{s\star})}\left[\Sigma^{X^l}_{ssn}\right] = 1 - \mathrm{E}\left[(\widetilde{\alpha}^l_{s\star})^T\,\widetilde{\kappa}^l_{\star\star n}\,\widetilde{\alpha}^l_{s\star}\right] + (\sigma^l_n)^2\,. \tag{5.109}$$

and the corresponding expectations evaluate to

$$\begin{aligned} \mathrm{E}[\alpha^l_{rn}\alpha^l_{tn}] &= \pi\int \mathcal{N}(A^l_{sn}\,|\,V^l_{rn}, 1/2)\,\mathcal{N}(A^l_{sn}\,|\,V^l_{tn}, 1/2)\,\mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn})\,\mathrm{d}A^l_{sn} \\ &= \widetilde{\mathcal{S}} \qquad (5.110a) \\ \mathrm{E}[\alpha'^l_{r'n}\alpha^l_{tn}] &= -2\pi\int \left(V'^l_{r'n} - A^l_{sn}\right)\mathcal{N}(A^l_{sn}\,|\,V'^l_{r'n}, 1/2)\,\mathcal{N}(A^l_{sn}\,|\,V^l_{tn}, 1/2)\cdot \\ &\qquad \mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn})\,\mathrm{d}A^l_{sn} \\ &= -2\,\widetilde{\mathcal{S}}\left(V'^l_{r'n} - \widetilde{\mu}\right) \qquad (5.110b) \\ \mathrm{E}[\alpha'^l_{r'n}\alpha'^l_{t'n}] &= 4\pi\int \left(V'^l_{r'n} - A^l_{sn}\right)\mathcal{N}(A^l_{sn}\,|\,V'^l_{r'n}, 1/2)\,\left(V'^l_{t'n} - A^l_{sn}\right)\,\mathcal{N}(A^l_{sn}\,|\,V'^l_{t'n}, 1/2)\cdot \\ &\qquad \mathcal{N}(A^l_{sn}\,|\,\widetilde{\mu}^{A^l}_{sn}, \widetilde{\Sigma}^{A^l}_{snn})\,\mathrm{d}A^l_{sn} \\ &= 4\,\widetilde{\mathcal{S}}\left[V'^l_{r'n}V'^l_{t'n} - \widetilde{\mu}\left(V'^l_{r'n} + V'^l_{t'n}\right) + \widetilde{\mu}^2 + \widetilde{\sigma}^2\right] \qquad (5.110c) \end{aligned}$$

with $\widetilde{\mu}$, $\widetilde{\sigma}^2$, and $\widetilde{\mathcal{S}}$ given by (5.99). Substituting these quantities into (5.101) propagates the variance from layer to layer including the virtual derivative observation to ensure monotonicity of the activation function samples.

Expectation of the Loss

It remains to calculate the expectation of the loss over the approximative normal distribution of the top-most GPN layer,

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} \mathrm{E}_{\widetilde{\mathrm{P}}_\theta(X^L_{s\star})}[L(X^L_{s\star}, T_{s\star})]\,. \tag{5.111}$$

In the case of regression the associated loss measure (5.54) results in

$$\mathscr{L}_{\mathrm{reg}}(\theta) = -\frac{1}{S}\sum_{s=1}^{S} \widetilde{\mathrm{P}}_\theta(X^L_{s\star} = T_{s\star})$$

and we obtain the standard negative log-likelihood objective function by applying Jensen's inequality over the sum,

$$\mathscr{L}_{\mathrm{ll}}(\theta) = -\frac{1}{S}\sum_{s=1}^{S} \log \widetilde{\mathrm{P}}_\theta(X^L_{s\star} = T_{s\star}) = -\frac{1}{S}\sum_{s=1}^{S} \mathcal{LN}(T_{s\star} \,|\, \widetilde{\mu}^{X^L}_{s\star}, \widetilde{\Sigma}^{X^L}_{s\star\star})\,. \tag{5.112}$$

Note that $\widetilde{\mu}^{X^L}_{s\star}$ and $\widetilde{\Sigma}^{X^L}_{s\star\star}$ are functions of the inputs $X^0_{s\star}$ and model parameters θ through iterated application of eqs. (5.87), (5.101) and (5.102). Thus under the normal approximation developed in this section the training objective for regression is a fully deterministic function. For this loss we made the additional assumption that the (learned) activation function of the last layer is linear enough so that X^L follows a normal distribution. This can easily be achieved without loss of expressive power by fixing the activation function of the last layer to the identity function. Then $X^L = A^L$ and the central limit theorem ensures a normal distribution.

For other loss functions, for example the cross-entropy loss for classification, we still need to evaluate the expectation in eq. (5.111). While we could compute the loss by sampling from $\widetilde{\mathrm{P}}_\theta(X^L_{s\star})$ and taking the average over the loss values for these samples, this would require our training procedure to be stochastic and thus not as efficient as when minimizing a deterministic objective such as (5.112). A method of propagating the mean and covariance of a normal distribution through an arbitrary function is the unscented transform, that was introduced in section 2.2.11. It works by propagating deterministically chosen points that represent the distribution through the function and using the transformed points to estimate the mean and covariance of the transformed distribution. Using the unscented transform the loss becomes

$$\mathscr{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S}\sum_{i=0}^{2N_L} W_i\, L(\boldsymbol{x}^s_i, T_{s\star}) \tag{5.113}$$

where $\boldsymbol{x}_i^s$ are the sigma points given by eqs. (2.41) and (2.47) for mean $\boldsymbol{\mu}^{\boldsymbol{x}} = \widetilde{\mu}_{s\star}^{X^L}$ and covariance $\Sigma^{\boldsymbol{x}} = \widetilde{\Sigma}_{s\star\star}^{X^L}$ and W_i are the corresponding weights of the unscented transform. Since the sigma points are differentiable w.r.t. the mean and covariance, the derivative of (5.113) w.r.t. the model parameters θ can be calculated using the chain rule. This method also assume that X^L follows a normal distribution thus it is sensible to fix the activation function of layer L to the identity function.

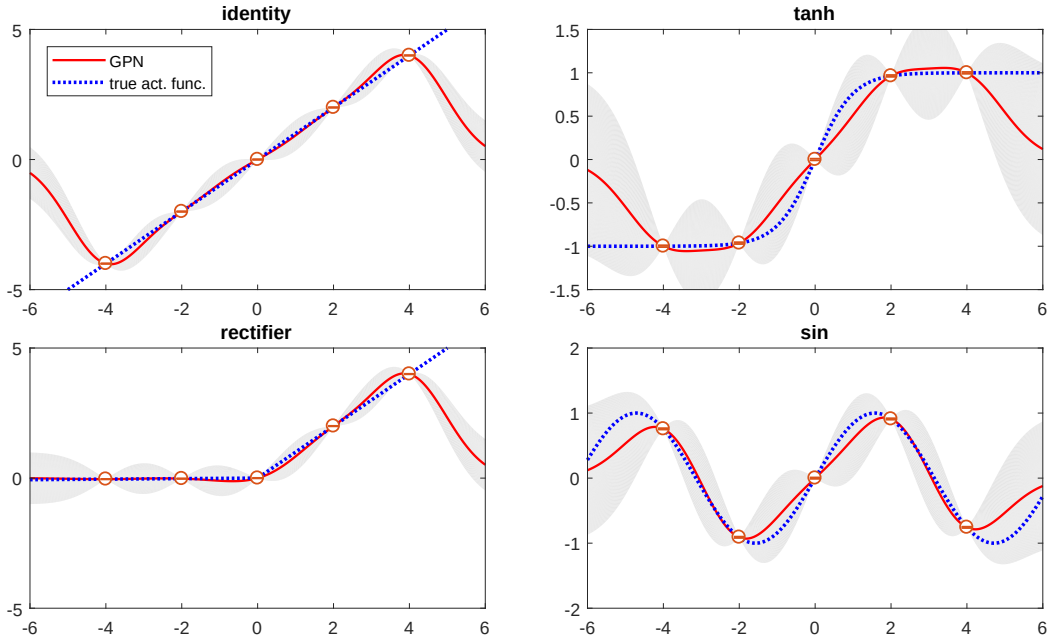
In conclusion, we have derived completely deterministic training objectives for a feed-forward parametric GPN network using analytic propagation of means and covariances from layer to layer, resulting in loss functions that can be minimized using the classic technique of backpropagation.

5.5.2 Computational and Model Complexity

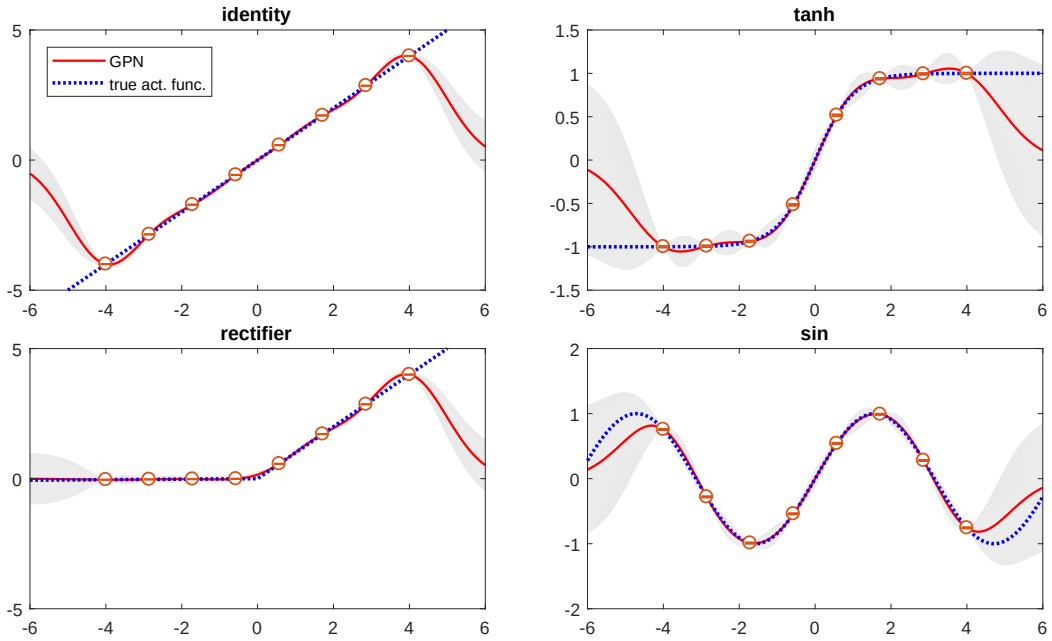
To represent the activation functions a parametric GPN layer requires $3R$ parameters per GPN, where R is the number of virtual observations. GPNs require no bias term, because it is equivalent to an offset in the targets corresponding to the inducing points. To reduce the number of parameters the locations of the inducing points V can be fixed, for example using equidistantly placed points, so that only $2R$ parameters are required. Furthermore, the standard deviations S of the targets U can be shared between all inducing points, resulting in only $R + 1$ parameters per GPN. Finally, to further reduce the number of required parameters we can apply the weight sharing idea from CNNs and use a common set of virtual observations and thus activation functions within a group of GPNs or even within a whole layer.

Since the flexibility of the parametric GPN is controlled by R, care must be taken not to choose a too small R since this would limit the activation functions representable by the GPN and thus the power of a feed-forward network constructed from these GPNs. A good heuristic for choosing R is so that the GPN is able to represent the most common activation functions currently in use. For that purpose, we empirically modeled the hyperbolic tangent, rectifier and sine activation functions with a GPN using a varying number of inducing points and compared the resulting approximation to the original function. From fig. 5.12 we can see that $R = 8$ virtual observations with equidistant inducing points are enough to represent these activation functions with high accuracy. However, outside of the range of the inducing points the approximation of the activation function will return to zero by design and cannot model a periodic function like the sine.

The computational complexities of propagating mean and covariance from layer to layer are shown in table 5.2. The complexity of calculating the responses can be significantly reduced from $\mathcal{O}(N_l\, R^3)$ to $\mathcal{O}(N_l\, R^2)$ by keeping the inducing points V^l and variances S^l fixed, because in



(a) GPN using 5 virtual observations



(b) GPN using 8 virtual observations

Figure 5.12: Common activation functions approximated by a parametric GPN with 5 or 8 virtual observations (red circles) respectively. The dotted line shows the actual activation functions, while the red line is the approximation by the parametric GPN. The standard deviation is shown using gray shading. Using 8 virtual observations all functions can be closely approximated within the range of the inducing points; outside this range the function values gradually return to zero.

expression			*complexity*	
activations given $\widetilde{\mu}_{s\star}^{X^{l-1}}$, $\widetilde{\Sigma}_{s\star}^{X^{l-1}}$				
mean	$\widetilde{\mu}_{s\star}^{A^l}$	(5.84a)	$\mathcal{O}(N_l\, N_{l-1})$	
variance	$\mathrm{diag}(\widetilde{\Sigma}_{s\star\star}^{A^l})$	(5.84b)	$\mathcal{O}(N_l\, N_{l-1})$	
covariance	$\widetilde{\Sigma}_{s\star\star}^{A^l}$	(5.84b)	$\mathcal{O}(N_l^2\, N_{l-1} + N_l\, N_{l-1}^2)$	
responses given $\widetilde{\mu}_{s\star}^{A^l}$, $\widetilde{\Sigma}_{s\star}^{A^l}$			*variable* V, S	*fixed* V, S
mean	$\widetilde{\mu}_{s\star}^{X^l}$	(5.87)	$\mathcal{O}(N_l\, R^3)$	$\mathcal{O}(N_l\, R^2)$
variance	$\mathrm{diag}(\widetilde{\Sigma}_{s\star\star}^{X^l})$	(5.101)	$\mathcal{O}(N_l\, R^3)$	$\mathcal{O}(N_l\, R^2)$
covariance	$\widetilde{\Sigma}_{s\star\star}^{X^l}$	(5.102)	$\mathcal{O}(N_l^2\, R^3)$	$\mathcal{O}(N_l^2\, R^2)$

Table 5.2: Computational complexity of propagating the mean and variance or covariance from layer $l-1$ to layer l. The exponent $\mathcal{O}(R^3)$ comes from the complexity of matrix multiplication and inversion. For matrix multiplication it can be reduced to $\mathcal{O}(R^{2.807})$ using the Strassen algorithm (Strassen, 1969).

this case the tensor κ^l given by (5.89) is fixed and can be precomputed. Another method to save computational complexity is to only propagate the variances, i.e. diagonal of the covariance matrix, through the GPN stack. This reduces the complexity of computing the activations from $\mathcal{O}(N_l^2\, N_{l-1} + N_l\, N_{l-1}^2)$ to $\mathcal{O}(N_l\, N_{l-1})$ and the complexity of computing the responses from $\mathcal{O}(N_l^2\, R^3)$ to $\mathcal{O}(N_l\, R^3)$.

Note, that the number of parameters and the computational complexity of propagating the means and covariances only depend on the number of virtual observations and parametric GPNs; therefore the memory requirement is *independent* of the number of training samples and the required training time per epoch scales *linearly* with the number of training samples. Thus, like a conventional neural network, a parametric GPN feed-forward can inherently be trained on datasets of unlimited size.

5.6 Approximate Bayesian Inference

In experiments with parametric GPNs it became apparent that the variances S_r of the virtual observations were driven to zero, thus leading to overfitting. This is to be expected since a parametric GPN feed-forward network is a model with more expressive power than a conventional neural network and maximum-likelihood solutions are usually prone to overfitting if no care is taken to limit the expressiveness of the model or ensure regularization by some other means. A method to avoid these problems is to perform approximate Bayesian inference by means of an variational approximation instead of maximum likelihood estimation. Bayesian inference requires a prior distribution on the parameters.

Thus we cannot directly apply it on the parametric GPN as introduced in section 5.3, since the inducing points V_r, targets U_r and variances S_r lack such a prior distribution. The original concept of the non-parametric GPN was to have a GP prior on the activation function of each unit, as given by (5.10); hence the natural choice for a prior on the virtual observations is such that the GP prior on the activation function is restored when the virtual observations are marginalized out. To our knowledge the first use of this prior-restoring technique was in finding inducing points for sparse GP regression (Titsias, 2009).

More formally, we want to find a prior $\mathrm{P}(V_\star, U_\star, S_\star)$ so that

$$\mathrm{P}(X_\star \mid A_\star) = \iiint \mathrm{P}(X_\star \mid A_\star, V_\star, U_\star, S_\star)\, \mathrm{P}(V_\star, U_\star, S_\star)\, \mathrm{d}V_\star\, \mathrm{d}U_\star\, \mathrm{d}S_\star$$

where $\mathrm{P}(X_\star \mid A_\star)$ is the GP prior given by (5.10) and $\mathrm{P}(X_\star \mid A_\star, V_\star, U_\star, S_\star)$ is the parametric GPN distribution given by (5.43). We remember that $S_\star$ was introduced to allow to control the precision and thus influence of the virtual observations when $V_\star$ and $U_\star$ were deterministic parameters. However, now that $V_\star$ and $U_\star$ have become random variables, it is redundant to have an explicit parameter $S_\star$ for the variance of the virtual observations and thus we remove it by setting $S_\star = \mathbf{0}$. Thus we are looking for a prior $\mathrm{P}(V_\star, U_\star)$ so that

$$\mathrm{P}(X_\star \mid A_\star) = \iint \mathrm{P}(X_\star \mid A_\star, V_\star, U_\star, S_\star = \mathbf{0})\, \mathrm{P}(V_\star, U_\star)\, \mathrm{d}V_\star\, \mathrm{d}U_\star\,.$$

From section 2.4 we know that the GP regression distribution $\mathrm{P}(X_\star \mid A_\star, V_\star, U_\star, S_\star)$ follows from observing a subset of variables, here $U_\star$, that share a common GP prior with $X_\star$. Thus $X_\star$ and $U_\star$ must follow a joint GP prior,

$$\mathrm{P}(X_\star, U_\star \mid A_\star, V_\star) = \mathcal{N}\left(\begin{bmatrix} X_\star \\ U_\star \end{bmatrix} \middle| \mathbf{0}, \begin{bmatrix} K(A_\star, A_\star) & K(A_\star, V_\star) \\ K(V_\star, A_\star) & K(A_\star, A_\star) \end{bmatrix} \right), \tag{5.114}$$

and using the marginalization property of the normal distribution we obtain

$$\mathrm{P}(X_\star \,|\, A_\star) = \mathcal{N}\big(X_\star \,|\, \mathbf{0}, K(A_\star, A_\star)\big)\,, \tag{5.115}$$

$$\mathrm{P}(U_\star \,|\, V_\star) = \mathcal{N}\big(U_\star \,|\, \mathbf{0}, K(V_\star, V_\star)\big)\,. \tag{5.116}$$

Consequently $\mathrm{P}(V_\star, U_\star) = \mathrm{P}(U_\star \,|\, V_\star)\,\mathrm{P}(V_\star)$, where $\mathrm{P}(V_\star)$ can be chosen freely, results in a GP prior on the activation function as desired. To avoid marginalization over $V_\star$ we keep that variable deterministic by choosing a delta distribution for its prior,

$$P(V_\star) = \delta(V_\star - v_\star)\,. \tag{5.117}$$

For clarity of notation we will also drop $V_\star$ from the conditioning set of probability distributions from now on.

The joint distribution of a parametric GPN feed-forward network with appropriate prior is thus given by

$$\mathrm{P}(\{X\}_1^L, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L \,|\, X^0) = \prod_{l=1}^{L} \mathrm{P}(A^l \,|\, X^{l-1})\,\mathrm{P}(U^l)\,\mathrm{P}(F^l \,|\, A^l, U^l)\,\mathrm{P}(X^l \,|\, F^l)\,, \tag{5.118}$$

where to notation $\{\bullet\}_1^L$ should be read as $\{\bullet^1, \bullet^2, \ldots, \bullet^L\}$. To keep notation brief we did not write out explicitly the dependencies of the occurring conditionals on samples and GPN units. Note that by marginalizing (5.118) over $\{U\}_1^L$ using the prior (5.116) we will obtain the distribution of a *non-parametric* GPN feed-forward network as given by (5.16). A graphical model corresponding to that distribution with training targets X^L observed is shown in fig. 5.13a.

As described in section 5.2.2 exact inference in this model requires the use of Monte Carlo methods that come with additional computational complexity. Instead, here we turn to the technique of variational inference, cf. section 2.2.12, to approximate the posterior of the model. Performing variational inference requires planning the approximation. It has to be decided which latent variables of the model should be inferred and which should be marginalized out. Also the structure of the approximative distribution Q has to be chosen; this entails the approximative distributions for the posteriors of the latent variable and assumed independence relations. We analyze three different approaches for performing variational inference in a GPN feed-forward network and present them in the following sections.

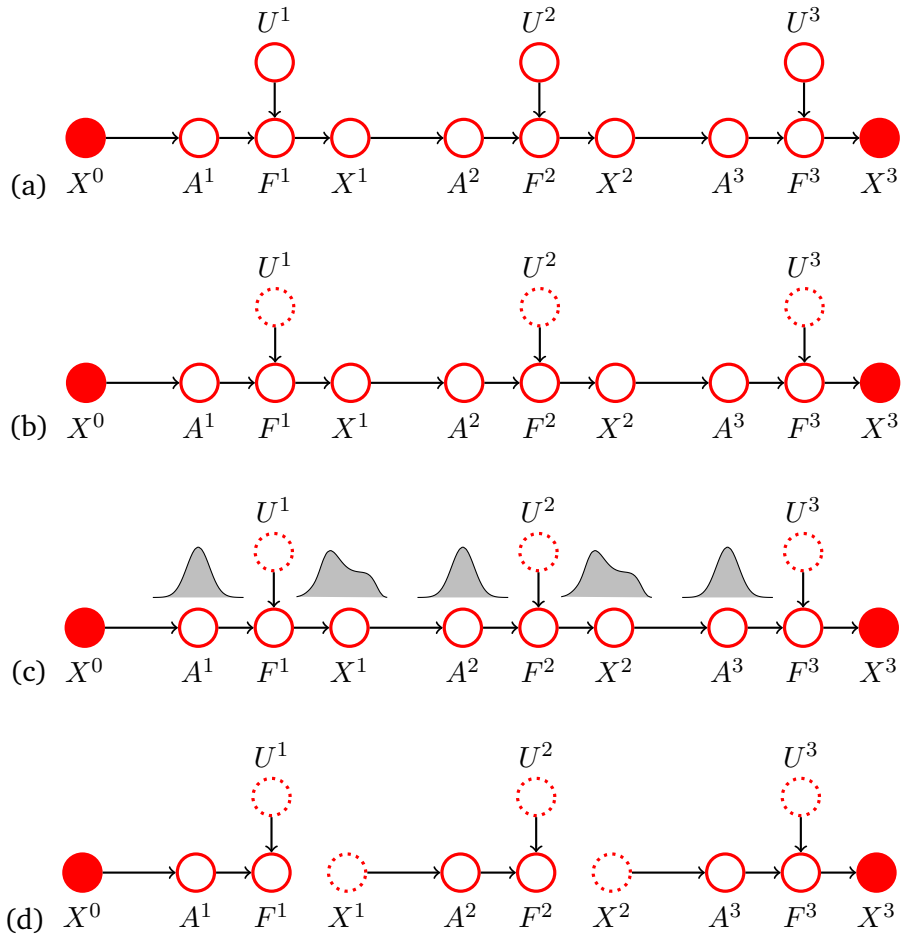


Figure 5.13: A GPN feed-forward network distribution for three layers and three approximations of its posterior. Each node corresponds to all samples and GPN units within a layer. (a) Exact posterior distribution $\mathrm{P}(\{X\}_1^{L-1}, \{A\}_1^L, \{F\}_1^L, \{U\}_1^L \mid X^0, X^L)$ that results in a *non-parametric* GPN feed-forward network when marginalized over $\{U\}_1^L$. (b) Variational approximation of the inducing targets U^l. The remaining conditional distributions of the approximative posterior are the same as the prior. (c) Variational approximation of the inducing targets U^l assuming that the central limit theorem holds for the marginals of the latent activations A^l. This is the case when there is a sufficient number of GPNs per layer. (d) Variational approximation of the inducing targets U^l and the latent values X^l factorizing over the layers (mean-field approach).

5.6.1 Stochastic Variational Inference

Since the information about the activation functions learned from the training data is mediated via the virtual observation targets U^l, their posterior must be adaptable in order to store that information. Hence, we choose a normal distribution factorized over the GPN units within a layer with free mean and covariance for the approximative posterior of U^l,

$$\mathrm{Q}(U^l) \triangleq \prod_{n=1}^{N_l} \mathrm{Q}(U^l_{\star n})\,, \qquad \mathrm{Q}(U^l_{\star n}) \triangleq \mathcal{N}(U^l_{\star n} \,|\, \widehat{\mu}^{U^l}_{\star n}, \widehat{\Sigma}^{U^l}_{\star\star n})\,. \tag{5.119}$$

This allows the inducing targets of a GPN to be correlated, but the covariance matrix can be constrained to be diagonal, if it is desired to reduce the number of model parameters. We keep the rest of the model distribution unchanged from the prior; thus the overall approximating posterior is given by

$$\mathrm{Q}(\{U\}_1^L, \{X\}_1^{L-1}, \{A\}_1^L, \{F\}_1^L) \triangleq \prod_{l=1}^{L} \mathrm{P}(A^l \,|\, X^{l-1})\, \mathrm{Q}(U^l)\, \mathrm{P}(F^l \,|\, A^l, U^l)\, \mathrm{P}(X^l \,|\, F^l)\,, \tag{5.120}$$

where the dependency on X^0 has been dropped from Q for clarity of notation. A graphical model corresponding to this approximative posterior is shown in fig. 5.13b.

The parameters $\widehat{\mu}^{U^l}$ and $\widehat{\Sigma}^{U^l}$ are estimated using the method of variational inference, cf. section 2.2.12, which does that by minimizing a lower bound on the KL-divergence between the approximation and true posterior,

$$\mathrm{KL}\big(\mathrm{Q}(\{U\}_1^L, \{X\}_1^{L-1}, \{A\}_1^L, \{F\}_1^L) \;||\; \mathrm{P}(\{X\}_1^{L-1}, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L \,|\, X^0, X^L)\big)\,.$$

This is performed implicitly by maximizing the ELBO given by

$$\mathcal{L} = -\int \cdots \int \mathrm{Q}(\{U\}_1^L, \{X\}_1^{L-1}, \{A\}_1^L, \{F\}_1^L)\, \log \frac{\mathrm{Q}(\{U\}_1^L, \{X\}_1^{L-1}, \{A\}_1^L, \{F\}_1^L)}{\mathrm{P}(\{U\}_1^L, \{X\}_1^L, \{A\}_1^L, \{F\}_1^L \,|\, X^0)}\cdot \\ \mathrm{d}\{U\}_1^L\, \mathrm{d}\{X\}_1^L\, \mathrm{d}\{A\}_1^L\, \mathrm{d}\{F\}_1^L \tag{5.121}$$

w.r.t. to the parameters of interest. Substituting the distributions into this equation results in

$$\mathcal{L} = -\mathcal{L}_{\mathrm{reg}} + \mathcal{L}_{\mathrm{pred}} \tag{5.122}$$

with the following terms after simplification,

$$\mathcal{L}_{\text{reg}} = \sum_{l=1}^{L} \int \mathrm{Q}(U^l) \, \log \frac{\mathrm{Q}(U^l)}{\mathrm{P}(U^l)} \, \mathrm{d}U^l \,, \tag{5.123}$$

$$\mathcal{L}_{\text{pred}} = \int \mathrm{Q}(F^L) \, \log \mathrm{P}(X^L \,|\, F^L) \, \mathrm{d}F^L \,. \tag{5.124}$$

The term $\mathcal{L}_{\text{reg}}$ can be identified as the sum of the KL-divergences of the virtual observation targets between their prior and variational posterior. Since this term enters $\mathcal{L}$ with a negative sign, its purpose is to keep the approximative posterior close to the prior; thus it can be understood as a regularization term. It is evaluated using the formula for the KL-divergence between two normal distributions (2.35). Disregarding an additive constant, its value is given by

$$\begin{aligned} \mathcal{L}_{\text{reg}} &= \sum_{l=1}^{L} \mathrm{KL}\big(\mathrm{Q}(U^l) \,||\, \mathrm{P}(U^l)\big) = \sum_{l=1}^{L} \sum_{n=1}^{N_l} \mathrm{KL}\big(\mathrm{Q}(U^l_{\star n}) \,||\, \mathrm{P}(U^l_{\star n})\big) \\ &\propto \frac{1}{2} \sum_{l=1}^{L} \sum_{n=1}^{N_l} \left(\mathrm{tr}\Big(K(V^l_{\star n}, V^l_{\star n})^{-1} \, \widehat{\Sigma}^{U^l}_{\star\star n}\Big) + (\widehat{\mu}^{U^l}_{\star n})^T K(V^l_{\star n}, V^l_{\star n})^{-1} \widehat{\mu}^{U^l}_{\star n} + \log \frac{\big|K(V^l_{\star n}, V^l_{\star n})\big|}{\big|\widehat{\Sigma}^{U^l}_{\star\star n}\big|} \right) . \end{aligned} \tag{5.125}$$

The term $\mathcal{L}_{\text{pred}}$ cannot be evaluated directly because the marginal $\mathrm{Q}(F^L)$ is intractable in general. However, expanding $\log \mathrm{P}(X^L \,|\, F^L)$ over the samples and writing $\mathrm{Q}(F^L)$ as a marginal over the layers, we note that $\mathcal{L}_{\text{pred}}$ can be written as an expectation,

$$\begin{aligned} \mathcal{L}_{\text{pred}} &= \mathrm{E}_{\mathrm{Q}(F^L)}\big[\log \mathrm{P}(X^L \,|\, F^L)\big] = \mathrm{E}_{\mathrm{Q}(F^L)}\left[\sum_{s=1}^{S} \log \mathrm{P}(X^L_{s\star} \,|\, F^L)\right] \\ &= \sum_{s=1}^{S} \mathrm{E}_{\mathrm{Q}(F^L_{s\star})}\big[\log \mathrm{P}(X^L_{s\star} \,|\, F^L_{s\star})\big] \\ &= \sum_{s=1}^{S} \mathrm{E}_{\mathrm{Q}(X^1_{s\star} \,|\, X^0_{s\star}) \, \mathrm{Q}(X^2_{s\star} \,|\, X^1_{s\star}) \cdots \mathrm{Q}(X^{L-1}_{s\star} \,|\, X^{L-2}_{s\star}) \, \mathrm{Q}(F^L_{s\star} \,|\, A^L_{s\star} = W^L X^{L-1}_{s\star})}\big[\log \mathrm{P}(X^L_{s\star} \,|\, F^L_{s\star})\big] \,, \end{aligned} \tag{5.126}$$

and thus we can evaluate it by successively sampling from the chain of conditional distributions the expectation is taken over. For this purpose we need to evaluate the conditional distributions

$$\mathrm{Q}(X^l \,|\, X^{l-1}) = \int \mathrm{Q}(F^l \,|\, A^l = W^l X^{l-1}) \, \mathrm{P}(X^l \,|\, F^l) \, \mathrm{d}F^l \,, \tag{5.127}$$

$$\mathrm{Q}(F^l \,|\, A^l) = \prod_{n=1}^{N_l} \int \mathrm{Q}(U^l_{\star n}) \, \mathrm{P}(F^l_{\star n} \,|\, A^l_{\star n}, U^l_{\star n}) \, \mathrm{d}U^l_{\star n} \,. \tag{5.128}$$

Since $\mathrm{Q}(F^l \,|\, A^l)$ is the conditional of a GP with normally distributed observations, the joint distribution $\mathrm{Q}(F^l_{\star n}, U^l_{\star n} \,|\, A^l_{\star n}) = \mathrm{Q}(U^l_{\star n})\,\mathrm{P}(F^l_{\star n} \,|\, A^l_{\star n}, U^l_{\star n})$ must itself be normal,

$$\mathrm{Q}(F^l_{\star n}, U^l_{\star n} \,|\, A^l_{\star n}) = \mathcal{N}\left(\begin{bmatrix} F^l_{\star n} \\ U^l_{\star n} \end{bmatrix} \,\middle|\, \begin{bmatrix} \widehat{\mu}^{F^l}_{\star n} \\ \widehat{\mu}^{U^l}_{\star n} \end{bmatrix}, \begin{bmatrix} \widehat{\Sigma}^{F^l}_{\star\star n} & \widetilde{\Sigma}_{FU} \\ \widetilde{\Sigma}_{UF} & \widehat{\Sigma}^{U^l}_{\star\star n} \end{bmatrix}\right), \tag{5.129}$$

and we can find the values for the unknown parameters $\widehat{\mu}^{F^l}_{\star n}$, $\widehat{\Sigma}^{F^l}_{\star\star n}$ and $\widetilde{\Sigma}_{FU} = \widetilde{\Sigma}^T_{UF}$ by equating the moments of its conditional distribution $\mathrm{Q}(F^l_{\star n} \,|\, U^l_{\star n}, A^l_{\star n})$ with $\mathrm{P}(F^l_{\star n} \,|\, U^l_{\star n}, A^l_{\star n})$. The conditional distribution is given by

$$\mathrm{Q}(F^l_{\star n} \,|\, U^l_{\star n}, A^l_{\star n}) = \mathcal{N}(F^l_{\star n} \,|\, \widetilde{\mu}, \widetilde{\Sigma})$$

and thus we obtain the following set of equations by comparing mean and covariances,

$$\begin{aligned} \widetilde{\mu} &\triangleq \widehat{\mu}^{F^l}_{\star n} + \widetilde{\Sigma}_{FU}\,(\widehat{\Sigma}^{U^l}_{\star\star n})^{-1}\,(U^l_{\star n} - \widehat{\mu}^{U^l}_{\star n}) = K(A^l_{\star n}, V^l_{\star n})\,K(V^l_{\star n}, V^l_{\star n})^{-1}\,U^l_{\star n}\,, \\ \widetilde{\Sigma} &\triangleq \widehat{\Sigma}^{F^l}_{\star\star n} - \widetilde{\Sigma}_{FU}\,(\widehat{\Sigma}^{U^l}_{\star\star n})^{-1}\,\widetilde{\Sigma}_{UF} = K(A^l_{\star n}, A^l_{\star n}) - K(A^l_{\star n}, V^l_{\star n})\,K(V^l_{\star n}, V^l_{\star n})^{-1}\,K(V^l_{\star n}, A^l_{\star n})\,, \end{aligned} \tag{5.130}$$

where the right sides are obtained from $\mu^{F^l}_{\star n}$, $\Sigma^{F^l}_{\star\star n}$ given by (5.41). Solving for the three unknowns[5] gives

$$\widehat{\mu}^{F^l}_{\star n} = K(A^l_{\star n}, V^l_{\star n})\,K(V^l_{\star n}, V^l_{\star n})^{-1}\,\widehat{\mu}^{U^l}_{\star n} \tag{5.131a}$$

$$\widehat{\Sigma}^{F^l}_{\star\star n} = K(A^l_{\star n}, A^l_{\star n}) - K(A^l_{\star n}, V^l_{\star n})\,\widehat{K}^{U^l}_{\star\star n}\,K(V^l_{\star n}, A^l_{\star n}) \tag{5.131b}$$

$$\widetilde{\Sigma}_{FU} = K(A^l_{\star n}, V^l_{\star n})\,K(V^l_{\star n}, V^l_{\star n})^{-1}\,\widehat{\Sigma}^{U^l}_{\star\star n} = (\widetilde{\Sigma}_{UF})^T \tag{5.131c}$$

with

$$\widehat{K}^{U^l}_{\star\star n} \triangleq K(V^l_{\star n}, V^l_{\star n})^{-1} - K(V^l_{\star n}, V^l_{\star n})^{-1}\,\widehat{\Sigma}^{U^l}_{\star\star n}\,K(V^l_{\star n}, V^l_{\star n})^{-1}\,. \tag{5.132}$$

Finally, we obtain for the sought-after marginal of F^l given A^l

$$\mathrm{Q}(F^l \,|\, A^l) = \prod_{n=1}^{N^l} \mathcal{N}(F^l_{\star n} \,|\, \widehat{\mu}^{F^l}_{\star n}, \widehat{\Sigma}^{F^l}_{\star\star n})\,. \tag{5.133}$$

At this point it is instructive to verify that the obtained mean and covariance are consistent with the deterministic case and with the GP prior. For deterministic observations, that is $\widehat{\Sigma}^{U^l}_{\star\star n} = 0$, we obtain $\widehat{K}^{U^l}_{\star\star n} = K(V^l_{\star n}, V^l_{\star n})^{-1}$ and thus recover the standard GP regression distribution as expected. If U^l follows its prior, that is $\widehat{\mu}^{U^l}_{\star n} = \mathbf{0}$ and $\widehat{\Sigma}^{U^l}_{\star\star n} = K(V^l_{\star n}, V^l_{\star n})$, we obtain $\widehat{K}^{U^l}_{\star\star n} = 0$

[5] The first line of the system contains two equations, since it must be valid for all values of $U^l_{\star n}$. Thus a unique solution exists for three unknowns.

and thus recover the GP prior on F^l. In that case the virtual observations behave as if they were not present.

Having $\mathrm{Q}(F^l \mid A^l)$ immediately allows us to evaluate (5.127) since $\mathrm{P}(X^l \mid F^l)$ just provides additive Gaussian noise. Thus we obtain

$$\mathrm{Q}(X^l \mid X^{l-1}) = \prod_{n=1}^{N^l} \mathcal{N}(F^l_{\star n} \mid \widehat{\mu}^{F^l}_{\star n}, \widehat{\Sigma}^{F^l}_{\star\star n} + (\sigma^l_n)^2 \mathbb{1}) \,. \tag{5.134}$$

As we can see all conditional distributions are multivariate normals and thus sampling is straightforward. In fact, it can be performed in the same way as described in section 5.3.4; only the means and covariances must be adapted due to the fact that U^l is now governed by a distribution. Training is then done by reparameterizing the distributions as described in that section and performing gradient descent using the stochastic derivatives of $\mathcal{L}$ w.r.t. the variational parameters $\widehat{\mu}^{U^l}$, $\widehat{\Sigma}^{U^l}$ and model parameters W^l, $\boldsymbol{\sigma}^l$. As before mini-batch training can be used since $\mathcal{L}_{\text{pred}}$ factorizes over the samples.

5.6.2 Variational Inference using a Marginalized Posterior Distribution

The drawback of stochastic training procedure is the increase in the number of required training iterations due to the noise that is introduced into the model by the sampling procedure. As described in section 5.5, the activations A^l will closely resemble a normal distribution due to the central limit theorem if there is a sufficient number of GPNs per layer and the weights W^l have a sufficiently random distribution. In this case sampling becomes unnecessary, as the moments of $\mathrm{P}(A^l)$ can be calculated exactly and propagated from layer to layer as described previously. Here we use the same approximative posterior as in (5.120) with the additional assumption that each *marginal* $\mathrm{Q}(A^l)$ for $l \in \{1, \ldots, L\}$ is indistinguishable from a normal distribution with appropriate mean and covariance, i.e.

$$\mathrm{Q}(A^l) = \prod_{s=1}^{S} \mathrm{Q}(A^l_{s\star}) \,, \qquad \mathrm{Q}(A^l_{s\star}) = \mathcal{N}\left(A^l_{s\star} \,\middle|\, \mu^{A^l}_{s\star}, \Sigma^{A^l}_{s\star\star}\right) . \tag{5.135}$$

A graphical model corresponding to this approximate posterior is shown in fig. 5.13c.

Writing $\mathcal{L}_{\text{pred}}$ from (5.124) as

$$\mathcal{L}_{\text{pred}} = \iiint \mathrm{Q}(A^L)\, \mathrm{Q}(U^L)\, \mathrm{P}(F^L \mid A^L, U^L)\, \log \mathrm{P}(X^L \mid F^L)\, \mathrm{d}A^L\, \mathrm{d}U^L\, \mathrm{d}F^L \,. \tag{5.136}$$

shows that we first need to obtain the distribution $\mathrm{Q}(A^L)$. This is done by iteratively calculating the marginals $\mathrm{Q}(A^l)$ for $l \in \{1, \ldots, L\}$ in a similar way as it was done in section 5.5.1 for fixed

values of U^l. For $l \geq 1$ we have

$$\mathrm{Q}(A^{l+1}) = \iiint \mathrm{Q}(A^l)\,\mathrm{Q}(F^l \,|\, A^l)\,\mathrm{P}(X^l \,|\, F^l)\,\mathrm{P}(A^{l+1} \,|\, X^l)\,\mathrm{d}A^l\,\mathrm{d}F^l\,\mathrm{d}X^l \tag{5.137}$$

where $\mathrm{Q}(F^l \,|\, A^l)$ is given by (5.133). We first evaluate the mean and covariance of the marginal $\mathrm{Q}(F^l)$ by following the course of action in section 5.5.1. For the mean of the response $\widetilde{\mu}_{sn}^{F^L} = \mathrm{E}_{\mathrm{Q}(F^L)}\big[F_{sn}^l\big]$ we obtain

$$\begin{aligned}\widetilde{\mu}_{sn}^{F^l} &= \mathrm{E}_{\mathrm{Q}(A_{s\star}^l)}\Big[K(A_{sn}^l, V_{\star n}^l)\Big]^T\, K(V_{\star n}^l, V_{\star n}^l)^{-1}\,\widehat{\mu}_{\star n}^{U^l} \\ &= (\psi_{\star n}^l)^T\, K(V_{\star n}^l, V_{\star n}^l)^{-1}\,\widehat{\mu}_{\star n}^{U^l}\end{aligned} \tag{5.138}$$

with ψ_{rn}^l as previously calculated in (5.94). Similarly for the response covariance matrix $\widetilde{\Sigma}_{snm}^{F^L} = \mathrm{Cov}_{\mathrm{Q}(F^L)}\big(F_{sn}^l, F_{sm}^l\big)$, we obtain that the diagonal is given by

$$\widetilde{\Sigma}_{snn}^{F^l} = 1 - \mathrm{tr}\Big[\Big(\widehat{K}_{\star\star n}^{U^l} - \beta_{\star n}^l\,(\beta_{\star n}^l)^T\Big)\,\Omega_{\star\star n}^l\Big] - \mathrm{tr}\Big(\psi_{\star n}^l(\psi_{\star n}^l)^T\,\beta_{\star n}^l(\beta_{\star n}^l)^T\Big) \tag{5.139}$$

with $\beta_{\star n}^l \triangleq K(V_{\star n}^l, V_{\star n}^l)^{-1}\,\widehat{\mu}_{\star n}^{U^l}$ and Ω_{rtn}^l from (5.99c). The off-diagonal elements of the covariance matrix evaluate to

$$\widetilde{\Sigma}_{snm}^{F^l} = \sum_r \sum_t \beta_{rn}^l \beta_{tm}^l\,\Lambda_{rtnm}^l - \widetilde{\mu}_{sn}^{F^l}\,\widetilde{\mu}_{sm}^{F^l} \tag{5.140}$$

with Λ_{rtnm}^l given by (5.105). Finally, assuming that the central limit theorem holds, the marginal distribution of the approximative posterior of A^{l+1} is given by

$$\mathrm{Q}(A^{l+1}) = \mathcal{N}(A_{s\star}^{l+1} \,|\, \widetilde{\mu}_{s\star}^{A^{l+1}}, \widetilde{\Sigma}_{s\star\star}^{A^{l+1}}) \tag{5.141}$$

with

$$\widetilde{\mu}_{s\star}^{A^{l+1}} = W^{l+1}\,\widetilde{\mu}_{s\star}^{F^l} \tag{5.142a}$$

$$\widetilde{\Sigma}_{s\star\star}^{A^{l+1}} = W^{l+1}\,\Big(\widetilde{\Sigma}_{s\star\star}^{F^l} + \mathrm{diag}(\sigma^l)^2\Big)\,(W^{l+1})^T\,. \tag{5.142b}$$

The term $\mathcal{L}_{\mathrm{pred}}$, given by (5.124), can be identified as the expected log-probability of the

observations under the marginal distribution $Q(F^L)$ and thus we can expand it as follows,

$$\begin{aligned}\mathcal{L}_{\text{pred}} &= \mathrm{E}_{\mathrm{Q}(F^L)}\big[\log \mathrm{P}(X^L \mid F^L)\big] \\ &\propto \mathrm{E}_{\mathrm{Q}(F^L)}\left[-S\sum_{n=1}^{N_l}\log\sigma_n^L - \frac{1}{2}\sum_{s=1}^{S}\sum_{n=1}^{N_L}\frac{\big(X_{sn}^L - F_{sn}^L\big)^2}{(\sigma_n^L)^2}\right], \\ &= -S\sum_{n=1}^{N_L}\log\sigma_n^L - \frac{1}{2}\sum_{s=1}^{S}\sum_{n=1}^{N_L}\frac{(X_{sn}^L)^2 - 2\,X_{sn}^L\,\mathrm{E}_{\mathrm{Q}(F^L)}\big[F_{sn}^L\big] + \mathrm{E}_{\mathrm{Q}(F^L)}\big[(F_{sn}^L)^2\big]}{(\sigma_n^L)^2}. \end{aligned} \tag{5.143}$$

The distribution $Q(F^L)$ itself is of arbitrary form, but as it can be seen from the above equation, only its first and second moments are required to evaluate $\mathcal{L}_{\text{pred}}$. For the first moment we obtain $\mathrm{E}_{\mathrm{Q}(F^L)}\big[F_{sn}^l\big] = \widetilde{\mu}_{sn}^{F^L}$ with $\widetilde{\mu}_{sn}^{F^L}$ given by (5.138) and the second moment evaluates to

$$\begin{aligned}\mathrm{E}_{\mathrm{Q}(F^L)}\big[(F_{sn}^L)^2\big] &= \mathrm{Var}_{\mathrm{Q}(F^L)}\big(F_{sn}^L\big) + \mathrm{E}_{\mathrm{Q}(F^L)}\big[F_{sn}^L\big]^2 = \widehat{\Sigma}_{snn}^{F^L} + \big(\widehat{\mu}_{snn}^{F^L}\big)^2 \\ &= 1 - \mathrm{tr}\big[\big(\kappa_{\star\star n}^L - \beta_{\star n}^L(\beta_{\star n}^L)^T\big)\,\Omega_{\star\star n}^L\big], \end{aligned} \tag{5.144}$$

with $\beta_{\star n}^L \triangleq K(V_{\star n}^L, V_{\star n}^L)^{-1}\,\widehat{\mu}_{\star n}^{U^L}$ and Ω_{rtn}^L from (5.99c).

This concludes the calculation of all terms of the variational lower bound (5.121). As with the normal approximation derived in section 5.5, the resulting objective is a fully deterministic function of the parameters. Training of the model is performed by maximizing $\mathcal{L} = -\mathcal{L}_{\text{reg}} + \mathcal{L}_{\text{pred}}$, with $\mathcal{L}_{\text{reg}}$ given by (5.125) and $\mathcal{L}_{\text{pred}}$ given by (5.143), w.r.t. to the variational parameters $\widehat{\mu}^{U^l}$, $\widehat{\Sigma}^{U^l}$ and the model parameters $\boldsymbol{\sigma}^l$, W^l. This can be performed using any gradient-descent based algorithm in a mini-batch training routine. As before, the necessary derivatives are not derived here and it is assumed that this can be performed automatically using symbolic or automatic differentiation in an appropriate framework.

5.6.3 Variational Inference using a Mean-Field Posterior Approximation

In the context of deep GPs it has been proposed (Damianou et al., 2013) to factorize the approximative posterior over the layer values, which in our case correspond to the variables X^l, $l \in \{1, \ldots, L\}$. We can use the same technique to perform approximative inference when the assumption of a normal distribution over the activations is not fulfilled, for example in the case of a feed-forward network that only has a few GPNs per layer. The approximative posterior for

this approach is given by

$$\mathrm{Q}(\{X\}_1^{L-1}, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L) = \left(\prod_{l=1}^{L} \mathrm{P}(A^l \mid X^{l-1})\, \mathrm{Q}(U^l)\, \mathrm{P}(F^l \mid A^l, U^l)\right) \left(\prod_{l=1}^{L-1} \mathrm{Q}(X^l)\right). \tag{5.145}$$

The conditional distribution $\mathrm{P}(F^l \mid A^l, U^l)$ coming from the GP regression has been inherited from the prior unchanged; thus, the approximative posterior must carry all information from the training data in $\mathrm{Q}(U^l)$ and $\mathrm{Q}(X^l)$. The approximative distribution over the layer outputs $\mathrm{Q}(X^l)$ is unconditional; this leads to the approximative posterior being fully factorized *over the layers* and inference is performed using a mean-field of X^l. For the virtual observation targets U^l we use the same form as in the non-factorized approximation, i.e. a multivariate normal distribution with free mean and covariance,

$$\mathrm{Q}(U^l) = \prod_{n=1}^{N_l} \mathrm{Q}(U^l_{\star n}), \qquad \mathrm{Q}(U^l_{\star n}) = \mathcal{N}(U^l_{\star n} \mid \widehat{\mu}^{U^l}_{\star n}, \widehat{\Sigma}^{U^l}_{\star\star n}). \tag{5.146}$$

As before the virtual observations are factorized over the GPNs, but may be correlated within a GPN. For the latent layer values X^l a normal distribution factorized over the samples with free mean and covariances is used,

$$\mathrm{Q}(X^l) = \prod_{s=1}^{S} \mathrm{Q}(X^l_{s\star}), \qquad \mathrm{Q}(X^l_{s\star}) = \mathcal{N}(X^l_{s\star} \mid \widehat{\mu}^{X^l}_{s\star}, \widehat{\Sigma}^{X^l}_{s\star\star}). \tag{5.147}$$

This allows for covariance between different GPN units within a layer. As the occurring covariance matrix grows quadratically with the number of GPNs within a layer, $\Sigma^{X^l}_{s\star\star}$ should be constrained to a diagonal matrix to limit the associated computational complexity. Note that in contrast to the non-factorizing approach, here we assume a normal distribution for the approximative posterior $\mathrm{Q}(X^l)$ and not for the marginals $\mathrm{P}(A^l)$ of the original model. A graphical model corresponding to this approximation is shown in fig. 5.13d.

The KL-divergence between the true posterior and the approximation Q is minimized implicitly by maximizing the ELBO given by

$$\mathcal{L} = -\int \cdots \int \mathrm{Q}(\{X\}_1^{L-1}, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L) \log \frac{\mathrm{Q}(\{X\}_1^{L-1}, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L)}{\mathrm{P}(\{X\}_1^L, \{A\}_1^L, \{U\}_1^L, \{F\}_1^L \mid X^0)} \cdot \mathrm{d}\{X\}_1^{L-1}\, \mathrm{d}\{A\}_1^L\, \mathrm{d}\{U\}_1^L\, \mathrm{d}\{F\}_1^L. \tag{5.148}$$

Substituting the distributions into that equation gives

$$\mathcal{L} = -\mathcal{L}_{\text{reg}} - \mathcal{L}_{\text{prop}} + \mathcal{L}_{\text{pred}} \tag{5.149}$$

with

$$\mathcal{L}_{\text{reg}} = \sum_{l=1}^{L} \int \mathrm{Q}(U^l) \, \log \frac{\mathrm{Q}(U^l)}{\mathrm{P}(U^l)} \, \mathrm{d}U^l \tag{5.150}$$

$$\mathcal{L}_{\text{prop}} = \sum_{l=1}^{L-1} \iiiint \mathrm{Q}(X^{l-1}) \, \mathrm{P}(A^l \,|\, X^{l-1}) \, \mathrm{Q}(U^l) \, \mathrm{P}(F^l \,|\, A^l, U^l) \, \mathrm{Q}(X^l) \, \log \frac{\mathrm{Q}(X^l)}{\mathrm{P}(X^l \,|\, F^l)} \cdot \mathrm{d}A^l \, \mathrm{d}U^l \, \mathrm{d}F^l \, \mathrm{d}X^{l-1} \, \mathrm{d}X^l \tag{5.151}$$

$$\mathcal{L}_{\text{pred}} = \iiiint \mathrm{Q}(X^{L-1}) \, \mathrm{P}(A^L \,|\, X^{L-1}) \, \mathrm{Q}(U^L) \, \mathrm{P}(F^L \,|\, A^L, U^L) \, \log \mathrm{P}(X^L \,|\, F^L) \cdot \mathrm{d}X^{L-1} \, \mathrm{d}A^L \, \mathrm{d}U^L \, \mathrm{d}F^L \, . \tag{5.152}$$

The term $\mathcal{L}_{\text{reg}}$ is unchanged from the stochastic and marginal inference approaches and as before it measures the difference between the prior and posterior distribution of the virtual observation targets U^l. Its value is given by (5.125). The term $\mathcal{L}_{\text{prop}}$ ensures the propagation of values from layer to layer and can be interpreted as the expectation of the KL-divergences between exact and approximative latent layer distributions,

$$\mathcal{L}_{\text{prop}} = \sum_{l=1}^{L} \mathrm{E}_{\mathrm{Q}(F^l)} \Big[\mathrm{KL} \Big(\mathrm{Q}(X^l) \,||\, \mathrm{P}(X^l \,|\, F^l) \Big) \Big] = \sum_{l=1}^{L} \mathrm{E}_{\mathrm{Q}(F^l)} [\mathcal{L}_\beta] \, , \tag{5.153}$$

where

$$\mathrm{Q}(F^l) = \iiint \mathrm{Q}(X^{l-1}) \, \mathrm{P}(A^l \,|\, X^{l-1}) \, \mathrm{Q}(U^l) \, \mathrm{P}(F^l \,|\, A^l, U^l) \, \mathrm{d}X^{l-1} \, \mathrm{d}A^l \, \mathrm{d}U^l \, . \tag{5.154}$$

Since the approximative posterior was chosen to factorize over the layers, we obtain a sum over the layers for $\mathcal{L}_{\text{prop}}$ and no calculation of (intractable) marginals is necessary. The occurring

KL-divergence $\mathcal{L}_\beta$ is between two normals and thus straightforward to calculate,

$$\begin{aligned}\mathcal{L}_\beta &= \mathrm{KL}\big(\mathrm{Q}(X^l)\,||\,\mathrm{P}(X^l \mid F^l)\big) = \sum_{s=1}^{S} \mathrm{KL}\big(\mathrm{Q}(X^l_{s\star})\,||\,\mathrm{P}(X^l_{s\star} \mid F^l_{s\star})\big) \\ &\propto \frac{1}{2}\sum_{s=1}^{S}\left(\mathrm{tr}\big(\mathrm{diag}(\sigma_\star^l)^{-2}\widehat{\Sigma}^{X^l}_{s\star\star}\big) + (F^l_{s\star} - \widehat{\mu}^{X^l}_{s\star})^T \mathrm{diag}(\sigma^l_\star)^{-2}(F^l_{s\star} - \widehat{\mu}^{X^l}_{s\star}) + \log\frac{\big|\mathrm{diag}(\sigma^l_\star)^2\big|}{\big|\widehat{\Sigma}^{X^l}_{s\star\star}\big|}\right) \\ &= \frac{1}{2}\sum_{s=1}^{S}\left[\sum_{n=1}^{N_l}\left(\frac{\widehat{\Sigma}^{X^l}_{snn} + \big(F^l_{sn} - \widehat{\mu}^{X^l}_{sn}\big)^2}{(\sigma^l_n)^2} + 2\log\sigma^l_n\right) - \log\big|\widehat{\Sigma}^{X^l}_{s\star\star}\big|\right]. \end{aligned} \tag{5.155}$$

Thus we obtain for $\mathcal{L}_{\mathrm{prop}}$,

$$\begin{aligned}\mathcal{L}_{\mathrm{prop}} = \frac{1}{2}\sum_{l=1}^{L}\sum_{s=1}^{S}\left[\sum_{n=1}^{N_l}\left(\frac{\widehat{\Sigma}^{X^l}_{snn} + \mathrm{E}_{\mathrm{Q}(F^l)}\big[(F^l_{sn})^2\big] - 2\,\widehat{\mu}^{X^l}_{sn}\,\mathrm{E}_{\mathrm{Q}(F^l)}\big[F^l_{sn}\big] + (\widehat{\mu}^{X^l}_{sn})^2}{(\sigma^l_n)^2} + 2\log\sigma^l_n\right) - \\ \log\big|\widehat{\Sigma}^{X^l}_{s\star\star}\big|\right].\end{aligned} \tag{5.156}$$

In order to calculate the expectations $\mathrm{E}_{\mathrm{Q}(F^l)}\big[F^l_{sn}\big]$ and $\mathrm{E}_{\mathrm{Q}(F^l)}\big[(F^l_{sn})^2\big]$ we need the distribution $\mathrm{Q}(F^l)$. This distribution cannot be calculated in closed form; however, by applying the law of total expectation we can transform the expectations according to

$$\mathrm{E}_{\mathrm{Q}(F^l)}[\bullet] = E_{\mathrm{Q}(A^l)}\Big[E_{\mathrm{Q}(F^l \mid A^l)}[\bullet]\Big]\,,$$

where $\mathrm{Q}(A^l)$ evaluates to

$$\mathrm{Q}(A^l) = \int \mathrm{Q}(X^{l-1})\,\mathrm{P}(A^l \mid X^{l-1})\,\mathrm{d}X^{l-1} = \prod_{s=1}^{S}\mathcal{N}\big(A^l_{s\star} \mid \widetilde{\mu}^{A^l}_{s\star}, \widetilde{\Sigma}^{A^l}_{s\star\star}\big) \tag{5.157}$$

with

$$\widetilde{\mu}^{A^l}_{s\star} \triangleq W^l\,\widehat{\mu}^{X^{l-1}}_{s\star}\,, \qquad \widetilde{\Sigma}^{A^l}_{s\star\star} \triangleq W^l\,\widehat{\Sigma}^{X^{l-1}}_{s\star\star}\,(W^l)^T\,. \tag{5.158}$$

This allows us to evaluate the expectations in a very similar way as we did in section 5.5.1, the main difference being that U^l is now a random variable. We note that $\mathrm{Q}(F^l \mid A^l)$, defined by

$$\mathrm{Q}(F^l \mid A^l) = \prod_{n=1}^{N_l}\int \mathrm{Q}(U^l_{\star n})\,\mathrm{P}(F^l_{\star n} \mid A^l_{\star n}, U^l_{\star n})\,\mathrm{d}U^l_{\star n}\,,$$

has the same form as (5.133) from the stochastic inference approach and thus the marginal

can be computed in the same way as done in section 5.6.2, yielding

$$\mathrm{Q}(F^l \mid A^l) = \prod_{n=1}^{N^l} \mathcal{N}(F^l_{\star n} \mid \widehat{\mu}^{F^l}_{\star n}, \widehat{\Sigma}^{F^l}_{\star\star n}) \,. \tag{5.159}$$

with $\widehat{\mu}^{F^l}_{\star n}$ and $\widehat{\Sigma}^{F^l}_{\star\star n}$ given by (5.131).

Continuing with the calculation of the expectations in (5.156) gives the following results for the first moment,

$$\begin{aligned} \mathrm{E}_{\mathrm{Q}(F^l)}[F^l_{sn}] &= \mathrm{E}_{\mathrm{Q}(A^l)}\Big[\mathrm{E}_{\mathrm{Q}(F^l \mid A^l)}[F^l_{sn}]\Big] = \mathrm{E}_{\mathrm{Q}(A^l)}\Big[\widehat{\mu}^{F^l}_{sn}\Big] \\ &= \psi^l_{\star n}\, K(V^l_{\star n}, V^l_{\star n})^{-1}\, \widehat{\mu}^{U^l}_{\star n} \,, \end{aligned} \tag{5.160}$$

with ψ^l_{rn} given by (5.94) using $\widetilde{\mu}^{A^l}_{s\star}$ and $\widetilde{\Sigma}^{A^l}_{s\star\star}$ from (5.158). For the second moment we obtain

$$\begin{aligned} \mathrm{E}_{\mathrm{Q}(F^l)}[(F^l_{sn})^2] &= \mathrm{E}_{\mathrm{Q}(A^l)}\Big[\mathrm{E}_{\mathrm{Q}(F^l \mid A^l)}[(F^l_{sn})^2]\Big] = \mathrm{E}_{\mathrm{Q}(A^l)}\Big[\widehat{\Sigma}^{F^l}_{snn}\Big] + \mathrm{E}_{\mathrm{Q}(A^l)}\Big[\big(\widehat{\mu}^{F^l}_{snn}\big)^2\Big] \\ &= 1 - \mathrm{tr}\Big[\Big(\kappa^l_{\star\star n} - \beta^l_{\star n}(\beta^l_{\star n})^T\Big)\, \Omega^l_{\star\star n}\Big] \,, \end{aligned} \tag{5.161}$$

where $\kappa^l_{\star\star n} \triangleq [K(V^l_{\star n}, V^l_{\star n})]^{-1}$ and $\beta^l_{\star n} \triangleq \kappa^l_{\star\star n}\, U^l_{\star n}$ and Ω^l_{rtn} is given by (5.99c) using $\widetilde{\mu}^{A^l}_{s\star}$ and $\widetilde{\Sigma}^{A^l}_{s\star\star}$ from (5.158) as above. This concludes the calculation of $\mathcal{L}_{\mathrm{prop}}$.

The term $\mathcal{L}_{\mathrm{pred}}$, given by (5.152), is the expected log-probability of the observations under the approximative distribution. As the latter factorizes over the layers, the expectation only needs to be calculated over the top-most layer L. This results in

$$\begin{aligned} \mathcal{L}_{\mathrm{pred}} &= \mathrm{E}_{\mathrm{Q}(F^L)}\big[\log \mathrm{P}(X^L \mid F^L)\big] \\ &\propto -S \sum_{n=1}^{N_L} \log \sigma^L_n - \frac{1}{2} \sum_{s=1}^{S} \sum_{n=1}^{N_L} \frac{(X^L_{sn})^2 - 2\, X^L_{sn}\, \mathrm{E}_{\mathrm{Q}(F^L)}\big[F^L_{sn}\big] + \mathrm{E}_{\mathrm{Q}(F^L)}\big[(F^L_{sn})^2\big]}{(\sigma^L_n)^2} \end{aligned} \tag{5.162}$$

where $\mathrm{E}_{\mathrm{Q}(F^L)}\big[F^L_{sn}\big]$ and $\mathrm{E}_{\mathrm{Q}(F^L)}\big[(F^L_{sn})^2\big]$ are given by (5.160) and (5.161) respectively.

This concludes the calculation of all terms of the mean-field variational lower bound (5.149). Training of the model is performed by maximizing $\mathcal{L}$ w.r.t. to the variational parameters and the model parameters. In contrast to the non-factorizing approach presented in the previous section, the layer means $\widehat{\mu}^{X^l}$ and covariances $\widehat{\Sigma}^{X^l}$ are now part of the variational parameters and we must include them in the optimization of $\mathcal{L}$. However, they do not enter the predictions, which will be calculated from the approximative posterior of the inducing points U^l and weights W^l. Thus $\widehat{\mu}^{X^l}$ and $\widehat{\Sigma}^{X^l}$ can be discarded after training.

5.6.4 Comparison and Discussion

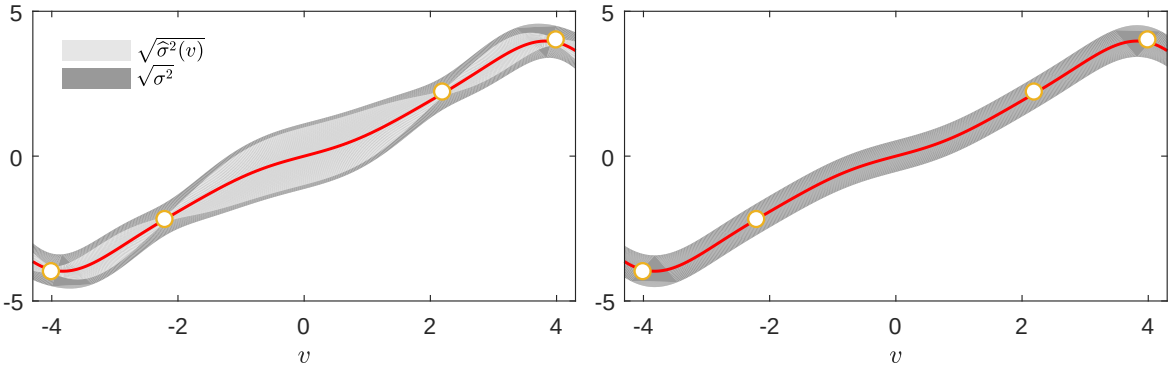
Three methods for approximate Bayesian inference were presented in this section: one based on stochastic approximations (section 5.6.1), and two based on approximations of intermediate distributions. Out of these two, one relies on the central limit theorem (section 5.6.2) and the other prescribes a mean-field factorization over the posterior (section 5.6.3).

Using sampling to approximate the marginal distribution $\mathrm{Q}(F^L)$ puts no constraints on the distributions occurring in the model and thus gives a GPN feed-forward network of two or more layers the power to learn arbitrary distributions $\mathrm{P}(X^L \mid X^0)$ for an input X^0. However, the stochasticity in the objective function introduced by sampling causes the training to require significantly more iterations than with a deterministic objective function and could therefore be prohibitive if GPNs are to be used as a drop-in replacement for conventional artificial neurons.

The variational method proposed in section 5.6.2 uses the central limit theorem to approximate the layer distributions with multivariate normals and thus avoids the stochasticity costs by allowing analytic calculation of the expected value of the objective function. The variational approach using the mean-field approximative posterior (section 5.6.3) is a method that was first proposed for training of deep GPs (Damianou et al., 2013) and, since a GPN feed-forward network can be treated as a deep GP when the calculation of the activations are absorbed into the GP covariance function, using a method developed for that model seems to be a sensible choice. Furthermore, the factorizing approximative posterior does not rely on having enough GPNs per layer and weights that are sufficiently random so that the central limit theorem holds. Indeed, the resulting variational optimization objective will penalize model parameters that lead to non-Gaussian posterior distributions on the intermediate layer values X^l and thus enforce a self-consistent solution.

Thus, so far the usage of the mean-field approach proposed in section 5.6.3 seems advantageous; however, on closer examination the factorization of the posterior distribution over the layers is very problematic. The values of neighboring layers are highly correlated, since the values of layer $l+1$ are determined by the weight matrix and activation function from the values of layer l, with only a minor amount of additional noise added from the GP regression uncertainty of the activation function and the GPN standard deviations σ_n^l. Approximating such highly correlated values with a factorizing distribution, which assumes *independence* per definition, is known to lead to a gross underestimation of variances when variational inference is performed (Bishop, 2006). For a GPN stack this is particularly problematic, because it leads to the loss of input-dependent uncertainty of the activation functions in the GPNs, as we will show in the following illustrative example.

Consider a very simple GPN layer consisting of a single unit and one incoming connection,



(a) GPN activation function

(b) mean-field approximation

Figure 5.14: Effects of the mean-field approximation on the propagation of a sample through a GPN activation function. (a) An exemplary activation function of a GPN has varying amounts of uncertainty due to varying distances from the virtual observations. The total uncertainty is the sum of the input-dependent variance $\widehat{\sigma}^2(v)$ and the input-independent variance σ^2. (b) The mean-field variational approximation using a distribution factorizing over the layers leads to the loss of the input-dependent variance and thus a massive underestimation of the total uncertainty.

i.e. $N_{l-1} = N_l = 1$. Let the weight matrix be the identity matrix ($W^l_{11} = 1$) and let U^l be chosen so that the activation function is any function but with varying amounts of noise due to uncertainty of the GP regression as shown in fig. 5.14a. Assume that we have a single sample and its incoming value $X^{l-1} = v$ is observed.[6] Then the marginal of the response F^l will be given by

$$\mathrm{P}(F^l) = \mathcal{N}(F^l \,|\, v, \widehat{\sigma}^2(v))$$

where $\widehat{\sigma}^2(v)$ is the uncertainty of the GP representing the activation function at position v. The distribution of the layer value X^l is given by

$$\mathrm{P}(X^l \,|\, F^l) = \mathcal{N}(X^l \,|\, F^l, \sigma^2)$$

with σ^2 being the additional, input-independent noise of the GPN. For this layer $\mathcal{L}_{\text{prop}}$ from eq. (5.151) can be written as a KL-divergence

$$\mathcal{L}_{\text{prop}} = \iint \mathrm{P}(F^l)\,\mathrm{Q}(X^l)\,\log\frac{\mathrm{P}(F^l)\,\mathrm{Q}(X^l)}{\mathrm{P}(F^l)\,\mathrm{P}(X^l \,|\, F^l)}\,\mathrm{d}F^l\,\mathrm{d}X^l = \mathrm{KL}\big(\mathrm{Q}(F^l, X^l)\,||\,\mathrm{P}(F^l, X^l)\big)$$

between the joint distribution $\mathrm{P}(F^l, X^l)$ and the factorized approximation $\mathrm{Q}(F^l, X^l) = \mathrm{P}(F^l)\,\mathrm{Q}(X^l)$.

[6] Because we are dealing with a single sample and a single GPN per layer we omit the sample and unit indexing in this example.

Evaluating the exact joint distribution $\mathrm{P}(F^l, X^l) = \mathrm{P}(F^l)\,\mathrm{P}(X^l \,|\, F^l)$ gives

$$\mathrm{P}(F^l, X^l) = \mathcal{N}\left(\begin{pmatrix} F^l \\ X^l \end{pmatrix} \,\middle|\, \boldsymbol{\mu}, \Sigma\right) \qquad \text{with } \boldsymbol{\mu} = \begin{pmatrix} v \\ v \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \widehat{\sigma}^2(v) & \widehat{\sigma}^2(v) \\ \widehat{\sigma}^2(v) & \widehat{\sigma}^2(v) + \sigma^2 \end{pmatrix}.$$

It can be shown analytically (Bishop, 2006) that for a two-dimensional Gaussian distribution

$$\mathrm{P}(Z_1, Z_2) = \mathcal{N}\left(\begin{pmatrix} Z_1 \\ Z_2 \end{pmatrix} \,\middle|\, \boldsymbol{\mu}^Z, \Sigma^Z\right)$$

the best factorizing approximation $\mathrm{Q}(Z_1, Z_2) = \mathrm{Q}(Z_1)\,\mathrm{Q}(Z_2)$, in the sense that the KL-divergence between Q and P is minimized, is given by

$$\mathrm{Q}(Z_1) = \mathcal{N}\big(Z_1 \,|\, \mu_1^Z, \Lambda_{11}^{-1}\big)\,, \qquad \mathrm{Q}(Z_2) = \mathcal{N}\big(Z_2 \,|\, \mu_2^Z, \Lambda_{22}^{-1}\big)\,,$$

where $\Lambda \triangleq (\Sigma^Z)^{-1}$ is the precision matrix of $\mathrm{P}(Z)$. Using this result to calculate the optimal approximation $\mathrm{Q}(X^l)$ gives

$$\mathrm{Q}(X^l) = \mathcal{N}\big(X^l \,|\, v, \sigma^2\big)\,.$$

The mean matches the exact joint distribution, however we see that only the input-independent uncertainty σ^2 of a GPN is kept and thus propagated to the next layer by the factorizing variational approximation. This can be seen in fig. 5.14b.

Consequently, the variance is not only underestimated by the $\mathcal{L}_{\text{prop}}$ term coming from the mean-field variational approach, but a central element of our model, namely the input-dependent uncertainty of the activation function $\widehat{\sigma}^2(v)$ has been lost. The marginalizing approach (section 5.6.2) does not suffer from this issue and thus should always be preferred when the number of GPNs is sufficient and the weights are initialized randomly.

5.7 Benchmarks and Experimental Results

Due to limitations of computational resources and time, the results presented here correspond only to the *parametric* GPN model as described in section 5.3. Experiments using variational Bayesian inference as presented in section 5.6 were still ongoing as this thesis was printed and thus their results could not be included with adequate statistical confidence. The experiments presented here were done by Basalla (2017) under my guidance and results are reproduced here in excerpts.

This section describes the experiments performed to estimate the performance of a GPN feed-forward network. The datasets on which the models are evaluated is introduced and the performance is compared to conventional feed-forward neural networks that were regularized using fast Dropout (Wang et al., 2013). Furthermore, this section discusses the activation functions learned by the GPNs in different layers of the feed-forward network and the computational and memory requirements.

5.7.1 Benchmark Datasets

To evaluate how well a parametric GPN model performs on real-world classification problems, we test it on three datasets from the UCI Machine Learning Repository (Lichman, 2013) as well as on the MNIST database of handwritten digits (Lecun et al., 1998). Hereby our aim is not yet to beat the current state of the art performance on these datasets, since the literature (Agostinelli et al., 2014) shows that trainable activation function are mostly beneficial to large convolutional models for image classification. Instead, we focus on verifying the implementation, efficiency and trade-offs of parametric GPN feed-forward models on different kinds of classification datasets compared to conventional neural networks with a fixed sigmoidal or hyperbolic tangent activation function regularized by the Dropout technique (Srivastava et al., 2014). The datasets were primarily chosen so that they do not only differ in size but also cover different kinds of features and targets. Consequently, successful training on this selection of datasets shows that GPNs are applicable to a variety of tasks and it is worthwhile to implement convolutional architectures based on GPNs to tackle current image classification problems on large datasets such as CIFAR-100 (Krizhevsky and G. Hinton, 2009) and ImageNet (Deng et al., 2009). We shortly describe the properties of the dataset before continuing with the training procedures.

UCI Letter Recognition Dataset

The UCI Letter Recognition dataset, first used in (Frey et al., 1991), consists of $20\,000$ samples with 16 continuous input features per sample. All features are calculated from pixel images of the 26 capital characters of the English alphabet, with each image showing a single letter in one of 20 different fonts. Furthermore the character images are randomly distorted to increase the variation of the dataset. The 16 precomputed features, consisting of statistical moments and edge counts, are used as input to the classifier. The objective is to identify the character.

UCI Adult Dataset

The UCI Adult dataset, introduced by (Kohavi, 1996), consists of 6 continuous and 8 categorical features containing census data taken from $48\,848$ U.S. citizens collected in the year 1994. The continuous features consists amongst others of the age, weight, work hours per week and years of educations. The categorical features include such information as highest obtained degree, martial status, race, sex and country of origin. The binary target objective is to predict whether a person's income exceeded $50\,000$ USD or not. This dataset contains missing features for some samples, which we replaced by an additional "unknown" category for categorical features and by zero for continuous features.

UCI Connect-4 Dataset

In contrast to the previous datasets, the Connect-4 dataset (John Tromp, Lichman (2013)) consists only of categorical features. Each of its 42 features represents the state of a field on the board of a Connect-4 game with a board size of 6×7. The categories encode whether the position is currently occupied by player 1, by player 2 or is free. The dataset contains all legal positions in which neither player has won yet and in which the next move is not forced; in total the dataset contains $67\,557$ samples. The data is used to classify the game result for player 1 if she plays optimally into one of the three classes: "win", "loss" or "draw".

MNIST Dataset

The MNIST database of handwritten digits (Lecun et al., 1998) is one of the most commonly used machine learning datasets for image classification. It consists of $60\,000$ training and $10\,000$ test examples. We further split the training examples into a training and validation set. Each input consists a 28×28 pixel image of a handwritten digit that has to be classified. This task is similar to classification on the letter-recognition dataset, with the main difference being, that instead of using precomputed image features the model receives the raw, grayscale

image data as input. The MNIST dataset has been widely used to evaluate the performance of neural network based classifiers (G. E. Hinton, 2007; G. E. Hinton and R. R. Salakhutdinov, 2006; G. E. Hinton, Srivastava, et al., 2012; R. Salakhutdinov et al., 2009) and is thus a natural choice for evaluating trainable activation functions in a neural architecture.

5.7.2 Training Procedures

In our preliminary testing we want to demonstrate that parametric GPNs can be used as a drop-in replacement for conventional artificial neurons. Consequently, we perform the initial experiments using the parametric GPN normal approximation developed in section 5.5, since it results in a deterministic loss function, allowing to optimize the model parameters using gradient descent just like in a conventional neural network.

As stated in theoretical analysis of the computational complexity in table 5.2, propagating mean, variance and the full covariance matrix from layer to layer comes with different computational and memory requirements. To analyze the trade-offs between runtime and prediction accuracy of these different approximations we train parametric GPNs by only propagating the mean, by propagating the mean and only the diagonal of the covariance matrix and by propagating the mean and the full covariance matrix. To only propagate the mean, all layer variances $\widetilde{\Sigma}^{X^l}_{s\star\star}$ are assumed to be zero and only eqs. (5.84a) and (5.87) are evaluated for each layer; thereby the probabilistic nature of the model is eliminated and the GPNs become conventional neurons with an activation function that is given by interpolating between their inducing points and targets. Including the variance $\mathrm{diag}(\widetilde{\Sigma}^{X^l}_{s\star\star})$ in the computations is done by setting all off-diagonal elements of the layer covariance matrices to zero and using eqs. (5.84b) and (5.101). For the full model we propagate $\widetilde{\Sigma}^{X^l}_{s\star\star}$ from layer to layer by employing eqs. (5.84b) and (5.102). The wall clock time and memory requirements of each approach are measured.

The virtual observations of the parametric GPN model can be shared between different GPNs resulting in GPNs that use the same activation function. Obviously this reduces the number of model parameters and furthermore the computational complexity, since $[K(V_{\star n}, V_{\star n}) + \mathrm{diag}(S_{\star n})]^{-1}$ in eqs. (5.47a) and (5.47b) is only computed once per group of GPNs with shared virtual observations. To assess the impact of sharing on model accuracy, we train two variants of GPN feed-forward networks: an independent, where each GPN has its individual virtual observations, and a layer-shared, where all GPNs within a layer share one activation function.

Preparatory experiments showed that the inducing points V^l_{rn} of each GPN remained mostly unchanged during training; hence the 14 inducing points are initialized using linear spacing in the interval $[-2, 2]$ and kept fixed during training. The corresponding targets U^l_{rn} are either initialized from a standard normal distribution or set equal to V^l_{rn}, resulting in the identity

function. We also tried initializing the targets to values of a well-known activation function, such as the hyperbolic tangent or the rectifier, but found no significant benefit. All virtual observation variances S^l_{rn} are initialized to the constant value of $\sqrt{0.1}$ and optimized during training alongside with the targets. In preparatory experiments it became apparent that the observation variances were driven to zero. To avoid this a penalty of the form

$$\mathscr{L}_S(\theta) = \sum_{l=1}^{L} \frac{1}{N_l} \sum_{n=1}^{N_l} \frac{1}{R} \sum_{r=1}^{R} \alpha_S \, \sigma\left(\frac{\beta_S}{|S^l_{rn}|}\right) \tag{5.163}$$

where σ is the logistic function, $\alpha_s = 0.1$ and $\beta_s = 10^{-3}$, was added to the loss function. This will become unnecessary once the variational Bayesian training objective derived in section 5.6 is used.

The weights W^l_{nm} of each layer l are initialized using a uniform distribution with support $[-r, r]$ where $r = \sqrt{6}/\sqrt{N_{l-1} + N_{l+1}}$. This initialization has been recommended by (Glorot and Bengio, 2010) for training of deep neural network using the hyperbolic tangent activation function. The motivation behind choosing r as described is to ensure that at the beginning of training the activations of most neurons start in the linear range of the hyperbolic tangent function. Although we are not using this function, it is desirable for the activations of GPNs to fall within the range of their inducing points; thus this weight initialization method is applicable here.

The continuous input features of all datasets are rescaled and shifted to lie in the interval $[0, 1]$; for categorical features the one-hot encoding scheme is used. It encodes a categorical feature as a vector having as many entries as there are categories with the entry for the active category being set to one and all other entries being set to zero. The split between training and test set is kept as provided in the datasets; furthermore the original training set is randomly split into a smaller training set and a validation set consisting of 10% of the original training samples. The test set is only used to report final classification accuracies and not used in any way during training.

Training is performed by minimizing the expected loss $\mathscr{L}(\theta)$ as calculated by the unscented transform (5.113) of the softmax cross-entropy loss (5.53) using the Adam optimizer (D. Kingma et al., 2014). The initial learning rate is 10^{-3} and is decreased by factor 10 each time the validation loss stops improving for a predefined number of training iterations. When the learning rate reaches 10^{-6} and no improvement is seen on the validation set, training is terminated and the model parameters of the best iteration as seen on the validation set are used to calculated the reported classification accuracies. Each experiment is repeated five times with different random seeds for the initialization. For comparison we also train a conventional

neural network of the same architecture with a fixed hyperbolic tangent activation function and regularized using the fast Dropout method (Wang et al., 2013).

Preparatory experiments showed that enforcing increasing activation functions (section 5.4) resulted in slightly worse results than without that constraint. Since this constraint imposes a significant increase in computational complexity due to the introduction of virtual derivative observations, the set of experiments presented here were all done without enforcing monotonicity.

5.7.3 Preliminary Results

An exemplary loss curve and training rate schedule of a GPN feed-forward network is shown in fig. 5.15. In this example the initial learning rate of 10^{-3} is automatically decreased after 400 iterations to 10^{-4} and then again after $9\,000$ iterations to 10^{-6}. Training is terminated after $20\,000$ iterations. Like a conventional feed-forward neural network, the losses decrease smoothly due to the use of a fully deterministic loss.

The accuracies of all experiments are reported in table 5.3 and the resource usage of the different propagation approaches is shown in table 5.4. As expected the conventional neural network with a fixed activation function is fastest; however relative to that using a GPN is only four times slower when propagating the means and the variances. This includes both the times for forward propagation and for calculating the gradient w.r.t. the weights and virtual observations using back propagation. The propagation of the GPN mean and variance leads to significantly better results than the propagation of the mean alone on all datasets. However, the propagation of the full covariance matrix is about 12 times computationally more expensive than the propagation of the mean and variance and it did not show significant benefits to the accuracy of the model in preparatory experiments. Sharing the GPN virtual observations over all GPNs within a layer does not provide any benefits on the test accuracy in our experiments, thus suggesting that the flexibility of having a separate activation function per GPN is beneficial to the model and the increase in the number of parameters does not lead to overfitting. On the UCL Adult dataset GPNs profited from initializing their virtual observations so that the initial activation function is the identity function; however doing the same on the UCL Letter Recognition dataset did not yield any significant improvement.

Figure 5.16 shows examples of activation functions that are commonly encountered in a GPN feed-forward network after training it on UCL Connect-4 dataset. The activation functions in the first layer vary much stronger than those in the upper two layers. Most commonly functions in the first layer resemble sine-like functions and are approximately axis-symmetric w.r.t. the y-axis. As we move to the second and third layer, sigmoid-shaped and linear functions

Dataset	*Layer sizes*	*Units*	*Sharing*	*Act. init.*	*Train error*	*Val. error*	*Test error*
UCL Letter Rec.	16x30x15x26	fixed tanh			0.0463 ± 0.0039	0.0690 ± 0.0068	0.0765 ± 0.0035
	16x30x15x26	fixed tanh (dropout)			0.0354 ± 0.0054	0.0603 ± 0.0046	$\mathbf{0.0625} \pm 0.0042$
	16x30x15x26	GPN mean only	none	random	0.0696 ± 0.0163	0.0690 ± 0.0138	0.0765 ± 0.0104
	16x30x15x26	GPN mean + variance	none	random	0.0366 ± 0.0052	0.0668 ± 0.0061	0.0709 ± 0.0043
	16x30x15x26	GPN mean + variance	none	identity	0.0348 ± 0.0043	0.0675 ± 0.0034	0.0715 ± 0.0041
	16x30x15x26	GPN mean + variance	layer	random	0.0410 ± 0.0060	0.0704 ± 0.0026	0.0733 ± 0.0033
UCL Adult	104x30x15x2	fixed tanh			0.1417 ± 0.0070	0.1586 ± 0.0028	0.1561 ± 0.0032
	104x30x15x2	fixed tanh (dropout)			0.1431 ± 0.0006	0.1420 ± 0.0015	0.1418 ± 0.0014
	104x30x15x2	GPN mean + variance	none	random	0.1413 ± 0.0015	0.1490 ± 0.0051	0.1514 ± 0.0059
	104x30x15x2	GPN mean + variance	none	identity	0.1469 ± 0.0003	0.1408 ± 0.0018	$\mathbf{0.1390} \pm 0.0008$
UCL Connect-4	126x30x15x3	fixed tanh			0.0129 ± 0.0044	0.1535 ± 0.0014	0.1637 ± 0.0021
	126x30x15x3	fixed tanh (dropout)			0.1372 ± 0.0037	0.1454 ± 0.0017	$\mathbf{0.1568} \pm 0.0019$
	126x30x15x3	GPN mean + variance	none	random	0.1222 ± 0.0063	0.1494 ± 0.0048	0.1608 ± 0.0017
	126x30x15x3	GPN mean + variance	layer	random	0.1273 ± 0.0059	0.1501 ± 0.0054	0.1617 ± 0.0062
MNIST Digits	784x30x15x10	fixed tanh			0.0162 ± 0.0023	0.0451 ± 0.0016	0.0546 ± 0.0021
	784x30x15x10	GPN mean + variance	none	random	0.0212 ± 0.0017	0.0426 ± 0.0009	$\mathbf{0.0521} \pm 0.0016$

Table 5.3: Misclassification rates of different models on the benchmark datasets. The error is calculated as the number of misclassified samples divided by the number of total samples. The error rates are given as the average of five experiments and with a confidence interval of 68%. For evaluation the model parameters at the training iteration with the lowest validation loss were used.

Propagation using	*Memory usage*	*Iteration time*
fixed act. function	30 MB	9 ms
GPN means only	94 MB	29 ms
GPN means and variances	113 MB	36 ms
GPN means and full covariances	227 MB	118 ms

Table 5.4: Memory usage and time for performing one iteration of forward- and backpropagation of a layer of 50 neurons or GPNs using different propagation methods. Values should only be used for relative comparisons within this table since some irrelevant operations, such as data reading, are included in the memory usage and iterations time. Memory usage includes the memory used for storing intermediate results for calculation of the derivatives using backpropagation.

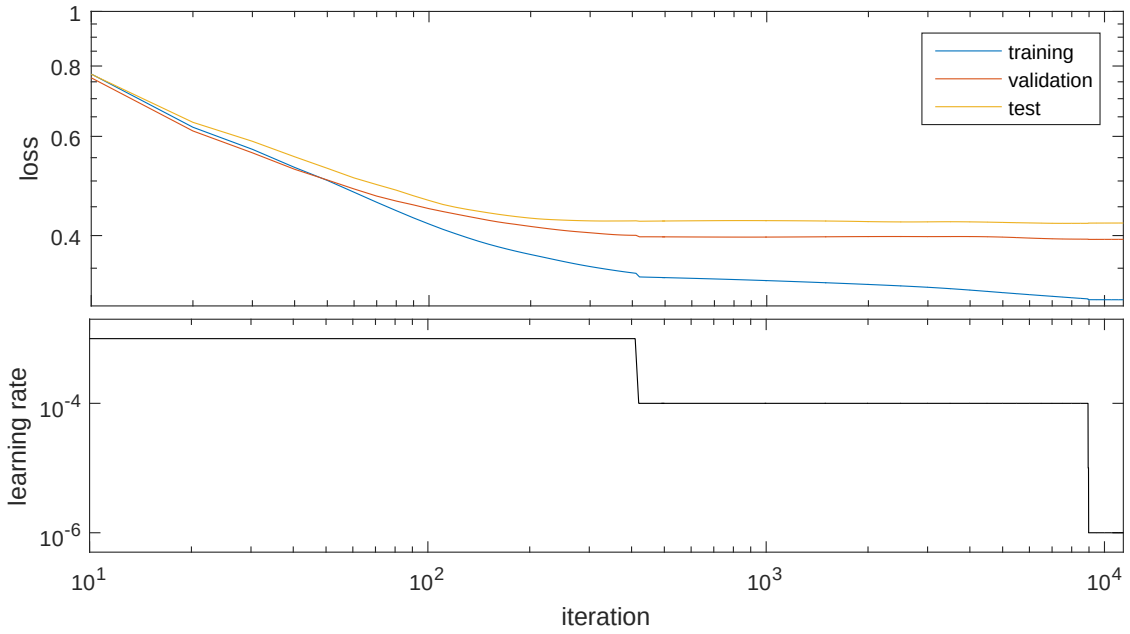


Figure 5.15: Training, validation and test losses during training of a parametric GPN feed-forward network with mean and variance propagation on the UCL Connect-4 dataset. The lower panel shows the scheduling of the learning rate. The progression of the loss is smooth and stable due to the use of a fully deterministic objective.

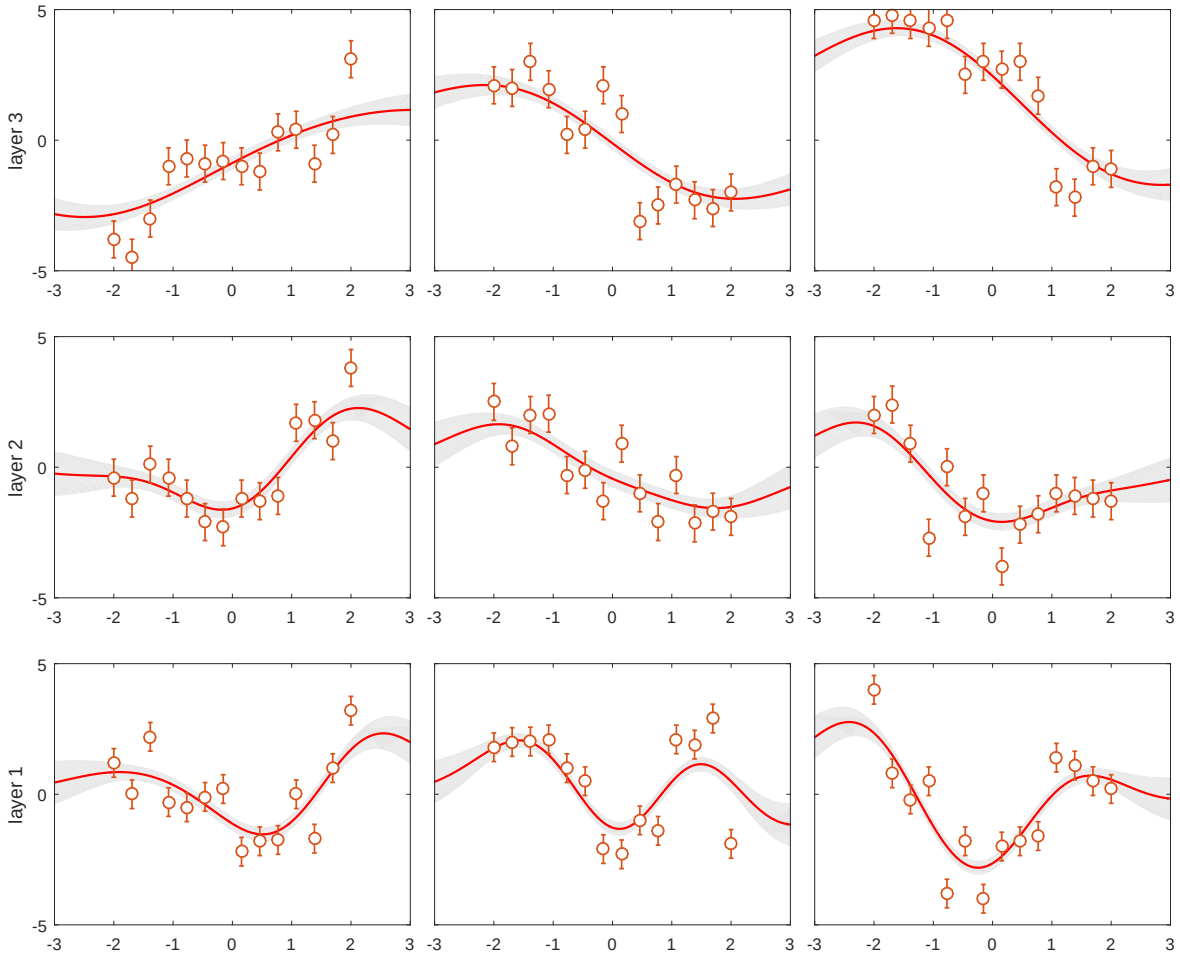


Figure 5.16: Three activation function from each layer of a parametric GPN feed-forward network that was trained on the UCL Connect-4 dataset. The virtual observations are shown as red dots together with their standard deviation. Activation functions in lower layers show oscillatory behavior, while the activation functions in the top-most layer resemble sigmoid-shaped functions.

become more common. This might indicate that the first layer exploits a periodicity in the input data while the top two layers act as feature-detectors by gating their inputs.

The preliminary results presented here show that GPNs have consistently better performance than a conventional neural network using the hyperbolic tangent activation function on real-world datasets of small size. When the conventional neural network is regularized using the Dropout method, the performance difference becomes marginal, with GPNs winning on some datasets while conventional neural networks perform better on others. The benefit of activation function initialization is dataset depended and sharing virtual observations does not yield

improvements. Enforcing monotonicity of the activation function has not proven helpful. Since the computational resources available for these experiments were limited, it was not possible to perform tuning of the network architecture, i.e. experimenting with the number of layers and their sizes.

Chapter 6

Conclusion

We proposed two novel classes of activation functions for artificial neural networks. The first class of activation functions introduced in chapter 3 allows neural networks to automatically introduce multiplicative interactions during training and thus reduces the need to hand-engineer such interactions or use computationally expensive methods to search over different network architectures. Compared to conventional activation functions, the proposed family of activation functions shows favorable results on datasets based on a multiplicative structure but displays slightly worse performance on standard image recognition tasks. The reason for this behavior is believed to be a more complex error surface and a greater risk of overfitting due to the increased expressive power of a neural network using such activation functions.

To evade these drawbacks, in chapter 5 we proposed to place a Gaussian process prior over the activation function of each neuron. This has three consequences. First, the neuron using this activation function becomes a probabilistic unit, allowing it to handle uncertain inputs and estimate the confidence of its output. Second, complexity of the activation function is penalized in a probabilistically sound Bayesian setting; this guards the model against overfitting. Third, the squared exponential covariance function ensures that all activation functions are smooth and therefore continuous derivatives are available. This resulted in the non-parametric GPN model, which shows these theoretically attractive properties, but performing inference is expensive due to its non-parametric nature.

An overview of the course of action we took to make GPNs tractable is shown in fig. 6.1. Starting from a non-parametric model we derived a variational approximation of the posterior. Based on methods proposed for sparse GP regression, we introduced an auxiliary model, the parametric GPN, that provides inexpensive inference but is also less attractive since inference is performed by maximizing the likelihood. We then showed that it is possible to recover the non-parametric GPN model by placing an appropriate prior over the parameters of a parametric GPN. Furthermore, we showed that the distribution of activations in both randomly initialized and

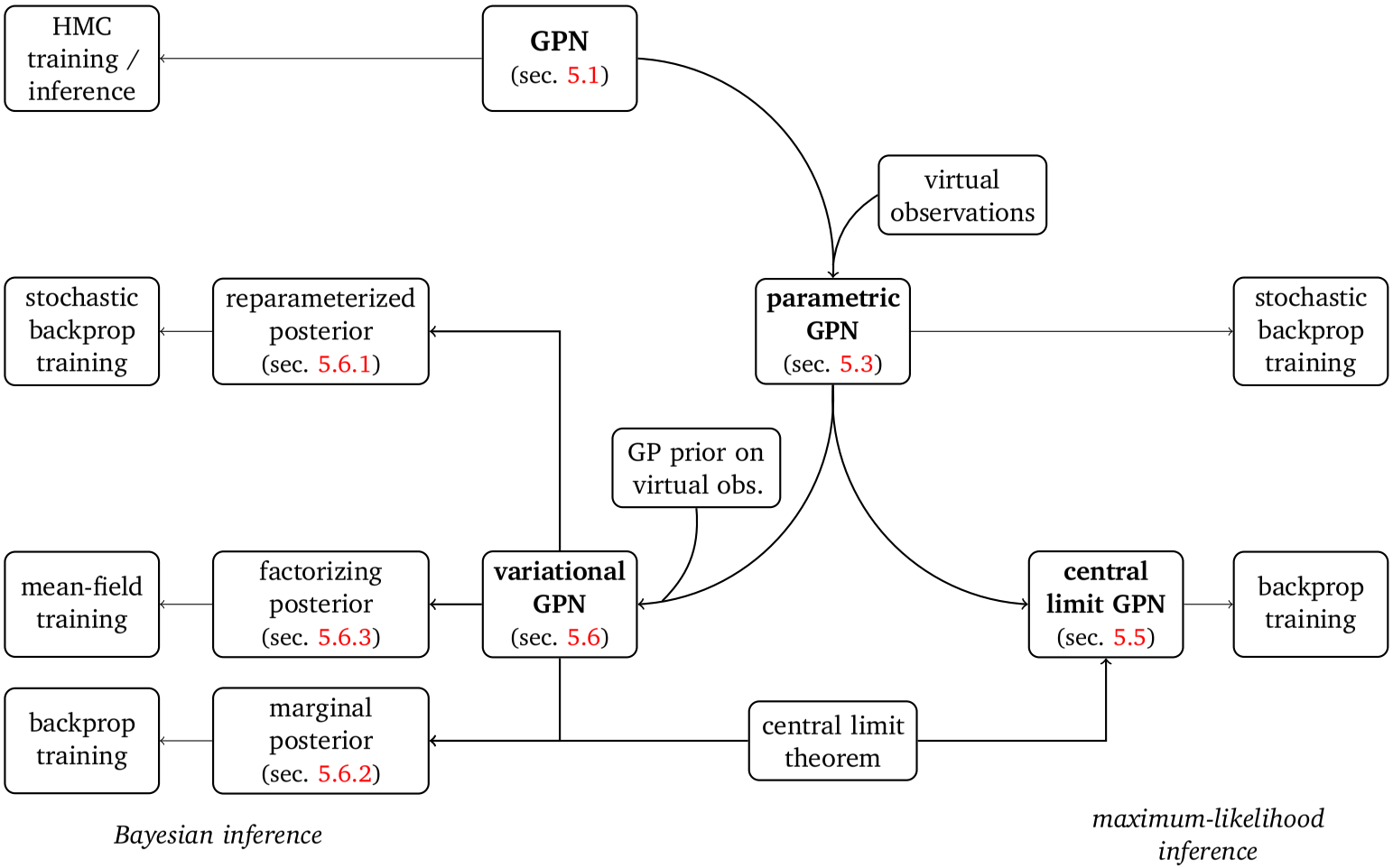


Figure 6.1: Overview of the family of GPN models and their relationships as well as training and inference methods.

fully trained neural networks closely resembles a normal distribution due to the central limit theorem. Taken together, these two steps allowed us to derive a fully deterministic, variational objective to train a non-parametric GPN by stochastic gradient descent. This objective has the same functional structure as that of a conventional neural network and thus GPNs can be directly included in CNNs and RNNs or any other architecture that uses neurons.

Implementing this model with high computational performance made it necessary to develop new methods for efficient evaluation of expressions on GPUs, as existing software package are not very efficient when dealing with a large number of small matrices that occur in our model. We wish to mention that this thesis mostly focused on the mathematical point of view of the proposed models; nevertheless a significant amount of time and effort went into optimizing the implementation of fractional exponentiation and GPNs so that their computational performance becomes comparable to that of conventional artificial neurons. This technical work is necessary to make these models applicable to large datasets that have become common in machine learning in the last few years. Although a full description of the employed implementation techniques and algorithms goes beyond the scope of this work, we believe that the methods developed in chapter 4 for efficient derivation of elementwise defined tensors are applicable to a wide variety of problems inside and outside the field of machine learning and thus decided to present them here.

In preliminary experiments we showed that on classification and regression tasks the parametric GPN performs as least as well as a Dropout regularized neural network with comparable computational performance. Due to time constraints we were not yet able to perform experiments on large datasets or using CNNs which have proven to benefit the most from novel activation functions over the course of the last years.

6.1 The Relation to Deep Gaussian Processes

Deep Gaussian processes (Damianou et al., 2013) is a framework for hierarchical composition of GP functions. Similar to the GPN model, outputs of one GP are used as the input for another one; thus graphical models resembling the structure of a feed-forward neural network can be formed. Deep GPs also employ the variational sparse GP method using inducing points developed by Titsias (2009) to make inference tractable. However, as we will show now, GPNs have a number of advantages over deep GPs both in model complexity and efficiency of inference.

A GP within the deep GP framework takes multidimensional input, i.e. each input connection adds an input dimension to the GP it connects to. The connections in a deep GP do not use weights to compute a weighted sum as it is done in the GPN model. Instead, each GP uses the ARD covariance function that has an individual lengthscale parameter per input dimension. By

interpreting the lengthscale of the ARD covariance function as the inverse of a weight, we can write for the covariance function of a deep GP,

$$k_{\mathrm{ARD}}(\boldsymbol{y}, \boldsymbol{y'}) = \exp\left[-\sum_d w_d^2 (y_d - y'_d)^2\right] .$$

Compared to that the effective covariance function of a GPN is

$$k_{\mathrm{GPN}}(\boldsymbol{y}, \boldsymbol{y'}) = \exp\left[-\left(\sum_d w_d (y_d - y'_d)\right)^2\right] .$$

Thus taking the square *before or after* summation is what distinguishes GPNs from deep GPs in their essence. Although at first glance this seems to be a rather small difference, it leads to a series of consequences that clearly distinguishes both models.

The first consequence is that each GP in the deep GP framework works in a *multidimensional* function space. The dimensionality is determined by the number of input connections and thus in a feed-forward model it equals the number of units in the previous layer. Hence the inducing points of the virtual observations used for efficient inference are also multidimensional. This implies that the number of virtual observations required to evenly cover the input space scales *exponentially* with the number of input dimensions and thus incoming connections. Figure 6.2a shows the predictive mean of a two-dimensional GP with the ARD covariance function with four observations. As one moves further away from these observations the predictive mean returns to zero.

On the other hand a GPN, like every artificial neuron, computes the projection of its inputs onto its weight vector resulting in a *scalar* value. Thus no matter how many input connections are present, a GPN always works in a *one-dimensional* function space. Hence the inducing points of the virtual observations are also one-dimensional and the number of virtual observations per GPN is unaffected by the number of incoming connections. Figure 6.2b shows the predictive mean of a GP using a projection of a two-dimensional input space and four observations. Thus inducing points become inducing lines or inducing hyperplanes when more than two dimensions are concerned. It might be argued that the expressive power is vastly reduced by using a projection, however this is not the case in a GPN feed-forward network as the following argument demonstrates. Figure 6.2d also shows the predictive mean of a GP using a projection of a two-dimensional input space but with different weights. Assume that fig. 6.2b and fig. 6.2d are the outputs of two GPNs located in the same layer. For the sake of argument further assume that this particular layer consists only of these two GPNs. Then the activations of a GPN in the subsequent layer is formed by a linear combination of the output of these two GPNs. The

resulting activation (using equal weights) is shown in fig. 6.2c and, as it can be seen, it produces functions varying in *both* input dimensions. Here, the number of virtual observations required to evenly cover the input space scales *linearly* with the number of input dimensions. Furthermore, the virtual observations can now be interpreted as a grid in input space, thus making it unlikely that an input point is located far away from all inducing hyperplanes.

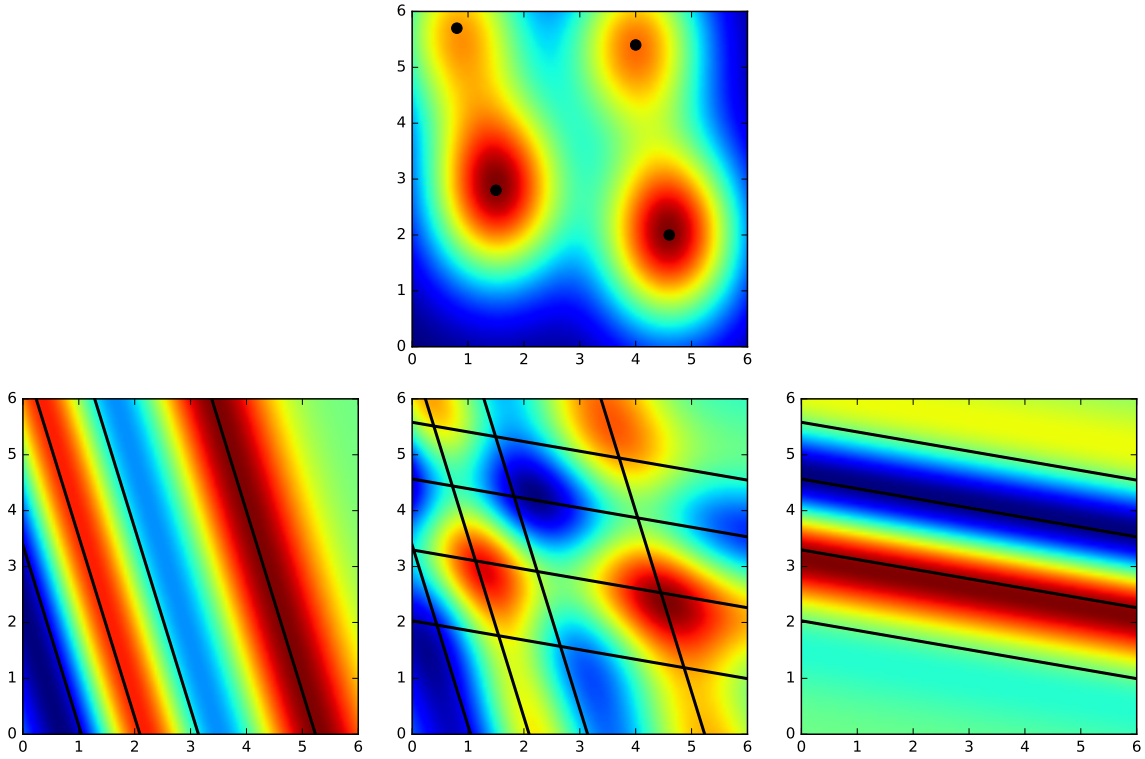
The second consequence is that the inputs to a GP in a deep GP model cannot converge to a normal distribution because no linear combination (as in a neural network) is performed. This leaves two methods for training and inference of a deep GP: stochastic variational inference (section 5.6.1), which is computationally expensive due to sampling, or using a *mean-field* variational posterior (section 5.6.3) which is a bad fit for the model and thus leads to poor propagation of uncertainties as discussed in section 5.6.4. Furthermore, the mean-field outputs of each GP within a deep GP must be inferred alongside the model parameters during minimization of the variational objective function, leading to many more parameters to optimize. Salimbeni et al. (2017) also observed that deep GPs are difficult to train for these reasons and reverted to a stochastic inference algorithm to avoid this problem, albeit at significantly higher computational costs.

On the other hand, the central limit theorem is applicable to the activation of each GPN, guaranteeing that the activations will converge to a normal distribution if a GPN has a sufficient number of input connection. This leads to the *marginally* normal variational posterior that we derived in section 5.6.2. The variational objective function resembles the structure of a neural network, which makes GPNs directly usable in other network architectures such as RNNs and CNNs simply by adjusting eq. (5.142) accordingly.

Taken together both consequences show that the design of a GPN leads to a more sound and efficient training procedure with fewer parameters to optimize compared to a deep GP. These advances were possible by using a weight projection inside a standard SE covariance function instead of an ARD covariance function. This crucial difference is what enabled us to derive the vastly more efficient variational objective.

In summary, from a neural network viewpoint we have introduced a novel, stochastic, self-regularizing activation function that is integratable into existing neural models with modest effort. From a GP viewpoint we introduced the idea of learnable projections into deep Gaussian processes, allowing us to derive a novel variational posterior that makes them as accessible and easy to train as neural networks.

(a) GP ARD



(b) GPN 1 (c) GPN 1 + GPN 2 (d) GPN 2

Figure 6.2: GP with ARD covariance function as it occurs in a deep GP (top) versus GPNs using projections (bottom) in two-dimensional space. (a) The predictive mean of a GP in two-dimensional space using the ARD covariance function and four observations placed in two-dimensional space. (b) The predictive mean of a GP using a projection of two-dimensional inputs and four one-dimensional observations that are represented as lines in input space. (d) Shows the same as (b) but using a different projection. (c) A linear combination of the predictive means of (b) and (d) as it occurs in the activation of a GPN receiving inputs from the GPNs shown in (b) and (d). Taken together their observations form a grid in two-dimensional space.

6.2 Future Work

Many extensions to and applications of the proposed model are imaginable.

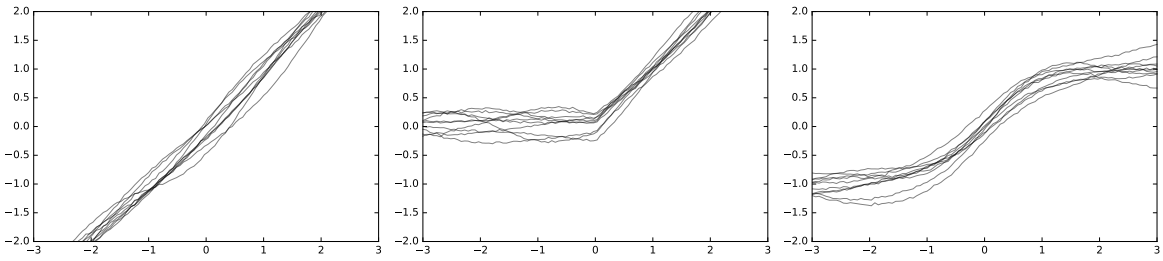
The behavior of the activation function at infinity

Although the GPN can approximate all popular activation functions within a limited range, the value of the GPN activation function will always return to zero as one moves towards negative or positive infinity. This follows directly from using the zero mean function and the squared exponential covariance function. However, in CNNs for image processing a neuron often fulfills the function of a feature detector that measures how well the image patch in its receptive field matches a reference pattern encoded in its weights. The usually employed logistic function thus serves as a soft threshold that results in an output of one if the pattern is matched and zero if not. When using GPNs an issue could arise when a unit obtains a very good match. In this case the resulting very high activation value could be outside the range of the inducing points and thus the thresholding behavior may be impacted.

Two possible solutions exist. One solution is to extend the covariance function of the employed GP. For this it should be remembered that a GP can also be interpreted as linear regression in a feature space with the weights following a normal prior. Thus, by using the identity function and the step function as the basis functions (feature maps) of this feature space, a GP with the appropriate covariance function, corresponding to the scalar product in feature space, will produce a linear combination of these basis functions. We can add the so-defined covariance function to the squared exponential covariance function to obtain a GPN that can represent any function within the range of its inducing points and also control its behavior at infinity by means of these basis functions. This approach requires adapting the equations for propagating of uncertainty to the extended covariance function. Another solution is to change the mean function of the GP as discussed below.

Non-zero activation function means

The GPN has been developed using the zero mean function for the GP representing its activation function. Using a particular mean function leads to that function having the highest prior probability in the distribution over activation functions. The zero mean function used so far assigns the highest prior probability to a GPN which has constant zero output regardless of its inputs and thus behaves as if it was not present in the network. In consequence the zero mean function regularizes GPN networks by keeping the number of active GPNs in each layer to a minimum.



(a) identity (b) rectifier (c) tanh

Figure 6.3: Using established activation functions as mean functions for the GPN. This figure shows samples from the resulting prior before training.

By using other mean functions we can impose different regularization goals. For instance a GPN using the identity function as its mean function (fig. 6.3a) will assign highest probability to a completely linear GPN and thus behaves as if its incoming connections were directly connected to its outgoing connections. Thus this function regularizes GPN networks by keeping non-linearities, which can be interpreted as the effective number of layer, to a minimum.

We can also use an established activation function like the rectifier (fig. 6.3b) or the hyperbolic tangent (fig. 6.3c) as the GPN mean function. If the virtual observations are also initialized to be zero, the GPN network starts training as if it was a conventional neural network with that particular activation function. However, it has the power to modify the activation function to obtain a better fit for the given training data. But, as before, a deviation from the prescribed mean function is penalized by the prior.

Using non-zero mean functions requires adjusting the equations for propagation of uncertainties through the GPN network. The best method depends on the particular mean function, but in general the unscented transform can be used to propagate the marginal normal distributions through any mean function.

Empirical evaluation of the variational posterior

The results shown in section 5.7 were performed using the parametric GPN using maximum likelihood inference. They showed that even the parametric model is already capable of beating the performance of Dropout regularized neural networks on some real world datasets. A full set of experiments using the variational posterior could not be performed before printing of this work due to limitations of available time and computational resources. Since the variational training objective is fully specified, a full evaluation of its empirical performance is the next logical step.

Application of GPNs in convolutional and recurrent networks

Novel activation functions like the ReLU and leaky ReLU have been developed for and were most successful in CNNs. Recurrent neural networks on the other hand have proven to benefit from an attention mechanism realized by multiplicative interactions, which can be implemented using a GPN. Thus it is natural to evaluate the proposed model on these architectures. Implementation is straightforward since a GPN retains the structure of a neuron. However, due to the large size of CNN and RNN models combined with our limited computational resources we were not yet able to perform these experiments.

Bibliography

Abel, N.H. (1826). “Untersuchung der Functionen zweier unabhängig veränderlichen Größen x und y, wie f(x, y), welche die Eigenschaft haben, daß f(z, f (x, y)) eine symmetrische Function von z, x und y ist.” In: *Journal für die reine und angewandte Mathematik* 1826.1, pp. 11–15.

Adkins, William A. and Steven H. Weintraub (1999). *Algebra: An Approach via Module Theory (Graduate Texts in Mathematics)*. Springer.

Agarwal, Ravi (2001). *Fixed point theory and applications*. Cambridge, UK New York, N.Y., USA: Cambridge University Press.

Agostinelli, Forest, Matthew Hoffman, Peter Sadowski, and Pierre Baldi (2014). “Learning activation functions to improve deep neural networks”. In: *arXiv preprint arXiv:1412.6830*.

Alain, Droniou and Sigaud Olivier (2013). “Gated autoencoders with tied input weights”. In: *International Conference on Machine Learning*, pp. 154–162.

Andoni, Alexandr, Rina Panigrahy, Gregory Valiant, and Li Zhang (2014). “Learning Polynomials with Neural Networks”. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, pp. 1908–1916.

Arnol’d, Vladimir Igorevich (2013). *Mathematical methods of classical mechanics*. Vol. 60. Springer Science & Business Media.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473*.

Bahdanau, Dzmitry, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio (2016). “End-to-end attention-based large vocabulary speech recognition”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, pp. 4945–4949.

Barron, Andrew R (1993). “Universal approximation bounds for superpositions of a sigmoidal function”. In: *IEEE Transactions on Information theory* 39.3, pp. 930–945.

— (1994). “Approximation and estimation bounds for artificial neural networks”. In: *Machine Learning* 14.1, pp. 115–133.

Basalla, Marcus (2017). “Gaussian Process Activation Functions for Stochastic Neural Networks”. Technical University Munich.

Bayer, Justin (2013). *Climin: optimization, straight-forward*. URL: http://climin.readthedocs.org.

Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and Trends in Machine Learning* 2.1, pp. 1–127.

Bergstra, James, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio (2010). “Theano: A CPU and GPU math compiler in Python”. In: *Proc. 9th Python in Science Conf*, pp. 1–7.

Billingsley, Patrick (2013). *Convergence of probability measures*. John Wiley & Sons.

Bishop, Christopher M (2006). *Pattern Recognition and Machine Learning*. Ed. by Bernhard Schölkopf Michael Jordan Jon Kleinberg. Springer Science and Media, LLC.

Blundell, Charles, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra (2015). “Weight uncertainty in neural networks”. In: *arXiv preprint arXiv:1505.05424*.

Boyd, Stephen and Lieven Vandenberghe (2004). *Convex optimization*. Cambridge University Press.

Brigham, E. (1988). *Fast Fourier Transform and Its Applications*. Prentice Hall.

Bromiley, Paul (2003). *Products and convolutions of gaussian probability density functions*. Tech. rep.

Casella, George and Edward I George (1992). “Explaining the Gibbs sampler”. In: *The American Statistician* 46.3, pp. 167–174.

Chorowski, Jan K, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio (2015). “Attention-based models for speech recognition”. In: *Advances in Neural Information Processing Systems*, pp. 577–585.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). “Fast and accurate deep network learning by exponential linear units (elus)”. In: *arXiv preprint arXiv:1511.07289*.

Coppersmith, Don and Shmuel Winograd (1987). "Matrix multiplication via arithmetic progressions". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, pp. 1–6.

Damianou, Andreas C and Neil D Lawrence (2013). "Deep Gaussian Processes." In: *AISTATS*, pp. 207–215.

Dantzig, George (2016). *Linear programming and extensions*. Princeton University Press.

Dantzig, George B and B Curtis Eaves (1973). "Fourier-Motzkin elimination and its dual". In: *Journal of Combinatorial Theory, Series A* 14.3, pp. 288–297.

Dantzig, George B and Mukund N Thapa (2006a). *Linear programming 1: introduction*. Springer Science & Business Media.

— (2006b). *Linear programming 2: theory and extensions*. Springer Science & Business Media.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, pp. 248–255.

Duane, Simon, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth (1987). "Hybrid Monte Carlo". In: *Physics Letters B* 195.2, pp. 216–222.

Durbin, Richard and David E Rumelhart (1989). "Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks". In: *Neural Computation* 1, pp. 133–142.

Eisenach, Carson, DHan Liu, and Zhaoran Wang (2017). "Nonparametrically Learning Activation Functions in Deep Neural Nets". In: *International Conference on Learning Representations*.

Farzad (2013). *CUDA atomic operation performance in different scenarios*. URL: https://stackoverflow.com/questions/22367238/cuda-atomic-operation-performance-in-different-scenarios.

Fischer, Hans (2010). *A history of the central limit theorem: From classical to modern probability theory*. Springer Science & Business Media.

Fox, Charles W. and Stephen J. Roberts (2012). "A tutorial on variational Bayesian inference". In: *Artificial Intelligence Review* 38.2, pp. 85–95.

Frey, Peter W and David J Slate (1991). "Letter Recognition Using Holland-Style Adaptive Classifiers". In: *Machine Learning* 6, p. 161.

Georgii, Hans-Otto (2015). *Stochastik: Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Walter de Gruyter GmbH & Co KG.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*. Ed. by Geoffrey J. Gordon and David B. Dunson. Vol. 15. Journal of Machine Learning Research - Workshop and Conference Proceedings, pp. 315–323.

Goodfellow, Ian J, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio (2013). "Maxout networks". In: *arXiv preprint arXiv:1302.4389*.

Graves, Alex (2011). "Practical variational inference for neural networks". In: *Advances in Neural Information Processing Systems*, pp. 2348–2356.

Griebel, Michael, Stephan Knapek, Gerhard Zumbusch, and Attila Caglar (2003). *Numerische Simulation in der Moleküldynamik: Numerik, Algorithmen, Parallelisierung, Anwendungen (Springer-Lehrbuch) (German Edition)*. Springer.

Griewank, Andreas and Andrea Walther (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Gybenko, G (1989). "Approximation by superposition of sigmoidal functions". In: *Mathematics of Control, Signals and Systems* 2.4, pp. 303–314.

Haykin, Simon (1994). *Neural networks: a comprehensive foundation*. Prentice Hall PTR.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

Hecht-Nielsen, Robert et al. (1988). "Theory of the backpropagation neural network." In: *Neural Networks* 1.Supplement-1, pp. 445–448.

Hinton, Geoffrey E (2007). "To recognize shapes, first learn to generate images". In: *Progress in brain research* 165, pp. 535–547.

Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). "Reducing the dimensionality of data with neural networks". In: *science* 313.5786, pp. 504–507.

Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580*.

Hinton, Geoffrey E and Drew Van Camp (1993). "Keeping the neural networks simple by minimizing the description length of the weights". In: *Proceedings of the sixth annual conference on Computational learning theory*. ACM, pp. 5–13.

Hoeting, Jennifer A, David Madigan, Adrian E Raftery, and Chris T Volinsky (1999). "Bayesian model averaging: a tutorial". In: *Statistical science*, pp. 382–401.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5, pp. 359–366.

Inc, Wolfram Research (2017). *Mathematica Version 11.2*.

Iverson, Kenneth E (1962). "A programming language". In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM, pp. 345–351.

Julier, Simon J and Jeffrey K Uhlmann (1996). *A general method for approximating nonlinear transformations of probability distributions*. Tech. rep. Robotics Research Group, Department of Engineering Science, University of Oxford.

— (1997). "New extension of the Kalman filter to nonlinear systems". In: *AeroSense'97*. International Society for Optics and Photonics, pp. 182–193.

Khamsi, Mohamed A. and William A. Kirk (2001). *An Introduction to Metric Spaces and Fixed Point Theory*. Wiley-Interscience.

Kingma, Diederik P and Max Welling (2013). "Auto-encoding variational Bayes". In: *arXiv preprint arXiv:1312.6114*.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Kjolstad, Fredrik, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe (2017). "The Tensor Algebra Compiler". In: *Proc. ACM Program. Lang.* 1.OOPSLA, 77:1–77:29.

Klambauer, Günter, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter (2017). "Self-Normalizing Neural Networks". In: *arXiv preprint arXiv:1706.02515*.

Kneser, H (1950). "Reelle analytische Lösungen der Gleichung $\varphi(\varphi(x)) = e^x$ und verwandter Funktionalgleichungen." In: *Journal für die reine und angewandte Mathematik*, pp. 56–67.

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kohavi, Ron (1996). "Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-tree Hybrid". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, pp. 202–207.

Köpp, Wiebke (2015). "A Novel Transfer Function for Continuous Interpolation between Summation and Multiplication in Neural Networks". Technical University Munich.

— (2016). "A Novel Transfer Function for Continuous Interpolation between Summation and Multiplication in Neural Networks". KTH, School of Computer Science and Communication (CSC).

Köpp, Wiebke, Patrick van der Smagt, and Sebastian Urban (2016). "A Differentiable Transition Between Additive and Multiplicative Neurons". In: *International Conference on Learning Representations, ICLR 2016*.

Krizhevsky, Alex and Geoffrey Hinton (2009). *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto.

Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton (2014). *The CIFAR-10 dataset*. URL: http://www.cs.toronto.edu/kriz/cifar.html.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., pp. 1097–1105.

Lange, Kenneth (2013). *Optimization (Springer Texts in Statistics)*. Springer.

Lauritzen, Steffen L (1996). *Graphical models*. Vol. 17. Clarendon Press.

Lawrence, Neil (2005). "Probabilistic non-linear principal component analysis with Gaussian process latent variable models". In: *Journal of Machine Learning Research* 6.Nov, pp. 1783–1816.

LeCun, Yann A, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller (2012). "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, pp. 9–48.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel (1990). "Handwritten digit recognition with a back-propagation network". In: *Advances in neural information processing systems*, pp. 396–404.

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551.

Lecun, Yann and Corinna Cortes (1998). *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

Lehmann, E.L. (2004). *Elements of Large-Sample Theory (Springer Texts in Statistics)*. Corrected. Springer.

Lichman, M. (2013). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. ICML*. Vol. 30. 1.

Maniezzo, Vittorio (1994). "Genetic evolution of the topology and weight distribution of neural networks". In: *IEEE Transactions on neural networks* 5.1, pp. 39–53.

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL: http://tensorflow.org/.

Memisevic, Roland (2011). "Learning to relate images: Mapping units, complex cells and simultaneous eigenspaces". In: *arXiv preprint arXiv:1110.0107*.

— (2013). "Learning to relate images". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1829–1846.

Metropolis, Nicholas, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller (1953). "Equation of state calculations by fast computing machines". In: *The journal of chemical physics* 21.6, pp. 1087–1092.

Minka, Thomas P (2001). “Expectation propagation for approximate Bayesian inference”. In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 362–369.

Murray, Iain (2016). “Differentiation of the Cholesky decomposition”. In: *arXiv preprint arXiv: 1602.07527*.

Nair, Vinod and Geoffrey E. Hinton (2010). “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, pp. 807–814.

Neal, Radford M (1997). “Monte Carlo implementation of Gaussian process models for Bayesian regression and classification”. In: *arXiv preprint physics/9701026*.

Neal, Radford M et al. (2011). “MCMC using Hamiltonian dynamics”. In: *Handbook of Markov Chain Monte Carlo* 2, pp. 113–162.

Nvidia (2017). *NVIDIA CUDA C Programming Guide*. Version 9.0. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

O’Hagan, Anthony and JFC Kingman (1978). “Curve fitting and optimal design for prediction”. In: *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–42.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling (1992). *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press.

Quarteroni, Alfio (2000). *Numerical mathematics*. New York: Springer.

Quiñonero-Candela, Joaquin, Agathe Girard, and Carl Edward Rasmussen (2002). *Prediction at an uncertain input for Gaussian processes and relevance vector machines-application to multiple-step ahead time-series forecasting*. Tech. rep. IMM, Danish Technical University.

Quiñonero-Candela, Joaquin and Carl Edward Rasmussen (2005). “A unifying view of sparse approximate Gaussian process regression”. In: *Journal of Machine Learning Research* 6.Dec, pp. 1939–1959.

Rall, Louis B. (1981). *Automatic Differentiation: Techniques and Applications*. Vol. 120. Lecture Notes in Computer Science. Berlin: Springer.

Rasmussen, Carl Edward and Christopher KI Williams (2006). *Gaussian processes for machine learning*. Vol. 1. MIT press Cambridge.

Rechenberg, I (1973). *Evolutionsstrategie–Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog.

Al-Rfou, Rami, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688.

Riedmiller, Martin and Heinrich Braun (1992). "RPROP-A fast adaptive learning algorithm". In: *Proc. of ISCIS VII*.

Riihimäki, Jaakko and Aki Vehtari (2010). "Gaussian processes with monotonicity information". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS-10)*. Ed. by Yee W. Teh and D. M. Titterington. Vol. 9, pp. 645–652.

Rumelhart, David E, Geoffrey E Hinton, Ronald J Williams, et al. (1988). "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3, p. 1.

Rumelhart, David E, James L McClelland, PDP Research Group, et al. (1987). *Parallel distributed processing*. Vol. 1. MIT press Cambridge, MA, USA:

Salakhutdinov, Ruslan and Geoffrey Hinton (2009). “Deep boltzmann machines”. In: *Artificial Intelligence and Statistics*, pp. 448–455.

Salimbeni, Hugh and Marc Deisenroth (2017). “Doubly Stochastic Variational Inference for Deep Gaussian Processes”. In: *arXiv preprint arXiv:1705.08933*.

Scardapane, Simone, Steven Van Vaerenbergh, and Aurelio Uncini (2017). “Kafnets: kernel-based non-parametric activation functions for neural networks”. In: *arXiv preprint arXiv:1707.04035*.

Schröder, Ernst (1870). “Ueber iterirte Functionen”. In: *Mathematische Annalen* 3, pp. 296–322.

Shin, Y. and J. Ghosh (1991). “The pi-sigma network: an efficient higher-order neural network for pattern classification and function approximation”. In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. i, 13–18 vol.1.

Smith, HJ Stephen (1860). “On Systems of Linear Indeterminate Equations and Congruences.” In: *Proceedings of the Royal Society of London* 11, pp. 86–89.

Smith, Stephen P (1995). “Differentiation of the Cholesky algorithm”. In: *Journal of Computational and Graphical Statistics* 4.2, pp. 134–147.

Solak, Ercan, Roderick Murray-Smith, William E Leithead, Douglas J Leith, and Carl E Rasmussen (2003). “Derivative observations in Gaussian process models of dynamic systems”. In: *Advances in neural information processing systems*, pp. 1057–1064.

Srivastava, Nitish, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of machine learning research* 15.1, pp. 1929–1958.

Stanley, Kenneth O (2007). “Compositional pattern producing networks: A novel abstraction of development”. In: *Genetic programming and evolvable machines* 8.2, pp. 131–162.

Stanley, Kenneth O, David B D’Ambrosio, and Jason Gauci (2009). “A hypercube-based encoding for evolving large-scale neural networks”. In: *Artificial life* 15.2, pp. 185–212.

Stanley, Kenneth O and Risto Miikkulainen (2002). “Evolving neural networks through augmenting topologies”. In: *Evolutionary computation* 10.2, pp. 99–127.

Strassen, Volker (1969). “Gaussian elimination is not optimal”. In: *Numerische mathematik* 13.4, pp. 354–356.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge.

Sutton, Richard S, David A McAllester, Satinder P Singh, and Yishay Mansour (2000). "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*, pp. 1057–1063.

Swokowski, Earl William (1979). *Calculus with analytic geometry*. Taylor & Francis.

Taylor, John (1997). *Introduction to error analysis, the study of uncertainties in physical measurements*. University Science Books.

Tieleman, T. and G. Hinton (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning.

Tikhonov, Andrei Nikolaevich, Vasilii Iakkovlevich Arsenin, and Fritz John (1977). *Solutions of ill-posed problems*. Vol. 14. Winston Washington, DC.

Titsias, Michalis K (2009). "Variational learning of inducing variables in sparse Gaussian processes". In: *International Conference on Artificial Intelligence and Statistics*, pp. 567–574.

Urban, Sebastian, Justin Bayer, Christian Osendorfer, Goran Westling, Benoni B Edin, and Patrick Van Der Smagt (2013). "Computing grip force and torque from finger nail images using gaussian processes". In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, pp. 4034–4039.

Urban, Sebastian, Marvin Ludersdorfer, and Patrick Van Der Smagt (2015). "Sensor Calibration and Hysteresis Compensation with Heteroscedastic Gaussian Processes". In: *IEEE Sensors Journal* 15.11, pp. 6498–6506.

Urban, Sebastian and Patrick van der Smagt (2015). "A Neural Transfer Function for a Smooth and Differentiable Transition Between Additive and Multiplicative Interactions". In: *arXiv preprint arXiv:1503.05724*.

Vapnik, Vladimir (2013). *The nature of statistical learning theory*. Springer science & business media.

Wang, Sida I. and Christopher D. Manning (2013). "Fast dropout training". In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28, pp. 118–126.

Wegert, E. (2012). *Visual Complex Functions: An Introduction with Phase Portraits*. Springer Basel.

Werbos, Paul J (1990). "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10, pp. 1550–1560.

Williams, Christopher KI and Carl Edward Rasmussen (1996). “Gaussian processes for regression”. In: *Advances in neural information processing systems*, pp. 514–520.

— (2006). *Gaussian processes for machine learning*. MIT Press.

Williams, Ronald J (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4, pp. 229–256.

Wu, Huaiqin (2009). “Global stability analysis of a general class of discontinuous neural networks with linear growth activation functions”. In: *Information Sciences* 179.19, pp. 3432–3441.

Xu, Bing, Naiyan Wang, Tianqi Chen, and Mu Li (2015). “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853*.

yidiyidawu (2012). *CUDA performance of atomic operation on different address in warp*. URL: https://stackoverflow.com/questions/22342685/cuda-performance-of-atomic-operation-on-different-address-in-warp.

Zeiler, Matthew D (2012). “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701*.

Zoph, Barret and Quoc V Le (2016). “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578*.

List of Acronyms

ANN artificial neural network

ARD automatic relevance determination

CDF cumulative density function

CNN convolutional neural network

CUDA Compute Unified Device Architecture

DFT discrete Fourier transform

ELBO evidence lower bound

GCD greatest common divisor

GP Gaussian process

GPN Gaussian process neuron

GPU graphics processing unit

HMC Hamiltonian Monte Carlo

iid. independent and identical distributed

MCMC Markov chain Monte Carlo

PDF probability density function

RNN recurrent neural network

SE squared exponential

Neural Network Architectures and Activation Functions: A Gaussian Process Approach

The success of applying neural networks crucially depends on the network architecture being appropriate for the task. Determining the right architecture is a computationally intensive process, requiring many trials with different candidate architectures. We show that the neural activation function, if allowed to individually change for each neuron, can implicitly control many aspects of the network architecture, such as effective number of layers, effective number of neurons in a layer, skip connections and whether a neuron is additive or multiplicative.

Motivated by this observation we propose stochastic, non-parametric activation functions that are fully learnable and individual to each neuron. Complexity and the risk of overfitting are controlled by placing a Gaussian process prior over these functions. The result is the Gaussian process neuron, a probabilistic unit that can be used as the basic building block for probabilistic graphical models that resemble the structure of neural networks. The proposed model can intrinsically handle uncertainties in its inputs and self-estimate the confidence of its predictions. Using variational Bayesian inference and the central limit theorem, a fully deterministic loss function is derived, allowing it to be trained as efficiently as a conventional neural network using stochastic gradient descent. The posterior distribution of activation functions is inferred from the training data alongside the weights of the network. The proposed model favorably compares to deep Gaussian processes, both in model complexity and efficiency of inference. It can be directly applied to recurrent or convolutional network structures, allowing its use in audio and image processing tasks. As an empirical evaluation we present experiments on regression and classification tasks, in which our model achieves performance comparable to or better than a Dropout regularized neural network.

We further develop a novel method for automatic differentiation of elementwise-defined, tensor-valued functions that occur in the mathematical formulation of Gaussian processes. The proposed method allows efficient evaluation of the derivatives on modern GPUs and is used in our implementation of the Gaussian process neuron to achieve computational performance that is about 25% of a convential neuron with a fixed logistic activation function.