CBS
COPENHAGEN
BUSINESS SCHOOL
HANDELSHØJSKOLEN

Bayesian Neural Networks

THEORY AND APPLICATIONS

BY

MAGNUS RAABO ANDERSEN (110307) & ULRIK ROED-SØRENSEN (111931)

SUPERVISOR: PETER DALGAARD

MASTER'S THESIS

COPENHAGEN BUSINESS SCHOOL — CAND.MERC.(MAT.)

DATE OF SUBMISSION: MAY 17 2021

NUMBER OF PAGES: 103 (155732 characters with spaces)

Abstract

This thesis examines Bayesian inference in neural networks with Markov chain Monte Carlo sampling for performing regression and classification, and how this is different from using non-Bayesian feedforward neural networks. Initially basic machine learning theory for supervised learning algorithms is outlined. This theory is subsequently used for examining how neural networks work, and how they can be trained and regularized. This provides the fundamentals for introducing Bayesian inference and how it can be used for neural networks. The Markov chain Monte Carlo based Bayesian neural networks will be the focal point for this analysis, and we examine the most popular sampling methods for these, while only covering the fundamentals on the choice of prior distributions. Through implementation of different neural networks and Bayesian neural networks we illustrate and evaluate how these perform when predicting house prices using regression and predicting probabilities for default of credit card clients for binary classification.

Keywords: Machine Learning, Neural Networks, Bayesian Neural Network, Deep Learning, TensorFlow, Markov Chain Monte Carlo, Python, Computational Statistics, Hamiltonian Monte Carlo, No-U-Turn Sampler, Bayesian Inference, Mathematics of Computing, Probability and Statistics, Bayesian Deep Learning, Stochastic Neural Network, PyMC3.

Resumé

Denne afhandling undersøger Bayesiansk-inferens i neurale netværk med Markov-kæde Monte Carlo-sampling til udførelse af regression og klassifikation, og hvordan dette er forskelligt fra at bruge ikke-Bayesianske feedforward neurale netværk. Først beskrives grundlæggende machine learning teori for supervised læringsalgoritmer. Denne teori bruges efterfølgende til at undersøge, hvordan neurale netværk fungerer, og hvordan de kan trænes og reguleriseres. Dette giver de grundlæggende forudsætninger for introduktion af Bayesiansk-inferens i neurale netværk. Markov-kæde Monte Carlo-baserede Bayesianske neurale netværk vil være omdrejningspunktet for denne analyse, og vi undersøger de mest populære sampling-metoder indenfor denne teori, mens der kun dækkes den grundlæggende teori om valget af prior fordelinger. Gennem implementering af forskellige neurale netværk og Bayesianske neurale netværk illustrerer og vurderer vi, hvordan disse fungerer ved prædiktion af huspriser ved hjælp af regression og prædiktion af sandsynligheder for misligholdt gæld af kreditkortklienter til binær klassifikation.

Nøgleord: Machine Learning, Neural Networks, Bayesian Neural Network, Deep Learning, TensorFlow, Markov Chain Monte Carlo, Python, Computational Statistics, Hamiltonian Monte Carlo, No-U-Turn Sampler, Bayesian Inference, Mathematics of Computing, Probability and Statistics, Bayesian Deep Learning, Stochastic Neural Network, PyMC3.

Acknowledgements

We owe a special thank you to our supervisor Peter Dalgaard for help and guidance, we are grateful that you have led us in the right direction. Furthermore, we would like to thank Eric J. Ma for increasing our interest in Bayesian neural networks and for being helpful with several questions along the way. We thank Caroline Enersen for providing us with food and sweets. We also thank our families for their love and support. Last but not least we would like to thank Rasmus Nielsen Kollegiet for providing us with a ping pong table for when we needed a break.

Contents

List of Figures

List of Tables

List of Algorithms

Notation

This section provides a concise reference describing the notation used throughout this thesis.

Datasets and Distributions

$\boldsymbol{X}$ A $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

$\boldsymbol{x}^{(i)}$ The ith (input) example from a dataset

$\mathbb{H}$ A set of hypothesis also called the hypothesis space

$\mathbb{X}$ A set of training examples

$\mathbb{Y}$ A set of target labels

$y^{(i)}$ or $\boldsymbol{y}^{(i)}$ The target associated with $\boldsymbol{x}^{(i)}$

S A data set or sample

Functions

$\mathbf{1}$ [condition] Equal to 1 if condition is true, 0 otherwise

$\sigma(x)$ Logistic sigmoid function $\frac{1}{1+e^{-x}}$

Indexing

a_i Element i of the random vector $\mathbf{a}$

a_i Element i of vector $\boldsymbol{a}$, with indexing starting at 1

$A_{i,j}$ Element i, j of matrix $\boldsymbol{A}$

Linear Algebra Operations

$\boldsymbol{A}^\top$ Transpose of matrix $\boldsymbol{A}$

Numbers and Arrays

$\boldsymbol{A}$ A Matrix

$\boldsymbol{a}$ A Vector

$\boldsymbol{I}_n$ Identity matrix with n rows and n columns. Sometimes n is omitted and implied by context

$\mathbf{A}$ A matrix of random variables

$\mathbf{a}$ A vector of random variables

a A scalar random variable

a A scalar (integer or real)

Probability and Information Theory

$\mathbb{E}_{\mathrm{x}\sim P}[f(\mathrm{x})]$ or $\mathbb{E}[f(\mathrm{x})]$ Expectation of $f(\mathrm{x})$ with respect to distribution $P(\mathrm{x})$. Sometimes subtext $\mathrm{x} \sim P$ is omitted and implied by context

$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

$\mathrm{a} \sim P$ Random variable a has distribution P

$\mathrm{Cov}(f(\mathrm{x}, g(\mathrm{x})))$ Covariance of $f(\mathrm{x})$ and $g(\mathrm{x})$ under distribution $P(\mathrm{x})$

$\mathrm{Var}(f(\mathrm{x}))$ Variance of $f(\mathrm{x})$ under distribution $P(\mathrm{x})$

$P(\mathrm{a})$ A probability distribution over a discrete variable

$p(\mathrm{a})$ A probability distribution over a continuous variable, or over a variable whose type has not been specified

Sets and Graphs

$(a, b]$ The real interval between a and b excluding a but including b

$[a, b]$ The real interval between and including a and b

$\mathbb{A}$ A set

$\mathbb{R}$ The set of real numbers

$\{0, 1\}$ The set containing 0 and 1

$\{0, 1, \ldots, n\}$ The set containing all integers between 0 and n

Chapter 1

Introduction

Neural networks have led to a revolution in machine learning, providing solutions to tackle more challenging and complex problems. However neural networks are prone to overfitting, which can negatively affect their generalization capabilities. Neural networks also tend to be overconfident about their predictions, even when they do provide a confidence interval. All of this can be problematic for applications such as medical diagnostics, self driving cars or finance, where wrong predictions can have dramatic outcomes.

To moderate this risk many approaches have been proposed, most noticeably by the use of stochastic neural networks to estimate the uncertainty of model prediction. The Bayesian paradigm provides a solid framework to analyse and develop these learning algorithms. Such Bayesian neural networks have become more attractive in the recent years with the increase in computational power of personal computers and the growing body of research on efficient sampling algorithms. One branch of such algorithms is the Markov chain Monte Carlo sampling methods, which provide clever ways of sampling to reduce computational cost. Also a growing number of programming languages and software packages, such as Stan, TensorFlow Probability and PyMC3, support Bayesian statistical analysis.

Another factor that makes Bayesian neural networks interesting is that they allow us to interpret predictions in terms of probability distributions, which traditional approaches do not. Further it provides a principled mechanism for practitioners to build on previous knowledge by incorporating relevant prior information into their analysis. This can be cru-

cial for tasks with small sample sizes, since the prior information regularize results, reducing the chances of poor generalization capabilities, which is common in cases with a relatively small sample size.

1.1 Problem Statement

This thesis aims to answer the following problem statement:

How can Markov chain Monte Carlo based Bayesian neural networks be used to perform regression and classification tasks and how is this different from using non-Bayesian feedforward neural networks?

This is done by answering the following research questions:

- What is a supervised machine learning algorithm and how does it work?
- What is over- and underfitting and how do we prevent this in a supervised machine learning algorithm?
- How is gradient based optimization used in training a supervised machine learning algorithm?
- What is a feedforward neural network and how is it trained and regularized?
- What is Bayesian inference and how is it different from frequentist inference?
- What is Monte Carlo methods and how do they work?
- What is a Bayesian neural network and how does it work?
- What is Markov chain Monte Carlo and how can we use it to sample?
- What is Hamiltonian Monte Carlo and how can we use it to sample?
- What is No-U-Turn Hamiltonian Monte Carlo and how can we use it to sample?
- What role do priors play in Bayesian neural networks and how can we choose them properly?
- How do non-Bayesian and Bayesian neural networks perform when used for predicting house prices in Boston?
- How do non-Bayesian and Bayesian neural networks perform when used for predicting default probabilities on credit card loans?

1.2 Method

We approach the questions posed in section 1.1 using the positivist paradigm. Positivism is a philosophical theory that states that genuine knowledge exists and is structured in a specific way regardless of ones perspective of it. According to this theory, information is derived from sensory experienced, that is interpreted through reason and logic, which is reflected by our mathematical approach of examining Bayesian- and non-Bayesian neural networks. We evaluate these neural networks on data, which is in accordance with a positivist view on verified data as empirical evidence.

This thesis is mainly a theoretical examination of Markov chain Monte Carlo based Bayesian neural networks, with a focus on efficient sampling methods for these, and how they differ from non-Bayesian neural networks. We examine this by first clarifying basic machine learning theory that is relevant for neural networks. This will cover how to train such supervised machine learning algorithms and how to regularize them to prevent over- and underfitting. Afterwards we cover numerous optimization algorithms used for training, which provide the foundations for examining Adaptive Moment Estimation (ADAM), that we use in our practical evaluation in a later section. This provides the fundamentals for examining feed-forward neural networks. We do this by covering its general structures and various options for designing it. We also cover how it trains using backpropagation combined with the previous covered optimization algorithms, and we cover how to do this with regularization techniques like early stopping.

With neural networks examined we have defined the model to which we take a Bayesian approach in Bayesian neural networks. Such networks are covered by first examining the difference in Bayesian and frequentist (read non-Bayesian) learning. Then we examine Monte Carlo methods, which are often used in Bayesian inference models.

We then construct a simple Bayesian neural network, to illustrate how such a network works, and to show the motivation for introducing more sophisticated sampling techniques when sampling from such networks. This naturally leads to an examination of Markov chain Monte Carlo methods, which provides the basics for such sampling methods. We use this knowledge to examine the Hamiltonian Monte Carlo method and the extensions of this

in the pursuit of the most efficient sampling method for Bayesian neural networks. This pursuit ends with the No-U-Turn Hamiltonian Monte Carlo, which we use in our practical evaluation of Bayesian neural networks in a later section.

Afterwards we examine the effect of prior choice and only briefly how to choose priors, as such choices often depend on the specific task and data. Instead we examine various general approaches and point to literature that supports these approaches. Finally we evaluate the Bayesian and non-Bayesian neural networks on real data. First a dataset for which we aim to predict house prices in Boston using regression and next a dataset on credit card defaults, where we try to predict default probabilities for binary classification.

1.3 Research Delimitation

To have a focused approach for answering our problem statement in section 1.1, we choose not to elaborate on certain subjects, as we deem them out of scope for answering the problem statement appropriately. As we focus on Markov chain Monte Carlo based Bayesian neural networks, we do not examine other methods for using Bayesian inference in neural networks such as methods based on variational inference. We also refrain from examining the choice of priors too thoroughly, and we only cover the consequences of the choice and the most general methods, since the method of choosing a prior often depends on the data, the task and the researches prior knowledge of such.

Non-Bayesian neural networks are examined to the extent needed in order to provide fundamental knowledge to the model for which we use Bayesian inference and to illustrate the most important differences. These differences are important to illustrate pros and cons of using the Bayesian approach to these networks. This means that we examine the most basic neural networks called feedforward neural networks and do not go into more sophisticated architecture like recurrent or convolutional neural networks. We also limit our examination to the most popular activation functions and training procedures for these networks.

Chapter 2

Machine Learning Basics

This chapter provides a brief introduction to the fundamental theory of machine learning that is used throughout the rest of this thesis. Machine learning is the study of computer algorithms that learn by analyzing data. Most machine learning algorithms can be divided into the two categories of supervised learning and unsupervised learning. In the following, we will describe the informal distinction between these and the most common tasks used for supervised learning. As neural networks are considered supervised learning the following sections will focus only on theory for supervised learning algorithms.

We proceed to describe the challenge of finding patterns and generalizing these to new data while describing the various machine learning components such as capacity, loss functions and regularization. Most machine learning algorithms are based on the idea of minimizing a loss function by using an optimization algorithm to select the optimal parameters. One of the most common optimization algorithms used in the context of machine learning is called stochastic gradient descent. We cover this algorithm and its variants in in section 2.6. We will later use the theory of stochastic gradient descent in combination with backpropagation in section 3.2 for describing learning in neural networks.

2.1 Supervised Learning

Supervised learning use labels on datapoints, which we call examples, as targets to create prediction rules for learning the label of new examples. In unsupervised learning useful

properties of the data structure are learned without the model being provided with target labels to predict. However as mentioned by Goodfellow et al. (2016) the distinction is not formally defined, since there is no objective way to determine whether a value is a feature or a target label.

One example of a learning algorithm that is typically viewed as supervised is classic linear regression that uses examples x and their labels y to create a linear function to determine y-values of new examples x. An example of a learning algorithm that is typically viewed as unsupervised is k-means clustering, that divides a dataset into k clusters based on distance.

As neural networks and Bayesian neural networks use target labels to predict labels of new examples it is considered a supervised learning algorithm. Therefore the following chapter will only cover the machine learning basics of supervised algorithms.

Typical tasks of supervised machine learning algorithms are:

- Classification when the output space $\mathbb{Y}$ are labels. This is called binary classification in cases with only two possible labels. In this case the algorithm must learn to separate examples of these two labels. Binary labels (be it male/female, yes/no etc.) are typically translated to a numerical representation by letting $\mathbb{Y} = \{\pm 1\}$ or $\mathbb{Y} = \{0, 1\}$. When working with more than two, but still a finite number of labels, we call the task multiclass classification. Depending of the setting and amount of labels it can be preferable to use regression instead of multiclass classification.
- Regression when the output space $\mathbb{Y} = \mathbb{R}$. For example predicting someones weight could be modeled as a regression problem.
- Structured prediction when the output is not merely a number or a label, but a structured object. An example of this is machine translation where the input is a sentence in one language and the output a sentence in another language.

2.2 Loss Functions

In order to evaluate the performance of a supervised machine learning algorithm we use a quantitative measure called a loss function. The loss function is an encoding, of how

much the algorithm should care about making certain kinds of mistakes, and it is based on this measure that the algorithm selects a hypothesis h from a set of possible hypothesis $\mathbb{H}$ called the hypothesis space. A hypothesis (also called prediction rule) can be understood as a recipe, that the algorithm develops to perform its task. An example would be the weights $\boldsymbol{w}$ for performing the linear regression $\hat{y}^{(i)} = \boldsymbol{w}^\top \boldsymbol{x}^{(i)}$. Another example would be a specific range of values for a specific set of features for classification, like a set of RGB color-values to classify if an input-image is depicting an apple or a pear. We treat the hypothesis as a function $h\left(\boldsymbol{x}^{(i)}\right)$ taking $\boldsymbol{x}^{(i)}$ as input and predicting $\hat{y}^{(i)}$. We define the loss function $\ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$ as being the loss of predicting $h\left(\boldsymbol{x}^{(i)}\right)$ when the target label is $y^{(i)}$.

We want the algorithm to return the "best" hypothesis h^*, which in this context should be interpreted as the hypothesis that gives the least amount of expected loss on new samples $\left(\mathbf{x}_{\text{new}}^{(i)}, \mathrm{y}_{\text{new}}^{(i)}\right)$

$$L(h) \equiv \mathbb{E}\left[\ell\left(h\left(\mathbf{x}_{\text{new}}^{(i)}\right), \mathrm{y}_{\text{new}}^{(i)}\right)\right]$$

where the expectation is taken with respect to the data distribution p_{data}. If such a value is satisfyingly low, it means that h^* is a useful tool for performing its task on new data. We define the optimal hypothesis formally by

$$h^* = \arg\min_{h \in \mathbb{H}} L(h)$$

Common loss functions in regression are squared loss (also called L2 loss)

$$\ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) = \left(h\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2 \tag{2.2.1}$$

or absolute loss (also called L1 loss)

$$\ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) = \left|h\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right|$$

These functions train the algorithm to create hypothesis that minimize the differences from the target labels.

A typical loss function used for binary classification is the zero-one loss

$$\ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) = \mathbb{1}\left(h\left(\boldsymbol{x}^{(i)}\right) \neq y^{(i)}\right) = \begin{cases} 0, & \text{if } h\left(\boldsymbol{x}^{(i)}\right) = y^{(i)} \\ 1, & \text{otherwise} \end{cases}$$

giving 0 loss for when the predicted label $h\left(\boldsymbol{x}^{(i)}\right)$ is equal to the true label $y^{(i)}$ and 1 otherwise. This loss function trains the algorithm to generate hypothesis, that makes the least number of mistakes.

Another classification loss function that do not just look at the predicted label, but at the probability of a certain prediction, is the cross-entropy loss or log-loss function. This assumes that instead of predicting the label, the hypothesis $h\left(\boldsymbol{x}^{(i)}\right)$ generated by the model is a set of probabilities that each provide the probability that the example belong to each of the classes. For binary classification it is defined as

$$\ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) = -\left(y^{(i)} \log h\left(\boldsymbol{x}^{(i)}\right) + \left(1 - y^{(i)}\right)\right) \log\left(1 - h\left(\boldsymbol{x}^{(i)}\right)\right) \tag{2.2.2}$$

which is actually the negative log likelihood (so minimizing this maximizes likelihood, see section 4.1.1 for maximum likelihood estimation) when assuming Bernoulli distribution for the targets. For multiclass classification this can be generalized by assuming multinomial distribution with one trail for the targets and taking the negative log likelihood. These probabilities are typically provided by a sigmoid or softmax function that maps $\mathbb{R} \to [0, 1]$, these are explained further in section 3.1. With a model, that outputs these probabilities as hypothesis, predictions are typically done by choosing the class for which the example has the highest probability of belonging to.

These are some of the convenient general choices, and there are many more. But such general loss functions is not necessarily the right choice for a particular application. For example if a company knows the exact monetary loss of packing a wrong item as a result of a misclassification happening in a robotic packaging system. Then it might be beneficial to use a table (perhaps formally written as a sum of indicatorfunctions) of the costs of packaging each of their items as a loss function instead of using the general ones found in the literature.

2.3 Training & Testing

The central challenge in machine learning is making sure that the algorithm creates a hypothesis that perform well on new data that was not used in selecting the hypothesis. As mentioned in section 2.2 this is done by minimizing $L(h)$, but as we do not know p_{data}

(if we did we wouldn't need the algorithm) we can not evaluate $L(h)$. We must instead approximate $L(h)$ by its empirical estimate

$$\hat{L}(h,S) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(h\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

However when we select a hypothesis $\hat{h}_S^*$ in $\mathbb{H}$ based on empirical loss $\hat{L}\left(h,S\right)$ then the loss of this hypothesis $\hat{h}_S^*$ becomes a biased estimate of $L(\hat{h}_S^*)$. This is because $\hat{h}_S^*$ is selected based on the minimum empirical error on S, so from the perspective of $\hat{h}_S^*$ new samples might not be interchangeable with samples in S, since including these new samples could result in a different hypothesis that minimizes the loss on S.

To get an unbiased estimate of $L(\hat{h}_S^*)$ for evaluating a model it is common practice to split the sample set S into a dataset S_{train} and a test set S_{test}. One can then find the best hypothesis using the training set h^*_{train} and afterwards use the test set for computation of $\hat{L}\left(h^*_{\text{train}}, S_{\text{test}}\right)$ to evaluate the performance. Based on the assumption that new samples $\left(\mathbf{x}^{(i)}_{\text{new}}, \mathrm{y}^{(i)}_{\text{new}}\right)$ are distributed identically with the samples in S_{test} then these new samples are exchangeable with the ones in S_{test} from the perspective of h^*_{train}. This means that $\mathbb{E}\left[\ell\left(h^*_{\text{train}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)\right] = \mathbb{E}\left[\ell\left(h^*_{\text{train}}\left(\mathbf{x}^{(i)}_{\text{new}}\right), \mathrm{y}^{(i)}_{\text{new}}\right)\right]$ and therefore $\hat{L}\left(h^*_{\text{train}}, S_{\text{test}}\right)$ is an unbiased estimate of $L(h^*_{\text{train}})$.

2.3.1 Cross-Validation

Splitting a dataset into a training set and a test set can be problematic if the resulting test set is too small. Such a small test set will give rise to statistical uncertainties around the estimated average loss and will make it problematic to evaluate the model. When we have large dataset this is not a problem, but when we work with small datasets it can become a serious issue.

To circumvent this issue is to use cross-validation. Cross-validation repeats the training and testing procedure on different randomly chosen splits of the original dataset. This enables us to use more examples for estimating the average test loss for the cost of computational power. This approach is also useful for choosing hyperparameters like regularization constants α covered in section 2.5.

A common method for cross-validation is the K-fold cross-validation that splits the dataset into K non-overlapping and roughly equally sized subsets. On trial i the ith subset is used as the test set while the rest $K-1$ subsets are used in the training set. The test loss is then estimated by the cross-validation loss computed as the average test loss across the K trials.

One problem with this approach is however measuring the uncertainty since there is no unbiased estimators of the variance of the average error estimators as mentioned by Bengio & Grandvalet (2004), one must instead use approximations. Another discussed issue is how to choose the value for K. This problem does not have a clear answer but 5- or 10-fold cross-validation are generally recommended as a good compromise, see Breiman & Spector (1992) and Kohavi (2001).

2.4 Overfitting & Underfitting

The average loss attained on the training set when selecting the best hypothesis $\hat{L}\left(h^*_{\text{train}}, S_{\text{train}}\right)$ is what we will call the training error or training loss. The average loss attained from the test set $\hat{L}\left(h^*_{\text{train}}, S_{\text{val}}\right)$, used for testing the model, is what we will call test error or test loss.

Having a low training error means that we have made a prediction rule that fit our training set well. Having a small gap between training and test error means that the prediction rule generalizes well on new data, which is why the test error is also sometimes called generalization error. These factors are what corresponds to underfitting and overfitting, which is used to measure performance of the model.

Underfitting occurs when the model is not able to obtain a sufficiently low training error. This is seen in figure 2.1 in the case of linear regression of degree 1, where the model fitted on the training data lies far from the training samples, meaning it must have a large mean squared error (MSE) on the training data. Overfitting occurs when the gap between the training error and test error is too large. This large gap indicates that the model faithfully reflects idiosyncrasies of the training data rather than the true underlying process generating the data. This can be seen in figure 2.1 in the case of linear regression of degree

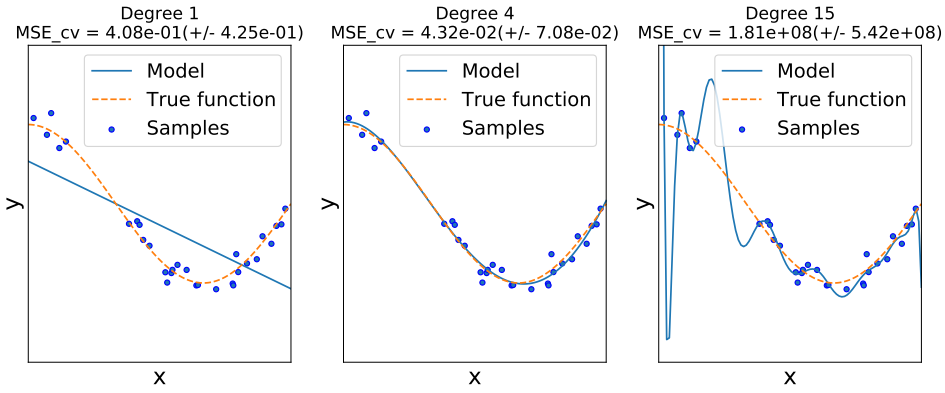


Figure 2.1: Regression of different degree polynomials on a cos curve. We see that a linear regression of degree 1 is insufficient to fit the training set and is underfitting. A polynomial of degree 4 fits the curve almost perfectly while a polynomial of degree 15 learns the noise of the data and overfits. How well the different regressors generalize to a test set is evaluated quantitatively by calculating the mean squared error using the loss function in equation 2.2.1 using 10-fold cross-validation. We see that this error is lowest when using 4-degree polynomial regression. Cross-validation is covered in section 2.3.1. The standard error of these losses are shown in the parenthesis. The Python code for producing this figure can be seen in appendix .1.

15, where the model lies very close to all of the training samples meaning it must have a very small MSE on the training data. But if we sample points from the true function we see that many of these will very likely lie far from the model, resulting in a large test error relative to the training error. Overfitting and underfitting are both something we want to avoid.

We can control whether a model under- or overfits by changing its capacity. Capacity is a model's ability to learn a larger variety of hypothesis. A model with a low capacity might underfit the training set if it can not capture the structure of the data. A model with high capacity might overfit if it learns properties of the training set that is not representative of the test set. As an example we can increase the capacity of linear regression by allowing it to fit higher order polynomials, thus increasing the hypothesis space from which we draw prediction rules. This example is seen in figure 2.1, where we see that a low capacity causes underfitting while a very large capacity causes the model to overfit.

2.5 Regularization

So far we have covered how to control over- and underfitting by changing a model's capacity. Another method is regularization which encodes the objective function with preferences for certain functions instead of excluding functions from the hypothesis space. The objective function is what we ask a machine learning algorithm to minimize, and has in the previous been synonymous with average loss. The hypothesis h is what we will now embed in the choice of model parameters $\boldsymbol{\theta}$.

A popular type of regularization is norm penalties, where a norm penalty function $\Omega(\boldsymbol{\theta})$ is added to the objective function J (which is usually average loss). Then the regularized objective function is

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}) \tag{2.5.1}$$

where $\alpha \in [0, \infty]$ is a parameter chosen before minimizing $\tilde{J}$, that controls how much the penalty term $\Omega(\boldsymbol{\theta})$ contributes relatively to the objective function J. When $\alpha = 0$ we have no regularization while larger values for α results in more regularization. In this way minimizing $\tilde{J}$ becomes a trade-off between fitting the training data and having small weights.

A common norm penalty function is the L2 parameter norm penalty also known as weight decay $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{\theta}||_2^2$. L2 regularization is also known as ridge regression or Tikhonov regularization. In a regression setting $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{w} + \boldsymbol{b}$, if we assume no bias parameter $\boldsymbol{b}$ and that model parameters are only weights $\boldsymbol{\theta} = \boldsymbol{w}$, the regularized objective function is defined by

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \frac{\alpha}{2}\boldsymbol{\theta}^\top\boldsymbol{\theta} \tag{2.5.2}$$

We can clearly see from this, that larger values of α punishes larger weights.

A regularization method often used for neural networks is early stopping. We cover this method in section 3.3 providing a popular intuitive approach to regularization other than weight decay.

There are many other techniques to reduce overfitting, which we will not cover. One such related type of regularization is norm penalties as constrained optimization problems where

$\Omega(\boldsymbol{\theta})$ is constrained by some constant k while seeking to minimize equation 2.5.1. Another is data augmentation that increases the size of the training set by augmenting existing data and adding the augmented copy to the dataset. A last worthy mention, used specifically for neural networks, is dropout (Srivastava et al. (2014)), which regularize by ignoring a random set of network-nodes during training.

2.6 Gradient Based Optimization

Supervised machine learning algorithms are often based on learning parameters by minimizing a regularized loss function. This can be done for a differentiable convex loss function by minimizing its gradient, which is what we do in gradient based optimization.

2.6.1 Gradient Descent

Gradient descent is an algorithm suggested by Cauchy (1847). The algorithm is an iterative process of selecting parameters that decrease the gradient of the loss function $J(\boldsymbol{\theta}, \boldsymbol{X}, y)$. We will consider sampled examples from the sample space $\mathbb{X}$ and aim to estimate the model parameters $\boldsymbol{\theta}$. The goal is to minimize the loss function which we assume to be convex, so that the function is minimized where the gradient is zero

$$\nabla J(\boldsymbol{\theta}, \boldsymbol{X}, y) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \ldots, \frac{\partial J}{\partial \theta_d}\right] = \mathbf{0}$$

The gradient tells us which direction that has the steepest slope on J, and we can thus change the model parameters such that we move in the opposite direction of where the gradient is pointing. This will bring us in the direction that locally decreases the objective function the fastest. In order to run the algorithm we need to initialize the model parameters $\boldsymbol{\theta} \equiv \boldsymbol{\theta}^0$, evaluate the objective function $J(\boldsymbol{\theta}, \boldsymbol{X}, y)$ and calculate the gradient respectively $\nabla J(\boldsymbol{\theta}, \boldsymbol{X}, y)$. Further we also need to specify a learning rate η, which affects the size of the step i.e. change in parameters in the direction of the negative gradient for each iteration. The model parameters are then updated iteratively by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla J(\boldsymbol{\theta}^{(t)}, \boldsymbol{X}, y)$$

This means, that when we are far away from optimum (large gradient) we are taking larger steps, but when we get closer to optimum (smaller gradient) we take smaller and smaller

steps. The algorithm is not assured to reach a global minimum unless the loss function is strictly convex. Bishop (2007) mentions that in order to find a "good" minimum, we might be forced to run the algorithm multiple times, where we each time initialize with a randomly chosen starting point $\boldsymbol{\theta}^0$.

2.6.2 Stochastic Gradient Descent

A widely popular and more computational feasible approach is to learn parameters using the stochastic gradient descent (SGD) algorithm. SGD is a stochastic approximation of the gradient descent algorithm described above, which replaces the gradient calculated on the entire dataset by an estimate calculated from a randomly selected subset of the data. This is useful when one faces high-dimensional optimization problems as it reduces the computational burden significantly (Bottou & Bousquet (2008)). Some authors append the term "mini-batch" to the algorithm name expect for the case where only one example has been used as a subset of the data.

The pseudocode for SGD can be seen in algorithm 1. Convergence and stopping criteria of descent algorithms have been studied extensively in the literature, and will not be discussed rigorously here. We will only provide some of the most common stopping criteria found in software-packages. One is to stop when the norm of the gradient is beneath some threshold, $\left\|\nabla J\left(\boldsymbol{\theta}^{(t-1)}\right)\right\| < \epsilon$ another is when the decrease in norm of the gradient drops under some threshold, $\left\|\nabla J\left(\boldsymbol{\theta}^{(t-1)}\right)\right\| - \left\|\nabla J\left(\boldsymbol{\theta}^{(t)}\right)\right\| < \epsilon$. Most software also include a number of max iterations as a stopping criteria to prevent non-converging cases to run forever.

2.6.3 Stochastic Gradient Descent with Momentum

One issue with SGD, shown by Sutton (1986), is that it performs poorly with surfaces containing ravines, i.e places where the curvature is more steep in one dimension than in another, which is common around optima. In these cases SGD will oscillate across the slopes of the ravines while slowly making progress towards the optima. A method for accelerating the process towards the optima while dampening oscillation is introducing momentum. This

Algorithm 1: Mini-Batch Stochastic Gradient Descent

Input: A dataset S

Input: A learning rate η

Input: Gradient function of objective function $\nabla J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})$

Input: Initial parameter vector $\boldsymbol{\theta}^0$

Output: Resulting parameters $\boldsymbol{\theta}$

initialize $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^0$

repeat

  pick randomly $S' \in S$

  $\boldsymbol{g} \leftarrow \frac{1}{|S'|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in S'} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})$

  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g}$

until *stopping criteria is met*;

is done by adding a fraction γ of the update vector from time $t-1$, to the update vector for time t. Thus updating $\boldsymbol{\theta}$ by

$$v_t = \gamma v_{t-1} + \eta \nabla J\left(\boldsymbol{\theta}^{(t)}\right)$$
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - v_t$$

The momentum term increases for dimensions with gradients pointing in the same directions and reduces updates for dimensions where gradients change directions. This results in faster convergence and reduced oscillation.

2.6.4 Adaptive Gradiant Algorithm (AdaGrad)

Another possible improvement to SGD is by adapting the learning rate to the parameters, so that stepsize decreases as we perform each iteration. This makes it possible to move faster initially and afterwards slow down to avoid overstepping the optima, which will result in faster convergence. This method was proposed by Duchi et al. (2011) with an SGD-extension called Adaptive Gradiant Algorithm (AdaGrad). AdaGrad performs the parameter-update by

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{\varepsilon + \sum_{\tau=1}^{t} \left(\nabla J\left(\theta_i\right)^{(t)}\right)^2}} \nabla J\left(\theta_i\right)^{(t)}$$

making each parameters have its own learning rate $\frac{\eta}{\sqrt{\varepsilon+\sum_{\tau=1}^{t}\left(\nabla J(\theta)^{(t)}\right)^2}}$, where ε is a small number that ensures that we do not divide by 0. As the sum of gradients increases with each step the learning rate decreases, which results in a smaller stepsize as the algorithm progresses.

Another upside to this approach, besides faster convergence, is that we do not need to manually tune the learning rate as it is now adapted. AdaGrad is also especially efficient when working with sparse data, as parameters for features that are often 0 (infrequent features), can receive more impact-full updates (higher learning rate) without affecting the impact on features that are often non-zero (frequent features). Without adaptive learning rates for each parameter we might inefficiently have a learning rate that decreases too slowly for frequent features or too quickly for infrequent features.

2.6.5 Root Mean Square Propagation (RMSprop)

AdaGrad's weakness is that the learning rate eventually can become infinitesimally small, making it impossible for the algorithm to learn i.e. update the parameters further. The Root Mean Square Propagation (RMSprop) method suggested by Tieleman & Hinton (2012) is an extension of AdaGrad, that seeks to overcome the weakness of AdaGrad's aggressive monotonically decreasing learning rate. RMSprob does this by replacing the sum of past gradients by an expectation of the gradient, thus updating parameters by

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{\varepsilon + \mathbb{E}\left[\nabla J\left(\theta_i^{(t)}\right)^2\right]}} \nabla J\left(\theta_i^{(t)}\right)$$

where

$$\mathbb{E}\left[\nabla J\left(\theta_i^{(t)}\right)^2\right] = (1-\gamma)\,\nabla J\left(\theta_i^{(t)}\right)^2 + \gamma \mathbb{E}\left[\nabla J\left(\theta_i^{(t-1)}\right)^2\right]$$

which makes it possible for the learning rate to either increase or decrease for each epoch. RMSprob also efficiently define the sum of gradients recursively as a decaying average of past gradients instead of inefficiently storing all of the past gradients as in AdaGrad. The running average $\mathbb{E}\left[\nabla J\left(\theta_i^{(t)}\right)^2\right]$ only depends on a hyperparameter-defined fraction γ (similar to the momentum term), the previous average and the current gradient.

2.6.6 Adaptive Moment Estimation (ADAM)

Kingma & Ba (2015) proposes an algorithm called Adaptive Moment Estimation (ADAM) which takes a similar approach to computing adaptive learning rate by storing an exponentially decaying average of past gradients $\boldsymbol{v}^{(t)}$ like RMSprop. ADAM however extends this idea by also keeping an exponentially decaying average of past gradients $\boldsymbol{m}^{(t)}$, that works similar to momentum. These components are calculated by

$$\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\nabla J\left(\boldsymbol{\theta}^{(t)}\right)$$
$$\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)\nabla J\left(\boldsymbol{\theta}^{(t)}\right)^2$$

where $\nabla J\left(\boldsymbol{\theta}^{(t)}\right)^2$ indicates the elementwise square $\nabla J\left(\boldsymbol{\theta}^{(t)}\right) \odot \nabla J\left(\boldsymbol{\theta}^{(t)}\right)$ and where $\boldsymbol{m}^{(t)}$ and $\boldsymbol{v}^{(t)}$ are estimates of the first moment and second moment of the time-t gradient respectively, hence the name Adaptive Moment Estimation. The authors observed that as $\boldsymbol{m}^{(t)}$ and $\boldsymbol{v}^{(t)}$ are initialized as $\mathbf{0}$ they will be biased estimates. They counteract this bias by deriving the bias-corrected estimates

$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t}$$
$$\hat{\boldsymbol{v}}^{(t)} = \frac{\boldsymbol{v}^{(t)}}{1-\beta_2^t}$$

The parameters are then updated by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \epsilon}\hat{\boldsymbol{m}}^{(t)}$$

The pseudocode for ADAM can be seen in algorithm 2.

Algorithm 2: ADAM (with mini-batch)

Input: A dataset S

Input: A learning rate η

Input: Exponential decay rates for moment estimates $\beta_1, \beta_2 \in [0, 1)$

Input: Gradient function of objective function $\nabla J(\boldsymbol{\theta})$

Input: Initial parameter vector $\boldsymbol{\theta}^0$

Output: Resulting parameters $\boldsymbol{\theta}$

initialize $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^0$

initialize $\boldsymbol{m}^{(0)}, \boldsymbol{v}^{(0)} \leftarrow \mathbf{0}$

initialize $t \leftarrow 0$

repeat

$t \leftarrow t + 1$ pick randomly $S' \in S$

$\boldsymbol{g}^{(t)} \leftarrow \frac{1}{|S'|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in S'} \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t-1)}, \boldsymbol{x}, \boldsymbol{y}\right)$

$\boldsymbol{m}^{(t)} \leftarrow \beta_1 \boldsymbol{m}^{(t-1)} + (1 - \beta_1)\boldsymbol{g}^{(t)}$

$\boldsymbol{v}^{(t)} \leftarrow \beta_2 \boldsymbol{v}^{(t-1)} + (1 - \beta_2)\boldsymbol{g}^{(t)^2}$

$\hat{\boldsymbol{m}}^{(t)} \leftarrow \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t}$

$\hat{\boldsymbol{v}}^{(}t) \leftarrow \frac{\boldsymbol{v}^{(t-1)}}{1-\beta_2^t}$

$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \varepsilon} \hat{\boldsymbol{m}}^{(t)}$

until *stopping criteria is met*;

Chapter 3

Neural Networks

Neural networks (NN) are machine learning algorithms inspired by the biological neural network that constitute the brain. A neural network is a collection of nodes called neurons that transmits and processes signals to and from other neurons connected to it, much like the biological neural network. The goal of a neural network is to approximate some function by learning parameters that results in the best approximation. They are popular as they can usually be constructed to do this fairly well, several people (Cybenko (1989), Funahashi (1989)) have shown that a neural network with one hidden layer can approximate any function on a compact domain arbitrarily closed, if provided with a sufficient number of neurons.

In section 3.1, we introduce the most common neural network, the feedforward neural network, which is the type of network we refer to throughout the dissertation, when we use the term neural network. In section 3.2 we examine a common method for training a neural network called backpropagation and in section 3.3 we introduce a popular method of regularizing neural networks to avoid overfitting.

3.1 Feedforward Neural Networks

The most simple neural networks are called feedforward neural networks or multilayer perceptrons (MLPs). They are called feedforward as information flows in one direction through the neurons. The neurons are typically arranged in layers to indicate which neurons receive and sends data to which neurons. These layers are typically vector-valued with each el-

ement of the vector playing the role as a neuron. As such one can describe a neural network as a chain of functions. Say we have three layers, then the neural network is $f(\mathbf{x}) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right)$, with $f^{(1)}$ being the first layer to pass the data, $f^{(2)}$ the second layer and so on. The final layer is called the output layer, while the intermediate layers, that do not directly show their output to the user, are called hidden layers.

A feedfoward neural network is shown in figure 3.1 where the data flows from the input layer through two hidden layers and finally to the output layer. While the number of neurons in the input- and output layer depends on the example and the label dimensions, the number of neurons in the hidden layers as well as the number of hidden layers are design decisions chosen by the architect.

The data is sent through these layers by multiplying the inputs with weights $\boldsymbol{w}$, adding a bias $\boldsymbol{b}$ and finally applying an activation function g to the result. An activation function is a non-linear function that ensures that we do not just pass linear transformations of the examples through the network, which is necessary to learn non-linear functions. Activation functions come in many different forms, some of the most popular can be seen in figure 3.2. For a network with two hidden layers the result of the first hidden layer is $f^{(1)}(\boldsymbol{X}) = g_1(\boldsymbol{X}\boldsymbol{w}_1 + \boldsymbol{b}_1)$ and the result of the second hidden layer is $f^{(2)}\left(f^{(1)}(\boldsymbol{X})\right) = g_2\left(f^{(1)}(\boldsymbol{X})\,\boldsymbol{w}_2 + \boldsymbol{b}_2\right)$. The subscripts indicate that the activation functions, weights and biases are unique for each layer. At last the result of the output layer is $f^{\text{out}}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{X})\right)\right) = g_3\left(f^{(2)}\left(f^{(1)}(\boldsymbol{X})\right)\boldsymbol{w}_3 + \boldsymbol{b}_3\right)$. Sometimes the biases are omitted, as they can be represented in the weight vector for an example-matrix that are padded with a column of ones.

While the choice of activation function are chosen as a design decision by the architect for the hidden layers, the activation function for the output layer are defined by the learning task. If the neural network is performing regression then we typically do not have an activation function for the output layer, as the label space for regression is the real numbers and need not be mapped to a specific set of values. However for binary classification a sigmoid-function is used, while a softmax-function is used for multiclass classification, as these map real values to the interval $[0, 1]$ and thus provides valid probabilities for belonging

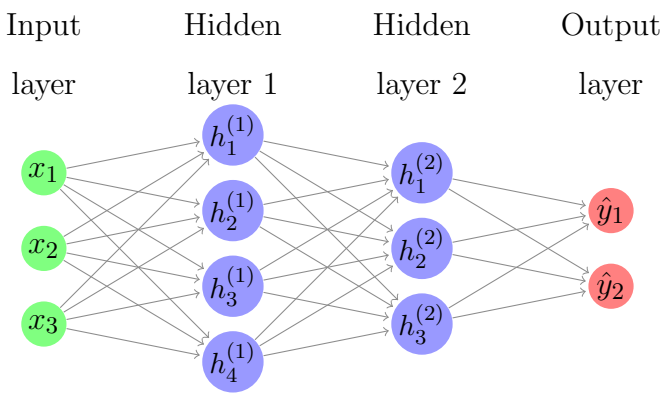


Figure 3.1: Feedforward neural network with 2 hidden layers and arrows indicating where neurons feed their data.

to a specific class. Some popular choices of activation functions are shown in figure 3.2. Taking an activation value a as input, these are defined as follows:

Logistic sigmoid:

$$g(a) = \sigma(a) \equiv \frac{1}{1 + e^{-a}} \tag{3.1.1}$$

Softmax for K classes:

$$g(a)_i = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}} \quad \text{for} \quad i = 1, \ldots, K$$

Hyperbolic tangens:

$$g(a) = \tanh(a) \equiv \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} \tag{3.1.2}$$

Rectified linear unit (ReLU):

$$g(a) = \max(0, a) \tag{3.1.3}$$

Exponential linear unit (ELU):

$$g(a) = \begin{cases} a & a \geq 0 \\ \alpha\left(e^{a} - 1\right) & a < 0 \end{cases} \quad \text{for } \alpha > 0$$

Step:

$$g(a) = \mathbf{1}_{[a>0]}$$

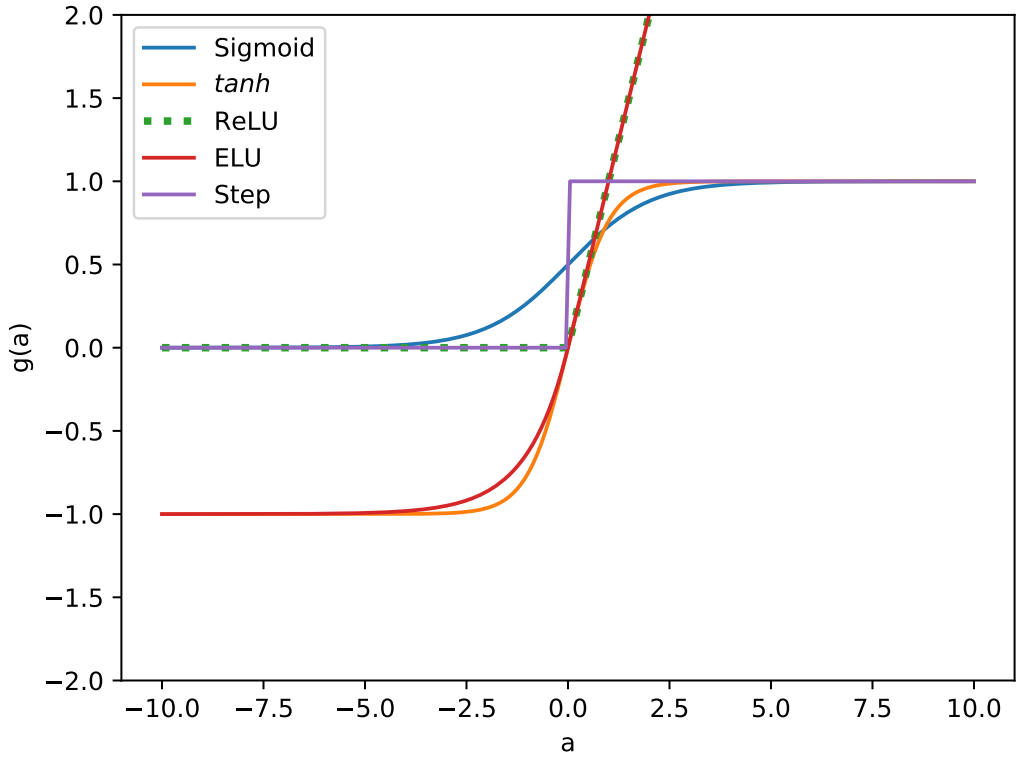


Figure 3.2: Plot of some of the most used activation functions. The Python code for producing this figure can be seen in appendix .2

3.2 Backpropagation

As mentioned in section 2.3 we seek to find the hypothesis $\hat{h}_S^*$ that minimizes our empirical loss. For notational convenience we will collect the bias and weights in a parameter vector $\boldsymbol{\theta}$, which can be interpreted as using examples padded with a column of ones, so that $\boldsymbol{x}\boldsymbol{w} + \boldsymbol{b} = \boldsymbol{x}_{\text{padded}}\boldsymbol{\theta}$. In the case of neural networks $\hat{h}_S^*$ determine the weights used in each neuron as described in section 3.1. To obtain these optimal weights we train the neural network by minimizing the gradient of the (perhaps regularized) loss function J with respect to the weights. To learn these weights we use a minimization algorithm like stochastic gradient descent described in section 2.6.2. However the gradient of the loss function is not readily available in neural networks as we have to trace the information back through the network that produced $\hat{y}$ for which we calculate the loss. The solution is an algorithm proposed by Rumelhart et al. (1986) called back-propagation or backprop for short.

Back-propagation uses the chain rule of calculus, which states that

$$\frac{d\, f\left(g\left(x\right)\right)}{d\, x} = \frac{d\, f\left(g\left(x\right)\right)}{d\, g\left(x\right)} \frac{d\, g\left(x\right)}{d\, x}$$

for a real number x and functions g and f. This rule can be generalized for a vector $\boldsymbol{x} \in \mathbb{R}^m$

$$\frac{\partial f\left(g\left(\boldsymbol{x}\right)\right)}{\partial x_i} = \sum_j \frac{\partial f\left(g\left(\boldsymbol{x}\right)\right)}{\partial g(x_j)} \frac{\partial g(x_j)}{\partial x_i}$$

where $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. In a neural network with one hidden layer we find the gradient of the loss function $J(\boldsymbol{\theta})$ (with $\boldsymbol{X}$ and $\boldsymbol{y}$ as implicit inputs to keep notation uncluttered) by

$$\nabla J\left(\boldsymbol{\theta}\right) = \begin{pmatrix} \frac{\partial J\left(f^{\text{out}}\left(f^{(1)}(\boldsymbol{\theta})\right)\right)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J\left(f^{\text{out}}\left(f^{(1)}(\boldsymbol{\theta})\right)\right)}{\partial \theta_m} \end{pmatrix} \tag{3.2.1}$$

where each element is found by

$$\frac{\partial J\left(f^{\text{out}}\left(f^{(1)}\left(\boldsymbol{\theta}\right)\right)\right)}{\partial \theta_i} = \frac{\partial J\left(f^{\text{out}}\left(f^{(1)}\left(\boldsymbol{\theta}\right)\right)\right)}{\partial f^{\text{out}}\left(f^{(1)}\left(\boldsymbol{\theta}\right)\right)} \frac{\partial f^{\text{out}}\left(f^{(1)}\left(\boldsymbol{\theta}\right)\right)}{\partial f^{(1)}\left(\boldsymbol{\theta}\right)} \frac{\partial f^{(1)}\left(\boldsymbol{\theta}\right)}{\partial \theta_i} \tag{3.2.2}$$

which can easily be generalized to networks with more than one hidden layer, by using the chain rule in a similar fashion. A computational graph for calculating this in a neural network is shown in figure 3.3. This provides the fundamentals to understand how the gradient is computed in practice. A vector of weights (perhaps chosen by an update of stochastic

gradient descent) is sent through the network as shown by algorithm 3 to provide a loss value. This is called forward-propagation. Then the gradient in equation 3.2.1 is computed using the relation in equation 3.2.2 by back-propagation as shown in algorithm 4. The result of the backpropagation algorithm is gradients on the weights that can be directly used for learning the weights that minimizes the gradient on the loss in a stochastic gradient descent algorithm as the one shown in algorithm 1.

Some activation functions like the ReLU-function $g(a) = \max(0, a)$ are not differential, which might sound problematic for backpropagation and gradient based learning. However, as mentioned by Goodfellow et al. (2016), gradient descent still performs well enough for these models, partly because training algorithms usually do not arrive at a local minimum of the loss function. This means that we do not expect to get a gradient of exactly $\mathbf{0}$, and therefore it is not a problem that the minimum of the loss function corresponds to points with an undefined gradient. Furthermore Goodfellow et al. (2016) mentions that most software return one of the one-sided derivatives, which can be heuristically justified since a computer is subject to numerical error anyway.

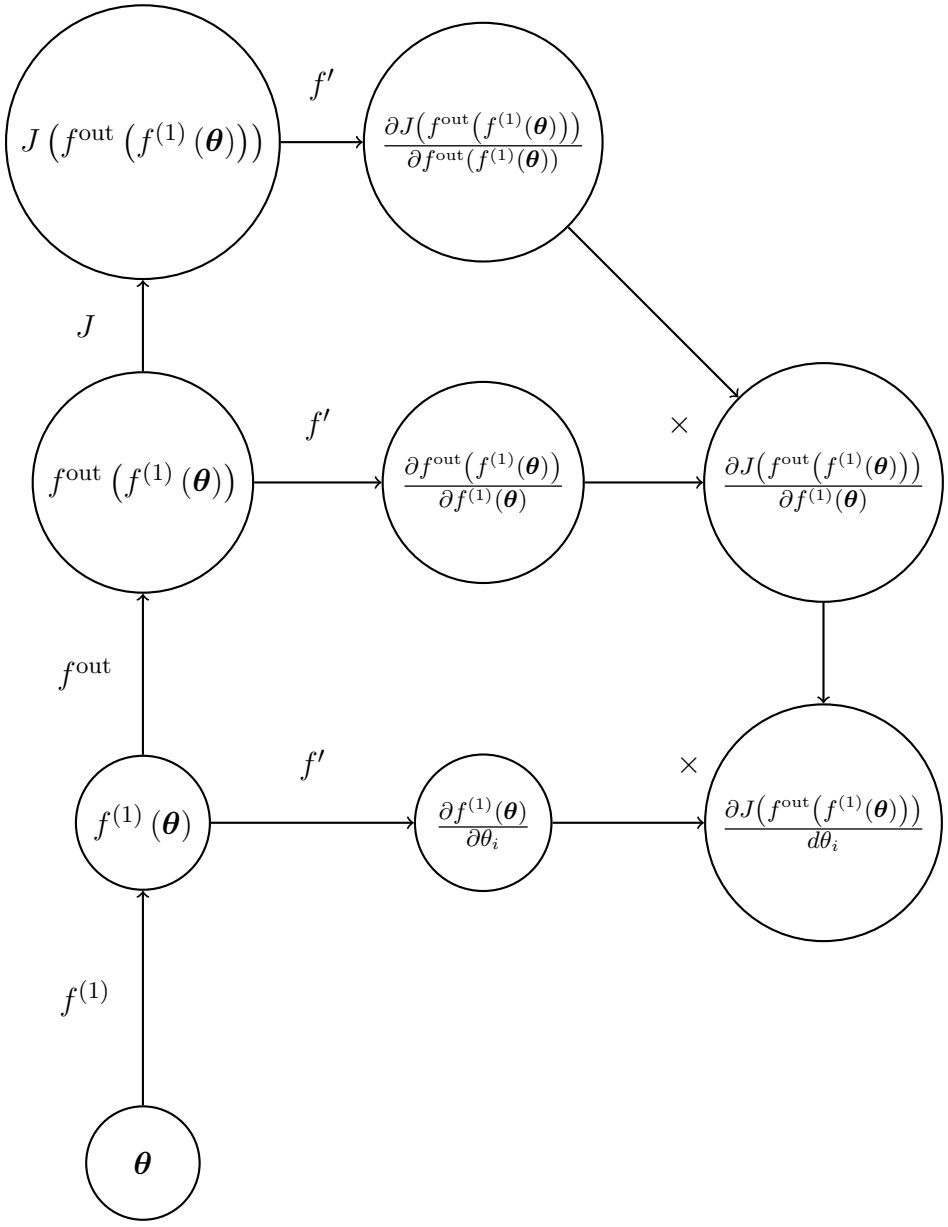


Figure 3.3: Computational graph of calculating the elements of the gradient in equation 3.2.2 for the network with one hidden layer. f' indicates that the function of the node is derived with regard to its input and $\times$ indicates that the node is a product of the nodes pointing to it.

Algorithm 3: Forwardpropagation through a neural network and the computation of the cost function. For simplicity this demonstration uses only a single input example $\boldsymbol{x}$, in practice one typically uses a minibatch of examples. We have also omitted the bias terms for simplicity, as these can be part of the weights $\boldsymbol{w}^{(i)}$ with an example $\boldsymbol{x}$ padded with a column of 1's. The collection of weights are denoted by $\boldsymbol{\theta}$.

Input: Number of layers, l

Input: The weight vectors of each layer $\boldsymbol{w}^{(i)}, i \in \{1, \ldots, l\}$

Input: The activation functions of each layer, $g^{(i)}, i \in \{1, \ldots, l\}$

Input: The example to process, $\boldsymbol{x}$

Input: The target label for the example, $\boldsymbol{y}$

Input: The (regularized) loss function J

Output: The (regularized) loss on the example, $J(\hat{\boldsymbol{y}}, \boldsymbol{y}, \boldsymbol{\theta})$

initialize $\boldsymbol{h}^{(0)} \leftarrow \boldsymbol{x}$

for $k = 1, \ldots, l$ **do**

$\quad \boldsymbol{a}^{(k)} \leftarrow \boldsymbol{w}^{(k)} \boldsymbol{h}^{(k-1)}$

$\quad \boldsymbol{h}^{(k)} \leftarrow g^{(k)}\left(\boldsymbol{a}^{(k)}\right)$

end

$\hat{\boldsymbol{y}} \leftarrow \boldsymbol{h}^{(l)}$

$J(\hat{\boldsymbol{y}}, \boldsymbol{y}, \boldsymbol{\theta}) \leftarrow L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{\theta})$

Algorithm 4: Back-propagation

Input: Same variables as forwardpropagation in algorithm 3

Output: The gradient of the regularized loss function on the weights,

$$\boldsymbol{d} = \nabla_{\boldsymbol{w}} J$$

Perform forward propagation as per algorithm 3

Computation of the loss-gradient on the output layer:

$\boldsymbol{d} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L\left(\hat{\boldsymbol{y}}, \boldsymbol{y}\right)$

for $k = l, l-1, \ldots, 1$ **do**

$\boldsymbol{d} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{d} \odot g^{(k)'}\left(\boldsymbol{a}^{(k)}\right)$

$\nabla_{\boldsymbol{\theta}^{(k)}} J \leftarrow \boldsymbol{d}\boldsymbol{h}^{(k-1)^\top} + \lambda \nabla_{\boldsymbol{\theta}^{(k)}} \Omega\left(\boldsymbol{\theta}\right)$

$\boldsymbol{d} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{\theta}^{(k)^\top} \boldsymbol{d}$

end

3.3 Early Stopping

For this regularization methods we introduce a validation set, which is a third split of the data that is separate from the training set, that trains our neural networks, and separate from the test set, that evaluates our neural network. This validation set is used for selecting a non-overfitting set of parameters proposed by the training of the algorithm. This means that the validation set contributes to the selection of the optimal hypothesis, which is why we must keep it separate from the test set, which we use to evaluate our algorithm.

Goodfellow et al. (2016) argues that when we have large models with sufficient capacity to overfit, we often observe that the training error decreases steadily over time, while the validation error begins to rise again at some point before ending the training. This means that we obtain a better model if we stop training and return the parameters at the point with the lowest validation error. This is the concept of regularizing using early stopping. The algorithm terminates training when no parameters have improved the validation error over some pre-specified number of iterations called the patience, ϕ. The measure of improvement can be controlled by a hyperparameter δ_{min}, making sure we only constitute an improvement if the decrease in validation error is more than δ_{min}. The parameters $\boldsymbol{\theta}$ used for calculating the loss are provided by a training algorithm like ADAM from section 2 using back-propagation from section 3.2, while the loss is provided by the network output for these parameters and the chosen loss function. This procedure is specified more formally in algorithm 5.

One can think of early stopping as a hyperparameter selection algorithm that chooses the number of training steps for the training algorithm. The only significant cost of choosing this parameter automatically with early stopping is running the evaluation of the validation error periodically during training. However this can be done in parallel to the training process, as in Tensorflow where one can run parallel on the GPU. An additional cost is the need to maintain a copy of the optimal parameters, but this is negligible according to Goodfellow et al. (2016) since they are written to infrequently and never during the training-algorithm, and since they can preferably be stored in a slower and larger form of memory like in host memory or disk drive.

Bishop (1995) and Sjöberg & Ljung (1992) argues that early stopping has the effect of restricting the optimization procedure to a small volume of parameter space in the neighborhood of the initial parameter value. Goodfellow et al. (2016) shows with a simple linear model with a quadratic error function and gradient descent how early stopping is equivalent to L2 regularization in equation 2.5.2, but that early stopping has the advantage of automatically determining the correct amount of regularization while L2 regularization requires many training experiments with different values of its hyperparameter α.

Algorithm 5: Early stopping

Input: The number of steps between evaluations, n

Input: The patience parameter i.e. the number of times of observing worsening validation error before terminating, ϕ

Input: The measure of improvement, δ_{min}

Output: Parameters when terminating $\boldsymbol{\theta}^*$, training step when terminating i^*

initialize $i, j \leftarrow 0$

initialize $v \leftarrow \infty$

Let $\boldsymbol{\theta}_0$ be the initial parameters

$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}_0$

$i^* \leftarrow i$

while $j < \phi$ **do**

  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps

  $i \leftarrow i + n$

  $v' \leftarrow \hat{L}\left(S_{\text{val}}, \boldsymbol{\theta}\right)$

  if $v' < v - \delta_{min}$ **then**

    $j \leftarrow 0$

    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$

    $i^* \leftarrow i$

    $v \leftarrow v'$

    else

      $j \leftarrow j + 1$

    end

  end

end

Chapter 4

Bayesian Neural Networks

A Bayesian neural network (BNN) is a probabilistic model build on the architecture of a neural network, but trained using Bayesian inference. The intent of such a model is to use the approximation capabilities of the neural networks, described in section 3, combined with the capability of estimating uncertainty from Bayesian inference. These uncertainty estimating capabilities comes from considering the model parameters $\boldsymbol{\theta}$ as a random variable, such that predictions are made by considering the probability distribution of $\boldsymbol{\theta}$. BNNs can therefore be viewed as a special case of ensemble learning (Zhou (2012)) where the model ensemble is constructed by considering all possible values for $\boldsymbol{\theta}$. In practice we are not able to consider all possible values of $\boldsymbol{\theta}$ and sampling methods are used to sample different models as an approximation and there predictions are aggregated to make a single prediction.

In section 4.1 we introduce two principal concepts of statistical inference; the frequentist and Bayesian paradigm and the main differences between these two paradigms. In section 4.2 we introduce the basic theory of Monte Carlo methods and in section 4.3 we illustrate the main ideas behind Bayesian neural networks with a simple example. Since basic Monte Carlo methods are often inefficient when sampling from complex distributions we examine more sophisticated sampling methods based on Markov chains in section 4.4.

One of these methods is the Metropolis algorithm, which we examine in section 4.4.2. Afterwards we examine the Hamiltonian Monte Carlo in section 4.5. This algorithm builds on the principles of Metropolis, but explores the target distribution more efficiently. Lastly

in section 4.5.5 we cover an extension of Hamiltonian Monte Carlo, that fixes possible inefficient "U-turns", called the No-U-Turn sampler. We end this chapter by briefly examining the choice of prior distributions in section 4.6 and how one can choose the parameters for these using a distribution.

4.1 Bayesian & Frequentist Views of Learning

In this section we will introduce two concepts of statistical inference. These are the Bayesian and the frequentist paradigm.

The ideas behind Bayesian statistics goes back to the 18th century and is named after Thomas Bayes (Stigler (1986)). In Bayesian learning we consider the model parameters $\boldsymbol{\theta}$ as random and aim to learn the probability distribution of these. On the other hand the conventional frequentist methodology considers the model parameters $\boldsymbol{\theta}$ as fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a random variable, as it is a function of the dataset, which is assumed to be random.

To illustrate the difference between these two approaches in more detail, we will consider an example, which involves a simple coin toss. The uncertainty of the coin showing head or tails can be expressed by the coins probability p of showing heads, which is often referred to as the bias of the coin. Since the properties of the coin is not known beforehand, we do not know the exact probability of showing heads. It could be a fair coin and have the probability $p = \frac{1}{2}$ or it could be an unbalanced coin meaning that $p \neq \frac{1}{2}$.

A Bayesian statistician would express this uncertainty by a probability distribution over possible values of the unknown probability p and would then update the distribution as more observations become known. Frequentists would find the introduction of a distribution over parameter weights as pure nonsense. The frequentist would instead flip the coin a given number of times to form a dataset, and choose some estimator, for the unknown probability p which would be most consistent with the data, an obvious choice would be the relative frequency of heads in the past coin tosses.

This example illustrates the main differences between these two paradigms and these differences will be highlighted in a more formal way in the subsequent sections.

4.1.1 Maximum Likelihood Estimation

Let us now consider a dataset $\boldsymbol{X}$, with n examples $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots \boldsymbol{x}^{(n)}$ drawn independently from the true but unknown distribution. We let $L(\boldsymbol{\theta} \mid \boldsymbol{X}) \equiv p(\boldsymbol{X} \mid \boldsymbol{\theta})$ denote the likelihood function, where $p(\boldsymbol{X} \mid \boldsymbol{\theta})$ is a parametric family of probability distributions with parameter $\boldsymbol{\theta}$. A frequentist is, as described earlier, trying to find an estimate of the true parameter that have generated the data, we call such an estimate a point estimate and denote it by $\hat{\boldsymbol{\theta}}$ to separate it from the true parameter. A point estimate can be viewed as a function of data $\hat{\boldsymbol{\theta}} = g\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots \boldsymbol{x}^{(n)}\right)$ which is drawn from a random process meaning that $\hat{\boldsymbol{\theta}}$ is a random variable. In most cases of frequentist inference a point estimate is found by maximizing the likelihood and is called the maximum likelihood estimate (MLE)

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{\text{MLE}} &= \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta} \mid \boldsymbol{X}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{\theta}\right)\end{aligned}$$

It is often more convenient to maximize a sum instead of a product, not only is it easier to handle sums when differentiating but it also helps stabilize the calculations numerically (Goodfellow et al. (2016)). Thus, we take the logarithm of the likelihood function to obtain the log-likelihood function $\ell(\boldsymbol{\theta} \mid \boldsymbol{X}) \equiv \log p(\boldsymbol{X} \mid \boldsymbol{\theta})$,

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{\text{MLE}} &= \arg\max_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta} \mid \boldsymbol{X}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{\theta}\right)\end{aligned}$$

and since the logarithm is a monotonic-increasing function, optimizing the log-likelihood is equivalent to optimizing the likelihood.

When the size of the dataset is small the MLE is often prone to overfitting and regularization methods such as penalized maximum likelihood are applied, see Hastie et al. (2001).

We can generalize the maximum likelihood estimator to the case where our goal is to

estimate a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta})$, in order to predict $\boldsymbol{y}$ given $\boldsymbol{X}$ as described for supervised learning in section 2. In this case the conditional maximum likelihood is given by

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta} \mid \boldsymbol{y}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta})$$

and if assume that the targets $\boldsymbol{y}^{(1)}, \dots \boldsymbol{y}^{(n)}$ are i.i.d, we can write

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p\left(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{\theta}\right)$$

The sum of squared errors, obtained by summing over all datapoints in equation 2.2.1, can be justified theoretically by the use of maximum likelihood with a Gaussian likelihood. To see this, consider a regression model that outputs $f(\boldsymbol{X}; \boldsymbol{\theta}) = \boldsymbol{\theta}\boldsymbol{X}$, with real-valued targets $\boldsymbol{y}$. If we define the likelihood as the conditional distribution of $\boldsymbol{y}$ as Gaussian with mean given by the regression output $\hat{y} \equiv f(\boldsymbol{X}; \boldsymbol{\theta})$ and standard deviation σ, we can write

$$L(\boldsymbol{\theta} \mid \boldsymbol{y}) = \prod_{i=1}^{n} p\left(y^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{\theta}\right) = \prod_{i}^{n} \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-(\hat{y}^{(i)} - y^{(i)})^2 / 2\sigma^2\right)$$

Next we take the logarithm of the likelihood function which gives us the log-likelihood function

$$\begin{aligned}
\ell(\boldsymbol{\theta} \mid \boldsymbol{y}) &= \sum_{i}^{n} \log \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-(\hat{y}^{(i)} - y^{(i)})^2 / 2\sigma^2\right) \\
&= -\frac{n}{2} \log \sigma^2 - \frac{n}{2} \log(2\pi) - \frac{1}{2\sigma^2} \sum_{i}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2
\end{aligned}$$

note that $-\frac{n}{2} \log \sigma^2 - \frac{n}{2} \log(2\pi)$ does not depend on the model parameters $\boldsymbol{\theta}$ and can therefore be omitted when maximizing. Maximizing the log-likelihood is the same as minimizing the negative log-likehood, so we can write

$$\min_{\boldsymbol{\theta}} -\ell(\boldsymbol{\theta} \mid \boldsymbol{y}) = \frac{1}{2\sigma^2} \sum_{i}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

and since $\frac{1}{2\sigma^2}$ does not depend on $\boldsymbol{\theta}$ either, we can see that minimizing the negative log-likelihood is the same as minimizing the sum of squared errors.

4.1.2 Bayesian Learning and Prediction

A different approach than the frequentist perspective of the parameter value $\boldsymbol{\theta}$ as fixed but unknown and the point estimator $\hat{\boldsymbol{\theta}}$ as a random variable, is taken with Bayesian inference.

The Bayesian paradigm considers the dataset as fixed and observable, while the true parameter $\boldsymbol{\theta}$ is unknown, and thus considered a random variable.

In Bayesian statistics, we begin with defining a prior distribution $p(\boldsymbol{\theta})$ over the parameters. This prior distribution expresses our initial view on the parameters, before any data has been observed. When data becomes available, we update our prior distribution to a posterior distribution. The posterior distribution is defined by Bayes' rule

$$p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{\theta})p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})}{p(\boldsymbol{y})}$$

and it combines the information about the data, that comes from the likelihood function $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})$, with the prior distribution. $p(\boldsymbol{y})$ is called the model evidence and is the distribution of the observed data marginalized over the parameters $p(\boldsymbol{y}) = \int p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$. The model evidence is often intractable, since it requires integration over all possible values of $\boldsymbol{\theta}$, which in many applications requires integration over high-dimensional spaces. As a result, finding analytical solutions of the posterior is not possible for complex models. We often ignore the evidence term, as it does not depend on $\boldsymbol{\theta}$ and thus for a fixed $\boldsymbol{y}$ can be interpreted as a constant, we thus write

$$p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{\theta})p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta}) \tag{4.1.1}$$

An important quality of the Bayesian method, is that it uses a full distribution over the parameters $\boldsymbol{\theta}$ to make predictions. Let us for example consider the case where we have observed a sample consisting of n examples $\boldsymbol{X} = \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$. To predict an unobserved label $\boldsymbol{y}^{(n+1)}$ for a new example $\boldsymbol{x}^{(n+1)}$ we need the posterior predictive distribution, that is to integrate the model predictions by the posterior

$$\begin{aligned} &p\left(\boldsymbol{y}^{(n+1)} \mid \boldsymbol{x}^{(n+1)}, \left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right) \\ &= \int p\left(\boldsymbol{y}^{(n+1)} \mid \boldsymbol{x}^{(n+1)}, \boldsymbol{\theta}\right) p\left(\boldsymbol{\theta} \mid \left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right) d\boldsymbol{\theta} \end{aligned} \tag{4.1.2}$$

This is quite different from the maximum likelihood method, that uses a point estimate for $\boldsymbol{\theta}$ to make predictions on any unobserved data, the Bayesian method takes the uncertainty of estimating $\boldsymbol{\theta}$ into account when making predictions, which tends to do well in avoidance of overfitting (Goodfellow et al. (2016)).

An important difference between the Bayesian approach and MLE, lies on the contribution of a prior distribution. The prior has the effect of shifting the probability mass towards regions of the parameter space, that are preferred a priori. According to Goodfellow et al. (2016), the prior often expresses a preference for models that are simpler or more smooth. Critics of the Bayesian method often point their fingers at the prior distribution, and criticize it for being a subjective component that can affect the predictions of the model. According to Neal (1996) Bayesian methods often do a lot better than a frequentist model, when training data is limited in availability, but suffers from high computational cost when the number of training examples are large.

Neal (1996) and MacKay (1991) argues that Bayesian models embodies Occam's Razor, which is the principle that we should prefer simpler models to complex ones. This principle is a component often found in machine learning since a too complex model might overfit the data. This belief is justified when model parameters are estimated by maximum likelihood, but Neal (1996) argues that one should not limit the complexity of Bayesian neural networks to prevent overfitting. The approach of training by minimizing loss might lead to a choice of model with increasing complexity the more data that is available. With a Bayesian approach adjusting the complexity of the model based on the amount of data available makes no sense, since a correct prior and model for 10 observations must be correct for 10.000 observations as well. One might however switch to a simple model if it seems unlikely that a complex computational expensive model will provide significant benefit.

As mentioned earlier, in many practical examples, the posterior distribution is intractable and therefore must be derived in some other way. Often we will use methods such as Monte Carlo to approximate the posterior distribution.

4.1.3 Maximum a Posteriori (MAP) Estimation

A way to avoid the computational hurdle of approximating the entire Bayesian posterior, is to use a point estimate as an approximation. Instead of turning completely to frequentist methods and use MLE, one can still benefit of the Bayesian method, by allowing the prior to influence the choice of the point estimate. One way to do this, is to use the maximum a posteriori (MAP) point estimate. The MAP estimate is obtained by maximizing the

posterior distribution

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{\text{MAP}} &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{X}, \boldsymbol{y}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\end{aligned} \tag{4.1.3}$$

note that the evidence term has been omitted, since it does not depend on the parameter $\boldsymbol{\theta}$ and thus vanishes under maximization anyway. The bottom part of equation 4.1.3 can be recognized as an equation consisting of the standard log-likelihood term plus a log-prior term.

As an example, consider a model with a Gaussian prior placed on the regression weights $\boldsymbol{\theta}$. If we specifically choose the prior to be given by $\mathcal{N}\left(\boldsymbol{\theta}, 0, \frac{1}{\alpha}\boldsymbol{I}^2\right)$, then the log-prior term in equation 4.1.3 is proportional to the L2 norm introduced in 2.5.2, plus a term that does not depend on $\boldsymbol{\theta}$. Note also that the MAP estimate is the same as the MLE, when choosing a uniform prior, since the $p(\boldsymbol{\theta})$ becomes a constant function in equation 4.1.3 and consequently we can ignore it when maximizing the expression. Compared to MLE, MAP estimation has the advantage that it can benefit from the information in the prior that cannot be found in the dataset. According to Goodfellow et al. (2016) this additional information, gained from the choice of prior, can reduce the variance in the MAP point estimate in direct comparison with MLE estimate, but this advantage has a price of an increased bias.

MAP is closely related to Bayes optimal estimation, instead of finding the most probable hypothesis (set of parameters), it aims at finding the most probable label for a new example. The Bayes optimal estimation is done by predicting the $\boldsymbol{y}^{(n+1)}$, which maximizes the posterior predictive distribution in equation 4.1.2. We do not pursue the idea of Bayes optimal estimation, as we aim to minimize a loss function which is not always equivalent to predicting the most probable label. In this way we preserve the idea that the loss function dictates how much the algorithm should care about making certain predictions as explained in 2.2.

An obvious disadvantage from using MAP estimation, is that it discards the information contained in the distribution. It is estimating distributions that make Bayesian methods attractive, especially if one wants to evaluate the uncertainty of the parameters. As we value access to this distribution higher than faster computational time we will not pursue

MAP estimation any further.

4.2 Monte Carlo Methods

One way of approximating intractable integrals, like the posterior predictive distribution in equation 4.1.2, is to use Monte Carlo methods. The idea behind Monte Carlo methods is to view the integral as an expectation of some random variable with respect to a probability distribution $p(\cdot)$. In the case of BNNs our random variable is the model parameters $\boldsymbol{\theta}$, and we can write the posterior predictive distribution as

$$\begin{aligned} & p\left(\boldsymbol{y}^{(n+1)} \mid \boldsymbol{x}^{(n+1)}, \left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right) \\ &= \int p\left(\boldsymbol{y}^{(n+1)} \mid \boldsymbol{x}^{(n+1)}, \boldsymbol{\theta}\right) p\left(\boldsymbol{\theta} \mid \left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right) d\boldsymbol{\theta} \\ &= \int f(\boldsymbol{\theta})\, \hat{p}(\boldsymbol{\theta})\, d\boldsymbol{\theta} \end{aligned} \tag{4.2.1}$$

where we use a simpler notation of $\hat{p}(\boldsymbol{\theta})$ to denote the posterior distribution. Such an integral can be interpreted as an expectation taking under the probability distribution $\hat{p}(\boldsymbol{\theta})$

$$s = \int f(\boldsymbol{\theta})\hat{p}(\boldsymbol{\theta})d\boldsymbol{\theta} = \mathbb{E}_{\hat{p}}[f(\boldsymbol{\theta})]$$

Now in order to approximate s we can draw samples from the distribution $\hat{p}(\boldsymbol{\theta})$ and approximate the expected value by the empirical average. If we draw n samples $\boldsymbol{\theta} \sim \hat{p}(\boldsymbol{\theta})$ we can approximate s by $\hat{s}_n$

$$\hat{s}_n = \frac{1}{n}\sum_{i=1}^{n} f\left(\boldsymbol{\theta}^{(i)}\right) \tag{4.2.2}$$

This implies the simplest situation, where it is possible to simulate directly from the probability distribution, which is often not possible.

We can justify this approximation theoretically, by noticing that $\hat{s}_n$ is an unbiased estimator of s

$$\mathbb{E}_{\hat{p}}\left[\hat{s}_n\right] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\hat{p}}\left[f\left(\boldsymbol{\theta}^{(i)}\right)\right] = \frac{1}{n}\sum_{i=1}^{n} s = s$$

additionally the law of large numbers, states that if the samples $\boldsymbol{\theta}^{(i)}$ are independent and identical distributed (i.i.d), the empirical average converges to the true expectation almost surely

$$\lim_{n\to\infty} \hat{s}_n = s$$

this only holds if the variance of the individual terms $\text{Var}[f\left(\boldsymbol{\theta}^{(i)}\right)]$ is bounded. To see this, we note that $\text{Var}[\hat{s}_n]$ converges to zero as n goes to infinity, if and only if $\text{Var}[f\left(\boldsymbol{\theta}\right)] < \infty$

$$\begin{aligned}\text{Var}\left[\hat{s}_n\right] &= \text{Var}\left[\frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{\theta}^{(i)})\right] \\ &= \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}[f(\boldsymbol{\theta}^{(i)})] = \frac{\text{Var}[f(\boldsymbol{\theta})]}{n}\end{aligned}$$

Further the central limit theorem states that, if $\mathbb{E}_{\hat{p}}[f(\boldsymbol{\theta})] = s$ and $\text{Var}[f(\boldsymbol{\theta})] < \infty$ then

$$\frac{\hat{s}_n - s}{\sqrt{\text{Var}[f(\boldsymbol{\theta})]/n}} \sim \mathcal{N}(0,1)$$

which is equivalent to

$$\hat{s}_n \sim \mathcal{N}\left(s, \frac{\text{Var}[f(\boldsymbol{\theta})]}{n}\right)$$

Which gives us a way of estimating confidence intervals around the estimate $\hat{s}$.

When it is infeasible or not possible to make simulations of $\boldsymbol{\theta}$ directly, Markov chain Monte Carlo methods can be used. such methods simulate from a target distribution by running a Markov chain, that eventually will converge to target distribution. Markov chain Monte Carlo methods will be examined more thoroughly in section 4.4.

4.3 A Simple Bayesian Neural Network

A simple example, inspired by Neal (1996), will illustrate the general concept of Bayesian learning for neural networks and the inefficiency of brute force methods of sampling. Figure 4.1 shows six BNNs whose weights and biases were drawn from independent standard normal prior distributions except output weights, which had a standard deviation of $\frac{1}{\sqrt{16}}$. The networks performs regression on six data points.

The six networks was chosen from a larger pool of 10^5 networks with weights and biases sampled from identical prior distributions. The likelihood was computed for each of these networks and scaled so that the largest likelihood was 1. The networks were then accepted with the probability of this scaled likelihood for which only six was accepted. This approach resembles rejection sampling and embodies the posterior from equation 4.1.1 by making the prior control the generation of candidate networks and the likelihood control which of these

candidates are rejected. We follow the suggestion of Neal (1996) to model regression tasks with a conditional distribution for the real valued targets $\boldsymbol{y}_k$, from k neural networks with outputs $f_k(\boldsymbol{x})$, defined by a Gaussian distribution

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \prod_k \frac{1}{\sqrt{2\pi}\sigma_k} \exp - \frac{\left(f_k(\boldsymbol{x}) - y_k\right)^2}{2\sigma_k^2} \tag{4.3.1}$$

with mean $f_k(\boldsymbol{x})$ and standard deviation σ_k as a hyperparameter, which we choose to be $\sigma_k = 0.1$ for all k.

According to Neal (1996) the optimal way to predict the target associated with various new examples, assuming we want to minimize the expected squared error, is to predict the mean of the predictive distribution in equation 4.1.2. For a regression model defined by equation 4.3.1 this is equal to predicting

$$\hat{\boldsymbol{y}}^{(n+1)} = \int f\left(\boldsymbol{x}^{(n+1)}, \boldsymbol{\theta}\right) p\left(\boldsymbol{\theta} \mid \left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \ldots, \left(\boldsymbol{x}^{(n+1)}, \boldsymbol{y}^{(n+1)}\right)\right) d\boldsymbol{\theta}$$

As we do not know this distribution, we resort to a Monte Carlo approximation by averaging over the outputs from the six neural networks, obtained by sampling parameters from the posterior. The average is shown in figure 4.1 by the solid line. But Bayesian inference can do more than a single-valued guess. By examining the function we can also see the uncertainty of the guesses, for example how rapidly uncertainty increases beyond the region of the data points.

This illustrates some of the benefits of using Bayesian inference for neural networks, but what remains is the downside of computational time. Generating 10^5 samples to get six draws from the posterior is not very efficient and this only becomes more infeasible as the number of data points increase.

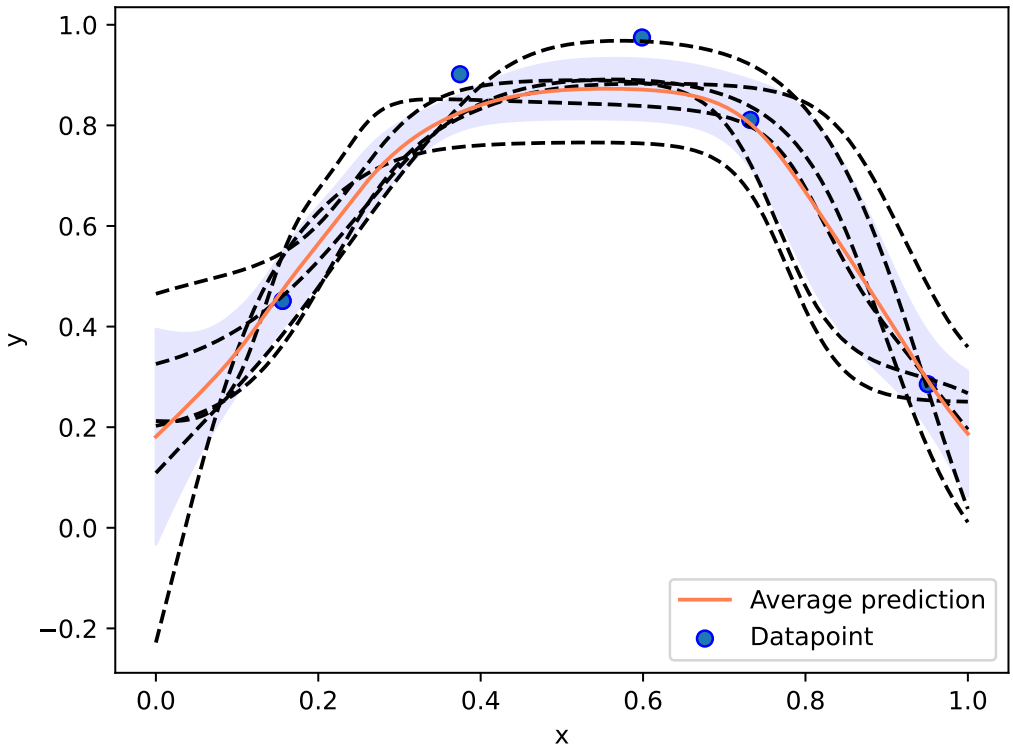


Figure 4.1: Sampled neural networks from a posterior predictive distribution that is based on a Gaussian prior and a Gaussian likelihood on the six data points. The average prediction of the networks are plotted along with a filled area defined by the average plus minus the standard deviation of the network predictions to represent uncertainty. The Python code for implementing this Bayesian neural network and the production of this figure can be seen in appendix .3

4.4 Markov Chain Monte Carlo

As mentioned in section 4.1 the posterior distribution is often intractable and we might need simulation based methods to find a feasible solution. We want to be able to calculate posterior summaries like $\mathbb{E}_{\hat{p}}[f(\boldsymbol{\theta})]$ where $f(\boldsymbol{\theta})$ is some function over the parameters and the expectation is taken under the posterior distribution. Such an expectation is straightforward to approximate using the simple Monte Carlo method, described in section 4.2, when dealing with nice and low dimensional distributions. Other methods like importance sampling has often been proposed in the litterateur for simple problems, but rarely for high-dimensional problems as we often face with neural networks.

The simple example in section 4.3 illustrated that brute force methods are also not very useful in complex models like BNNs and we are instead motivated to use a collection of more sophisticated methods. This thesis will mainly pursue this motivation by exploring the Markov chain Monte Carlo (MCMC) methods. The main idea is to simulate a Markov chain, which has the posterior distribution as its stationary distribution. We will initially give a concise explanation of the fundamentals of Markov chains in section 4.4.1 and follow this by exploring the most popular MCMC algorithms for sampling in BNNs.

4.4.1 Markov Chains

A stochastic process is a set of random variables that are defined over some probability space $\left\{X^{(t)}\right\}_{t\in T}$, where $T \subseteq \mathbb{R}$ is the indexation and may be interpreted as a time index. In the context of machine learning, the set T can be interpreted as the iterations of a simulation scheme. In stochastic simulation we typically have that $T = \mathbb{N}$ and we can write the stochastic process as $\left\{X^{(n)}\right\}_{n\geq 0}$, where n reminds us that we have a discrete index set. Throughout this thesis we will only consider discrete time stochastic processes, since this is found to be most relevant when discussing stochastic simulation.
A stochastic process is defined over a space of possible values called the state space $\mathbb{S}$ and will in most applications be either integers or real values.

With the objective of modelling BNNs, we want to sample model parameters from the posterior distribution. These samples will then be used to make predictions for unseen

data by approximating the posterior predictive distribution from equation 4.1.2 via Monte Carlo integration. In order to do this using MCMC we will consider the stochastic process $\{\boldsymbol{\theta}^{(n)}\}_{n\geq 0}$. To make the description of MCMC methods more general in the next sections, we will denote the distribution we aim to sample from the target distribution and denote it $\hat{p}(\cdot)$. The target distribution in the case of BNNs is the posterior of the weights such that $\hat{p}(\boldsymbol{\theta}) \equiv p(\boldsymbol{\theta} \mid \boldsymbol{X}, \boldsymbol{y})$.

Using Markov chain Monte Carlo we aim to sample from a stochastic process satisfying the Markov property. Such a process is defined by being dependant only on the previous state of the process and a set of transitional probabilities, or densities for a infinite state space, and is called a Markov chain. The Markov chain is defined by an initial distribution for the first state of the chain $\boldsymbol{\theta}^{(0)}$, and a transition density for the next states in the system. We write the transition density from transitioning from state $\boldsymbol{\theta}^{(n-1)}$ to another state $\boldsymbol{\theta}^{(n)}$ as $T(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)})$.

When sampling from the Markov chain we want to make sure that samples are actually coming from the desired distribution $\hat{p}(\boldsymbol{\theta})$ no matter the initial distribution. To ensure this we must generate a Markov chain that has our desired distribution $\hat{p}\,(\boldsymbol{\theta})$ as a stationary distribution. That is if $\boldsymbol{\theta}^{(n-1)}$ has distribution $\hat{p}(\cdot)$, then $\boldsymbol{\theta}^{(n)}$ will have the same distribution, and this will hold for all future states of the chain. The property of a Markov chain having a stationary distribution is called the invariance property and is defined by

$$\pi(\boldsymbol{\theta}^{(n)}) = \int T(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta})\pi(\boldsymbol{\theta})d\boldsymbol{\theta}$$

A sufficient, but not necessary condition, that ensures that a particular $\hat{p}(\boldsymbol{\theta})$ is stationary distribution is the detailed balance condition. The condition states if we let $T(\cdot,\cdot)$ be a transition density, which satisfies the following condition

$$T(\boldsymbol{\theta}^{(n-1)}, \boldsymbol{\theta}^{(n)})\hat{p}(\boldsymbol{\theta}^{(n)}) = T(\boldsymbol{\theta}^{(n)}, \boldsymbol{\theta}^{(n-1)})\hat{p}(\boldsymbol{\theta}^{(n-1)})$$

then $\hat{p}(\cdot)$ is a stationary distribution of the Markov chain associated with the transition density $T(\cdot,\cdot)$. This property however only ensures that $\hat{p}\,(\boldsymbol{\theta})$ is stationary distribution but not that it is the only one, meaning that our Markov chain can end up sampling from the wrong distribution, even though it satisfies the detailed balance condition.

To guarantee that we sample from $\hat{p}(\boldsymbol{\theta})$ we need to ensure that the Markov chain has only this distribution as a stationary distribution. A Markov chain which has a unique stationary distribution, from which it converges to, from any initial state is called an ergodic Markov chain (see e.g Turkman et al. (2019)). For a Markov chain on a finite state space to be ergodic, it has to be irreducible and aperiodic. The same goes for a Markov chain on a infinite state space, but a slightly stronger condition called Harris recurrence is needed, see Gamerman & Lopes (2006).

Often we discard or burn some of the initial states, since these may not be representative of the desired distribution, as the chain might not have reached the stationary distribution yet. These discarded steps are part of what is called the burn-in period of the Markov chain. When the chain has reached the stationary distribution, it is possible to draw as many identical distributed samples as we wish for, but one should note that any successive samples will be highly correlated with each other and therefore not necessarily a good representative for the target distribution. Goodfellow et al. (2016) suggest a way to mitigate this problem, by only returning every n successive sample. Because of both the burn-in period and the time required for the chain to return uncorrelated samples MCMCs are often computational expensive.

In order to produce truly independent samples, Goodfellow et al. (2016) suggest to run multiple Markov chains in parallel. They also mention that practitioners often chooses the number of chains to run in parallel, similar to the number of examples in a mini-batch and then draw the samples needed from this set of Markov chains.

4.4.2 The Metropolis algorithm

The Metropolis algorithm is a MCMC method, which is used to sample from a distribution. The Metropolis algorithm was originally introduced by Metropolis et al. (1953) and was developed to simulate the states for a system of molecules. This was later further developed by Hastings (1970), so that the algorithm could now simulate from a general distribution and not just a symmetric one, as it was previously based on. The Metropolis algorithm is an attractive MCMC method due to its versatility and simplicity.

The algorithm considers a target distribution $\hat{p}(\boldsymbol{\theta})$ and a proposal distribution $q(\boldsymbol{\theta})$. The algorithm generates a Markov chain, by starting the chain at some arbitrarily point generated from the proposal distribution $\boldsymbol{\theta}^{(0)} \sim q(\boldsymbol{\theta})$ and then proposing a candidate state for the next state in the chain $\boldsymbol{\theta}^{\text{cand}}$, where this candidate state is drawn from the conditional distribution on the previous state $\boldsymbol{\theta}^{\text{cand}} \sim q(\boldsymbol{\theta}^{(n+1)} \mid \boldsymbol{\theta}^{(n)})$. The next step is to decide whether or not to reject this new state based on the relative density to the old state. If the relative density is larger than one, we accept the new state, if the relative density is less than one, we accept the new state with probability $\frac{\hat{p}(\boldsymbol{\theta}^{\text{cand}})}{\hat{p}(\boldsymbol{\theta}^{(n)})}$. In this context the Metropolis algorithm imposes the symmetry condition of on the proposal distribution, so that

$$q(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}) = q(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{(n)})$$

A pseudocode version of the Metropolis algorithm can be seen in algorithm 6. One apparent

Algorithm 6: Metropolis algorithm

Input: A proposal distribution q

Output: A set of parameters $\boldsymbol{\theta}^{(n)}$ for $n = 1, \ldots, N$

initialize $\boldsymbol{\theta}^{(0)} \sim q(\boldsymbol{\theta})$;

for $n = 1, 2, \ldots, N$ **do**

  Propose: $\boldsymbol{\theta}^{\text{cand}} \sim q\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right)$

  Acceptance Probability:

  $\alpha\left(\boldsymbol{\theta}^{\text{cand}} \mid \boldsymbol{\theta}^{(n-1)}\right) = \min\left\{1, \frac{\hat{p}\left(\boldsymbol{\theta}^{\text{cand}}\right)}{\hat{p}\left(\boldsymbol{\theta}^{(n-1)}\right)}\right\}$

  $u \sim$ Uniform $(0, 1)$

  if $u < \alpha$ **then**

    Accept the proposal: $\boldsymbol{\theta}^{(n)} \leftarrow \boldsymbol{\theta}^{\text{cand}}$;

  else

    Reject the proposal: $\boldsymbol{\theta}^{(n)} \leftarrow \boldsymbol{\theta}^{(n-1)}$;

  end

end

problem is that due to the evidence term we can not calculate the posterior exactly, which is our target distribution $\hat{p}\,(\boldsymbol{\theta})$, so we can not directly calculate $\frac{\hat{p}(\boldsymbol{\theta}^{(n)})}{\hat{p}\left(\boldsymbol{\theta}^{(n-1)}\right)}$. But a nice property of Metropolis acceptance probability is that we only need a function that is proportional

to the posterior, as any constant of proportionality will cancel out in the calculation of the acceptance probability. As the evidence term can be interpreted as a constant of proportionality, it will cancel out and we can instead use equation 4.1.1 and calculate the ratio of the likelihood times the prior, which we are often able to.

To show that the Metropolis algorithm is in fact sampling from the target distribution, we must show that the Markov chain converges to a stationary distribution which is our target distribution. If we assume that the Markov Chain is ergodic we can do this by showing that the metropolis satisfies the detailed balanced condition explained in section 4.4.

For $\boldsymbol{\theta}^{(n)} \neq \boldsymbol{\theta}^{(n-1)}$ the Metropolis algorithm has the following transitions densities,

$$T\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right) = q\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right) \min\left(1, \frac{\hat{p}\left(\boldsymbol{\theta}^{(n)}\right)}{\hat{p}(\boldsymbol{\theta}^{(n-1)})}\right)$$

We can show that this satisfies the detailed balanced condition by

$$\begin{aligned}
T\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right)\hat{p}(\boldsymbol{\theta}^{(n-1)}) &= q\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right) \min\left(1, \frac{\hat{p}\left(\boldsymbol{\theta}^{(n)}\right)}{\hat{p}(\boldsymbol{\theta}^{(n-1)})}\right)\hat{p}(\boldsymbol{\theta}^{(n-1)}) \\
&= q\left(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}\right) \min\left(\hat{p}(\boldsymbol{\theta}^{(n-1)}), \hat{p}\left(\boldsymbol{\theta}^{(n)}\right)\right) \\
&= q\left(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{(n)}\right) \min\left(\hat{p}\left(\boldsymbol{\theta}^{(n-1)}\right), \hat{p}(\boldsymbol{\theta}^{(n-1)})\right) \\
&= q\left(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{(n)}\right) \min\left(1, \frac{\hat{p}(\boldsymbol{\theta}^{(n-1)})}{\hat{p}\left(\boldsymbol{\theta}^{(n)}\right)}\right)\hat{p}\left(\boldsymbol{\theta}^{(n)}\right) \\
&= T\left(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{(n)}\right)\hat{p}\left(\boldsymbol{\theta}^{(n)}\right)
\end{aligned}$$

This shows that the transitions proposed by the algorithm leaves the target distribution $\hat{p}(\boldsymbol{\theta})$ invariant and therefore samples produced by this Markov chain all has the same stationary distribution $\hat{p}(\boldsymbol{\theta})$, provided that the Markov chain is ergodic. However according to Neal (1996) the Metropolis algorithm will not always produce an ergodic Markov chain and that this depends on the details of the target distribution and proposal distribution. If the produced Markov chain is not ergodic we might end up sampling from a stationary distribution that is not our target distribution.

There are many possible choices for the proposal distribution. Neal (1996) mentions that

a simple choice could be a Gaussian centered on $\boldsymbol{\theta}^{(n)}$ with standard deviation chosen so that the acceptance probability of the candidate state is reasonably high, since a very low acceptance ratio is usually unwanted, as many rejections means that we are inefficiently wasting computation time. He also notes that when sampling from high-dimensional and complex distributions, which is often the case with posteriors in BNNs, the standard deviation of such a proposal distribution will often have to be small compared to the extent of the target distribution, as large changes almost certainly will lead to a region of low probability. This will result in highly dependant states and many steps will be needed to arrive at distant points in the distribution. As suggested in section 4.4.1 one way of coping with this problem is to throw away some of the samples or run multiple chains in parallel.

Neal (1996) further mentions that this problem with Metropolis is made worse due to its movements taking the inefficient form of a random walk instead of a systematic path, as can be seen in the illustration of a sampling with Metropolis in figure 4.2. This inefficient movement yields slower convergence to the target distribution. This drawback, will be even more prominent in higher dimension and for more complex target distributions according to Gelman et al. (2004).

A more generalized version of the algorithm is the one introduced by Hastings (1970), which allows for non-symmetric proposal distributions $q(\boldsymbol{\theta}^{(n)} \mid \boldsymbol{\theta}^{(n-1)}) \neq q(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{(n)})$. To correct for this asymmetry in the proposal distribution, the acceptance ratio is replaced by

$$\alpha\left(\boldsymbol{\theta}^{cand} \mid \boldsymbol{\theta}^{(n-1)}\right) = \min\left\{1, \frac{q\left(\boldsymbol{\theta}^{(n-1)} \mid \boldsymbol{\theta}^{cand}\right)\hat{p}\left(\boldsymbol{\theta}^{cand}\right)}{q\left(\boldsymbol{\theta}^{cand} \mid \boldsymbol{\theta}^{(n-1)}\right)\hat{p}\left(\boldsymbol{\theta}^{(n-1)}\right)}\right\} \tag{4.4.1}$$

One should note that the Metropolis algorithm is an instance of the generalized version, since equation 4.4.1 is identical to the acceptance ratio in the original algorithm when allowance for a symmetric distribution.

The introduction of an asymmetric proposal distribution, is often useful when we want to increase the speed of the random walk generated by the Metropolis algorithm. However, this is often not sufficient for complicated models with high-dimensional target distributions as we face with Bayesian neural networks, see Gelman et al. (2004).

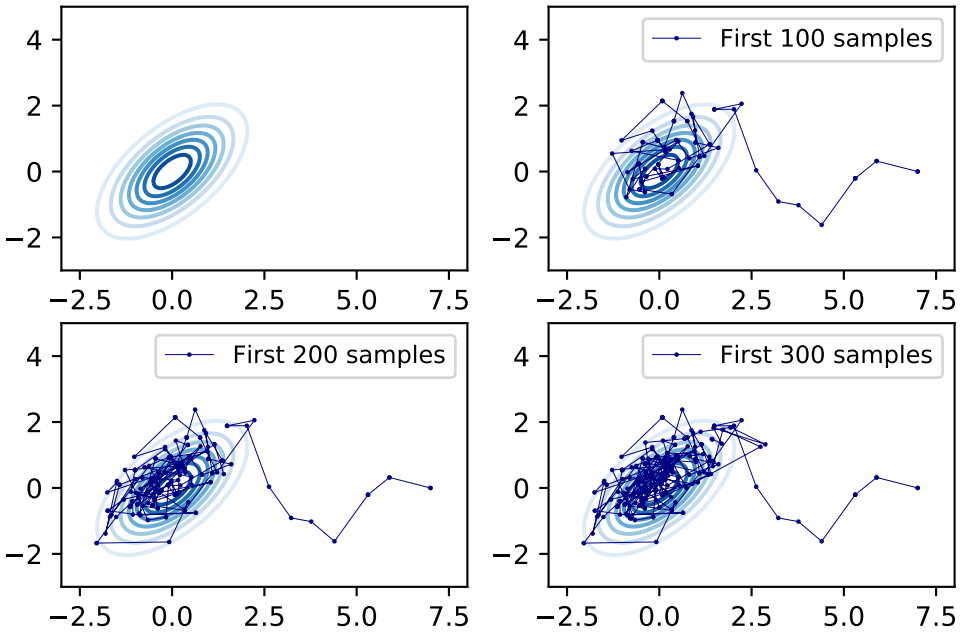


Figure 4.2: Illustration of the convergence to a target distribution with 300 samples from the Metropolis algorithm. The target distribution is a bivariate Gaussian with mean $\boldsymbol{\mu} = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and covariance matrix $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0.6 \\ 0.6 & 1 \end{bmatrix}$. The Python code for producing this figure can be found in appendix .4.

4.5 Hamiltonian Monte Carlo

Another way to generate proposals with a higher efficiency, is by updating the parameters by dynamical simulation and then use the Metropolis algorithm to accept or reject these proposals. Such an algorithm suppress the local random walk behavior and allow it to move faster and more rapidly through the target distribution. This method is called Hamiltonian Monte Carlo (HMC), and is commonly used in computational physics and statistics. The algorithm was originally proposed by Duane et al. (1987) for calculations used in lattice quantum chromodynamics, but was later introduced to the field of computational statistics when it was used for Bayesian neural networks in Neal (1996). This means that the Markov chain from which we sample in BNNs is produced analogous to paths of particles using Hamiltonian dynamics, and we will explain these dynamics using this physical analogy, as it gives a more intuitive idea of HMC.

It turns out that the HMC algorithm reduces correlation between successive sampled states, compared to the Metropolis Hastings algorithm in section 4.4.2, by proposing moves to distant states of the target distribution, which maintain a high probability of acceptance due to the properties of the simulated Hamiltonian dynamics. The reduced correlation imply that fewer Markov chain samples are needed to approximate integrals with respect to the target probability distribution.

4.5.1 Hamiltonian Dynamics

Before we move to the actual HMC algorithm, we will explain the Hamiltonian dynamics from which we produce the Markov chain for the algorithm. The Hamiltonian dynamics are used to describe how an object move around in a system or space. It is defined by the objects position $\boldsymbol{q} \in \mathbb{R}^d$ and its momentum $\boldsymbol{\rho} \in \mathbb{R}^d$, which in the field of physics is equivalent to an object's mass times its velocity at some point in time. When performing HMC to generate samples from the target distribution $\hat{p}(\boldsymbol{\theta})$, the position variable plays the role of the parameter vector and we will from now on let $\boldsymbol{\theta} \equiv \boldsymbol{q}$. The object's position is associated with a potential energy $U(\boldsymbol{\theta})$ and the momentum is associated with a kinetic energy $K(\boldsymbol{\rho})$. The sum of the potential and kinetic energy is regarded as the total energy

of the system often called the Hamiltonian

$$H(\boldsymbol{\theta}, \boldsymbol{\rho}) = U(\boldsymbol{\theta}) + K(\boldsymbol{\rho})$$

An important property of the Hamiltonian is that it conserves the sum of the potential and kinetic energy, meaning that it is constant over time. Taking the partial derivative with respect to time of the position and momentum shows us how they evolve over time

$$\begin{aligned} \frac{d\theta_i}{dt} &= \frac{\partial H}{\partial \rho_i} = \frac{\partial K(\boldsymbol{\rho})}{\partial \rho_i} \\ \frac{d\rho_i}{dt} &= -\frac{\partial H}{\partial \theta_i} = -\frac{\partial U(\boldsymbol{\theta})}{\partial \theta_i} \end{aligned} \tag{4.5.1}$$

for $i = 1, 2, \ldots, d$. These are named Hamiltonian equations and represent a differential equations system. The Hamiltonian equations are useful, since if we are able to evaluate the partial derivatives from equation 4.5.1, we are able to predict the position and momentum variables of the object at any point in the future $t' > t$. Neal (1996) shows that these dynamics, along with the energy conserving property of the Hamiltonian, results in the process being reversible and preserving volume of the state space, which in turn provides a stationary distribution.

4.5.2 Discretizing Hamiltonian Equations

The Hamiltonian equations describe how an objective evolve in continuous time, but for simulating Hamiltonian dynamics on a computer we have to approximate the differential equations which is done by discretizing time. We do this by splitting the time interval dt into smaller intervals ε.

The usual discretizing scheme for simulating Hamiltonian equations, is the Leapfrog method. The Leapfrog method takes a half step to update momentum variable then a whole step to update the position value, and finally the last half step to update momentum

$$\begin{aligned} \rho_i^{(t+\varepsilon/2)} &= \rho_i^{(t)} - (\varepsilon/2)\frac{\partial U}{\partial \theta_i^{(t)}} \\ \theta_i^{(t+\varepsilon)} &= \theta_i^{(t)} + \varepsilon \frac{\partial K}{\partial \rho_i^{(t+\varepsilon/2)}} \\ \rho_i^{(t+\varepsilon)} &= \rho_i^{(t)} - (\varepsilon/2)\frac{\partial U}{\partial \theta_i^{(t+\varepsilon)}} \end{aligned}$$

According to Neal (2012) the Leapfrog method preserves the HMC properties of being reversible and preserve volume of the state space, ensuring that we sample from a stationary distribution.

4.5.3 The Hamiltonian and Probability Distributions

We have now explained what a Hamiltonian is and how we can simulate its dynamics by using the Leapfrog method. We will now connect this to MCMC theory from the previous sections in order to explain how to use HMC to sample from the posterior of the parameters in a BNN. In order to perform this connection, we need to relate the target distribution and the Hamiltonian, such that we can use the Hamiltonian equations to sample from the target distribution. A way of doing this, proposed by Neal (1996), is to use a concept from statistical mechanics known as the canonical (Boltzmann) distribution. We can write a probability distribution on $\boldsymbol{\theta}$ under the canonical distribution as

$$p(\boldsymbol{\theta}) \propto \exp\left(\frac{-E(\boldsymbol{\theta})}{T}\right)$$

where $E(\boldsymbol{\theta})$ can be any energy function defined over $\boldsymbol{\theta}$. T is often called the temperature of the system and usually chosen to be equal to 1 as it plays no role in this application, see Neal (1996). One should note that any probability distribution that is nowhere zero can be put into this form by letting $E(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta}) - \log Z$, for any convenient choice of normalization constant Z. Since the Hamiltonian is an energy function for the joint state of both position and momentum, a joint distribution can be defined by

$$p(\boldsymbol{\theta}, \boldsymbol{\rho}) \propto \exp(-\mathrm{H}(\boldsymbol{\theta}, \boldsymbol{\rho})) = \exp(-U(\boldsymbol{\theta}))\exp(-K(\boldsymbol{\rho}))$$

Assuming Independence between $\boldsymbol{\theta}$ and $\boldsymbol{\rho}$ we can by the equation above write $U(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta})$ and $K(\boldsymbol{\rho}) = -\log p(\boldsymbol{\rho})$ meaning that the Hamiltonian can be interpreted as the log joint distribution on $(\boldsymbol{\theta}, \boldsymbol{\rho})$.

We now have a joint distribution, in terms of the Hamiltonian function, which we know how to simulate from. But we are in fact only interested in samples of the position variable $\boldsymbol{\theta}$, which is samples from our target distribution, and not samples from the momentum variable $\boldsymbol{\rho}$, which is only introduced to make the algorithm move faster through the parameter space. This means that we can choose the marginal distribution of the momentum

arbitrarily. This is often chosen by practitioners to be Gaussian, $\boldsymbol{\rho} \sim \mathcal{N}(0, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$ is some symmetric, positive-definite covariance matrix and often chosen to be diagonal, such that $\boldsymbol{\rho}$ is d-dimensional multivariate Gaussian, with the d variables being independent. We follow the simple approach from Hoffman & Gelman (2011) and let $\boldsymbol{\Sigma}$ be the identity matrix $\boldsymbol{I}$. This makes the dynamics of equation 4.5.1 simplify to

$$\begin{aligned} \frac{d\theta_i}{dt} &= \rho_i \\ \frac{d\rho_i}{dt} &= \frac{\partial \log p(\theta_i)}{\partial \theta_i} \end{aligned}$$

A more rigorous examination of possible choices for the covariance matrix is provided by Neal (2012).

4.5.4 The Hamiltonian Monte Carlo Algorithm

We start the HMC algorithm from an initial state $\left(\boldsymbol{\theta}^{(0)}, \boldsymbol{\rho}^{(0)}\right)$, and then we simulate the Hamiltonian dynamics for $t + \varepsilon$ using the Leapfrog method. The states generated for the position and momentum variables at the end of the Leapfrog simulation is used as proposals for a new state $(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}})$. These states needs to be accepted according to a criterion, because the leapfrog-discretization provides an error term in its approximation of the continuous Hamiltonian dynamics. This criterion is the Metropolis acceptance criterion,

$$\begin{aligned} \alpha\left((\boldsymbol{\theta}, \boldsymbol{\rho}) \mapsto (\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}})\right) &= \min\left\{1, \frac{p(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}})}{p(\boldsymbol{\theta}, \boldsymbol{\rho})}\right\} \\ &= \min\{1, \exp\left(\log p(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}}) - \log p(\boldsymbol{\theta}, \boldsymbol{\rho})\right) \\ &= \min\left\{1, \exp\left(-\mathrm{H}(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}}) + \mathrm{H}(\boldsymbol{\theta}, \boldsymbol{\rho})\right)\right\} \end{aligned} \tag{4.5.2}$$

This means that we follow the same logic as in the Metropolis algorithm, but use distributions provided by the Hamiltonian dynamics. With $\boldsymbol{\rho} \sim \mathcal{N}(0, \boldsymbol{I})$ this is equivalent to

$$\alpha\left((\boldsymbol{\theta}, \boldsymbol{\rho}) \mapsto (\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}})\right) = \min\left\{1, \frac{\exp\left(\mathcal{L}(\boldsymbol{\theta}^{\text{cand}}) - \frac{1}{2}\boldsymbol{\rho}^{(\text{cand})^\top}\boldsymbol{\rho}^{(\text{cand})}\right)}{\exp\left(\mathcal{L}(\boldsymbol{\theta}) - \frac{1}{2}\boldsymbol{\rho}^\top\boldsymbol{\rho}\right)}\right\}$$

where $\mathcal{L}(\boldsymbol{\theta})$ is the log-probability distribution on $\boldsymbol{\theta}$.

We see from the last part in equation 4.5.2 that if we could perfectly discretize the Hamiltonian dynamics, the Metropolis acceptance criterion would always be equal to 1 due to the energy conservation property of the Hamiltonian. Since this is usually not possible, the

Metropolis acceptance criterion will often be lower than 1. We can see that if we get a small error in the discretization the term $\mathrm{H}(\boldsymbol{\theta}, \boldsymbol{\rho}) - \mathrm{H}(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}})$ in the exponent should be small, yielding a high acceptance rate. This way of making proposals is beneficial since it allows the Markov chain to effectively make large and uncorrelated moves in the state space, while keeping a high acceptance probability. HMC is written in pseudocode in algorithm 7. It takes in an initial value for the parameters $\boldsymbol{\theta}^{(0)}$, which is the starting point of the algorithm. The input $\mathcal{L}$ is the log-probability distribution of $\boldsymbol{\theta}$, which is defined to be equal to the negative potential energy function. In BNN this is identical to the log-posterior distribution on the neural network parameters. One need to be able to at least evaluate the posterior distribution and its gradient or alternatively something proportional to it. In our case where the target distribution is the posterior distribution, we know that it is proportional to the likelihood times the prior, which we in most cases are able to evaluate. The M input is the number of total iterations one would like to perform.

The algorithm also relies on a stepsize ε variable, that defines the size of the leapfrog step. If ε is chosen too be too large, the leapfrog simulation of the Hamiltonian will be inaccurate and lead to a very low rate of acceptance making the algorithm ineffectively waste of computational time. On the other hand, if we choose ε to be too small, we will waste computational time by taking too small steps. The sampling is also affected by a hyperparameter L, that defines how many leapfrog steps the algorithm performs before proposing a new candidate state. A very small value for L will give successive samples that lie close to each other which results in the same undesirable random walk behavior as the Metropolis algorithm from section 4.4.2. Too large a value for L might produce trajectories that loop back and retrace their steps, a behavior called U-turns. This behavior results in the algorithm inefficiently wasting time, sampling from the same area of the distribution again and again. Tuning these parameters can be hard and one are often forced to rely on heuristics based preliminary runs, see Neal (2012).

In figure 4.3 we have illustrated how HMC propose a candidate sample for a bivariate $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ target distribution. In subfigure (a) and (b) we have chosen a proper value for ε and L such that the algorithm generates proposals that are appropriately far from the previous ones. In subfigure (c) and (d) we have chosen larger values for ε and L, and we

can see that the algorithm starts to loop-back, which results in identical proposals for each iteration and a large proportion of the target distribution will therefore never be visited. In the next section we will look into a modification of the HMC algorithm, so that we can avoid this kind of U-turn behavior.

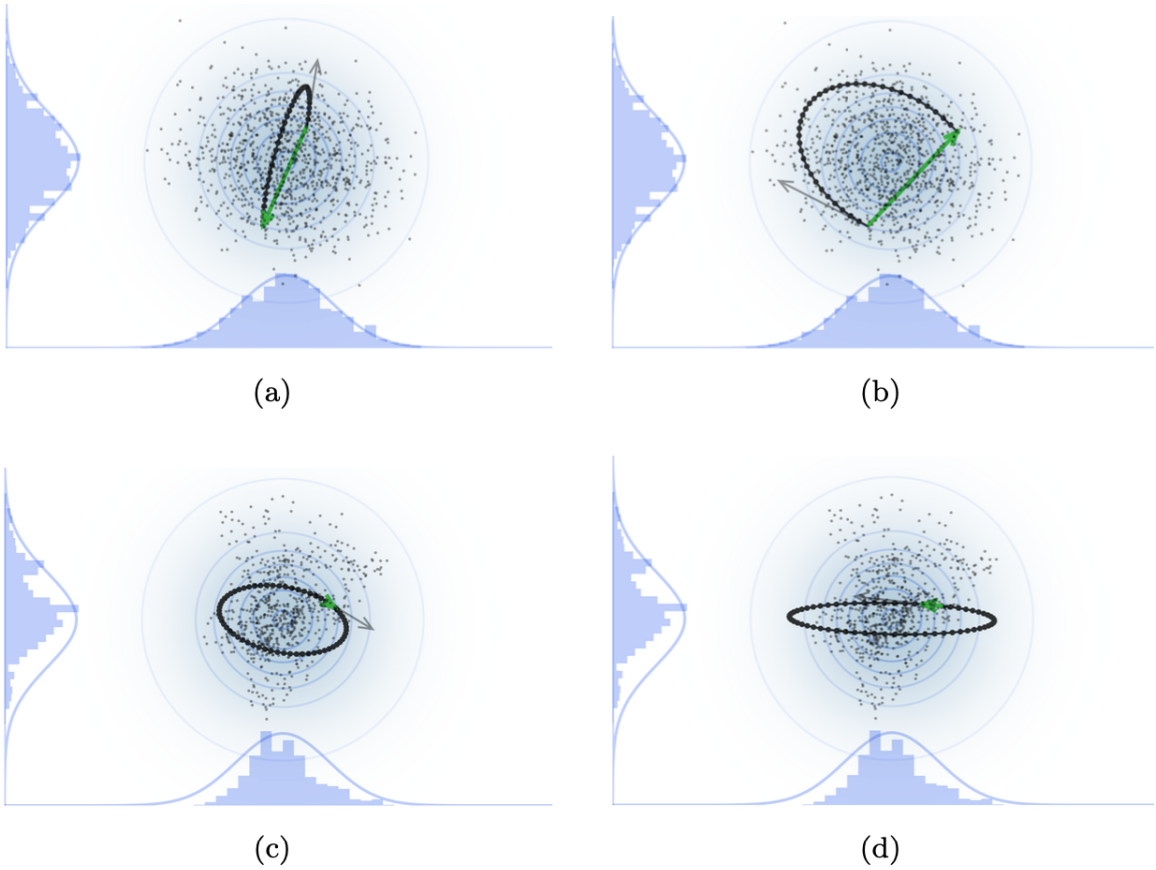


Figure 4.3: A simulation example with HMC for a bivariate Gaussian target distribution with $\boldsymbol{\mu} = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and covariance matrix $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Subfigure (a) and (b) shows a HMC simulation example with a proper choice for ε and L, where the target distribution is thoroughly explored. Subfigure (c) and (d) is the same example, modified with a poor choice of values for ε and L, where the target distribution is poorly approximated. This is especially clear on the plots of the marginal distributions where the histograms are far from the plotted correct distribution. The result is the U-Turn effect, which yield an ineffective exploration of the target distribution. The example has been generated with the interactive gallery provided by Feng (2016) ©MIT.

Algorithm 7: Hamiltonian Monte Carlo

Input: An intial parameter $\boldsymbol{\theta}^{(0)}$

Input: A log-probability distribution $\mathcal{L}$

Input: Total number of iterations M

Input: Size of leapfrog-steps ε

Input: Number of leapfrog-steps before generating candidate state, L.

Output: Set of accepted parameters

Given $\boldsymbol{\theta}^{(0)}, \mathcal{L}, M, \varepsilon, L,$

for *m=1,2 . . . , M* **do**

Sample $\boldsymbol{\rho}^{(0)} \sim \mathcal{N}(0, \boldsymbol{I})$

Set $\boldsymbol{\theta}^{\text{cand}} \leftarrow \boldsymbol{\theta}^{(m-1)}, \boldsymbol{\rho}^{\text{cand}} \leftarrow \boldsymbol{\rho}^{(0)}$

for *i = 1 to L* **do**

Set $\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}} \leftarrow \texttt{Leapfrog}(\boldsymbol{\theta}^{\text{cand}}, \boldsymbol{\rho}^{\text{cand}}, \varepsilon)$

end

Negate momentum: $\boldsymbol{\rho}^{\text{cand}} \leftarrow -\boldsymbol{\rho}^{\text{cand}}$

Compute acceptance probability:

$$\alpha = \min\left\{1, \frac{\exp\left\{\mathcal{L}(\boldsymbol{\theta}^{\text{cand}}) - \frac{1}{2}\boldsymbol{\rho}^{(\text{cand})^\top}\boldsymbol{\rho}^{\text{cand}}\right\}}{\exp\left\{\mathcal{L}\left(\boldsymbol{\theta}^{(m-1)}\right) - \frac{1}{2}\boldsymbol{\rho}^{(0)^\top}\boldsymbol{\rho}^{(0)}\right\}}\right\}$$

Sample $u \sim \text{Uniform}(0, 1)$

if $u < \alpha$ **then**

Accept the proposal: $\boldsymbol{\theta}^{(m)} \leftarrow \boldsymbol{\theta}^{cand}$

else

Reject the proposal: $\boldsymbol{\theta}^{(m)} \leftarrow \boldsymbol{\theta}^{(m-1)}$

end

end

Function `Leapfrog(`$\boldsymbol{\theta}$, $\boldsymbol{\rho}$, ε`)`**:**

Set $\boldsymbol{\rho}^{\text{cand}} \leftarrow \boldsymbol{\rho}^{\text{cand}} + (\varepsilon/2)\nabla_\theta \mathcal{L}(\theta)$

Set $\boldsymbol{\theta}^{\text{cand}} \leftarrow \theta + \varepsilon\boldsymbol{\rho}^{\text{cand}}$

Set $\boldsymbol{\rho}^{\text{cand}} \leftarrow \boldsymbol{\rho}^{\text{cand}} + (\varepsilon/2)\nabla_\theta \mathcal{L}(\boldsymbol{\theta}^{\text{cand}})$

return $\boldsymbol{\theta}^{cand}, \boldsymbol{\rho}^{cand}$

4.5.5 No-U-Turn Hamiltonian Monte Carlo

In this section we present an algorithm introduced by Hoffman & Gelman (2011) and evaluated by Nishio & Arakawa (2019), whose explanation we found more clear and concise. No-U-Turn (NUTS) extends HMC by eliminating the need to specify the trajectory length L. This algorithm gets its name by avoiding the possible U-Turning behavior shown in figure 4.3 (c) and (d). This is done by introducing a criterion that tells us when to stop simulating the dynamics to prevent a possible U-turn. The authors define this criterion to be when performing another leapfrog step will no longer increase the distance between the proposed state $\boldsymbol{\theta}^{\text{cand}}$ and the initial value $\boldsymbol{\theta}^{(0)}$. More specifically, they choose a criterion, based on the derivative with respect to time of half the squared distance between the initial parameter $\boldsymbol{\theta}^{(0)}$ and the current state $\boldsymbol{\theta}^{\text{cand}}$, meaning that leapfrog steps are performed until

$$\begin{aligned}\frac{d}{dt}\frac{(\boldsymbol{\theta}^{\text{cand}}-\boldsymbol{\theta})^\top\cdot(\boldsymbol{\theta}^{\text{cand}}-\boldsymbol{\theta})}{2} &= (\boldsymbol{\theta}^{\text{cand}}-\boldsymbol{\theta})^\top\cdot\frac{d}{dt}(\boldsymbol{\theta}^{\text{cand}}-\boldsymbol{\theta})\\ &= (\boldsymbol{\theta}^{\text{cand}}-\boldsymbol{\theta})^\top\cdot\boldsymbol{\rho}^{\text{cand}} < 0\end{aligned}$$

However, Hoffman & Gelman (2011) notes that by doing this we do not have the guarantee of time reversibility, so the sampling algorithm might not converge to the correct distribution. NUTS overcomes this problem by using slice sampling and applying a double method suggested by Neal (2003).

NUTS augments the distribution of HMC, $p(\boldsymbol{\theta},\boldsymbol{\rho}) \propto \exp\left(\mathcal{L}(\boldsymbol{\theta}) - \frac{1}{2}\boldsymbol{\rho}^\top\boldsymbol{\rho}\right)$, to include a slice variable u so that the joint probability of $\boldsymbol{\theta}, \boldsymbol{\rho}$ and u is

$$p(\boldsymbol{\theta},\boldsymbol{\rho},u) \propto \mathbf{1}\left[u \in \left[0, \exp\left(\mathcal{L}(\boldsymbol{\theta}) - \frac{1}{2}\boldsymbol{\rho}^\top\boldsymbol{\rho}\right)\right]\right]$$

meaning that the un-normalized marginal probability of $\boldsymbol{\theta}$ and $\boldsymbol{\rho}$ (gained by integrating over u) is

$$p(\boldsymbol{\theta},\boldsymbol{\rho}) \propto \exp\left(\mathcal{L}(\boldsymbol{\theta}) - \frac{1}{2}\boldsymbol{\rho}^\top\boldsymbol{\rho}\right)$$

The conditional probabilities $p(u \mid \boldsymbol{\theta},\boldsymbol{\rho})$ and $p(\boldsymbol{\theta},\boldsymbol{\rho} \mid u)$ are each uniform as long as the condition

$$u \leq \exp\left(\mathcal{L}(\boldsymbol{\theta}) - \frac{1}{2}\boldsymbol{\rho}^\top\boldsymbol{\rho}\right) \tag{4.5.3}$$

is satisfied. The challenge of slice sampling is to find the bounds of the region for which this condition is satisfied. Neal (2003) proposes a doubling method, where the size of the initial segment containing the current value of $\boldsymbol{\theta}$ is randomly chosen and afterwards expanded by

doubling its size until the samples are outside of the region. The expanding directions are randomly chosen to be leapfrog steps forward or backward in the Markov chain to satisfy reversibility.

NUTS generates a finite set of all $(\boldsymbol{\theta}, \boldsymbol{\rho})$ by iteratively doubling its size. The doubling process is stopped to satisfy the condition

$$\left(\boldsymbol{\theta}^{+} - \boldsymbol{\theta}^{-}\right)^{\top} \boldsymbol{\rho}^{-} < 0 \quad \text{or} \quad \left(\boldsymbol{\theta}^{-} - \boldsymbol{\theta}^{+}\right)^{\top} \boldsymbol{\rho}^{+} < 0$$

where $\boldsymbol{\theta}^{+}, \boldsymbol{\rho}^{-}$ and $\boldsymbol{\theta}^{+}, \boldsymbol{\rho}^{-}$ is the leftmost and rightmost variables of all $(\boldsymbol{\theta}, \boldsymbol{\rho})$ generated by the doubling process respectively.

A subset of proposal candidates $(\boldsymbol{\theta}, \boldsymbol{\rho})$, denoted by $\mathbb{C}$, is selected from the doubling process to satisfy the condition in equation 4.5.3. The new values of $(\boldsymbol{\theta}^{*}, \boldsymbol{\rho}^{*})$ are then afterwards sampled uniformly from $\mathbb{C}$.

To further improve this algorithm Hoffman & Gelman (2011) used the following transition kernel in each step of doubling

$$T\left(\boldsymbol{\theta}^{*}, \boldsymbol{\rho}^{*} \mid \boldsymbol{\theta}, \boldsymbol{\rho}, \mathbb{C}\right) = \begin{cases} \frac{\mathbf{1}[\boldsymbol{\theta}^{*}, \boldsymbol{\rho}^{*} \in \mathbb{C}^{\text{new}}]}{|\mathbb{C}^{\text{new}}|} & \text{when } |\mathbb{C}^{\text{new}}| > \left|\mathbb{C}^{\text{old}}\right| \\ \frac{|\mathbb{C}^{\text{new}}|}{\left|\mathbb{C}^{\text{old}}\right|} \frac{\mathbf{1}[\boldsymbol{\theta}^{*}, \boldsymbol{\rho}^{*} \in \mathbb{C}^{\text{new}}]}{|\mathbb{C}^{\text{new}}|} + \left(1 - \frac{|\mathbb{C}^{\text{new}}|}{\left|\mathbb{C}^{\text{old}}\right|}\right) \mathbf{1}\left[(\boldsymbol{\theta}^{*}, \boldsymbol{\rho}^{*}) = (\boldsymbol{\theta}, \boldsymbol{\rho})\right] & \text{when } |\mathbb{C}^{\text{new}}| \leq \left|\mathbb{C}^{\text{old}}\right| \end{cases}$$

where $\mathbb{C}^{\text{new}}$ is the subset of $(\boldsymbol{\theta}, \boldsymbol{\rho})$, added by the last step of the doubling process and $\mathbb{C}^{\text{old}}$ is the disjoint subset of $\mathbb{C}$ so that $\mathbb{C} = \mathbb{C}^{\text{new}} \cup \mathbb{C}^{\text{old}}$. This transition kernel proposes a move from a state in $\mathbb{C}^{\text{old}}$ to a random state in $\mathbb{C}^{\text{new}}$ and accepts this move with probability $\frac{\mathbb{C}^{\text{new}}}{\mathbb{C}^{\text{old}}}$. The authors show that T satisfies the detailed balance condition so that it leaves the uniform distribution over $\mathbb{C}$ invariant. According to Nishio & Arakawa (2019) this transitional kernel permits memory-efficient implementation and produces larger jumps on average than simple uniform sampling.

NUTS is especially efficient as it can automatically choose a step size that achieves an acceptance probability around a desired level. The stepsize ε for the jth iteration of a

NUTS Markov chain is tuned as follows

$$\log\left(\varepsilon_{j+1}\right) = \mu - \frac{\sqrt{j}}{\gamma}\frac{1}{j+j_0}\sum_{i=1}^{j}\left(\delta - \alpha_i\right)$$

$$\log\left(\bar{\varepsilon}_{j+1}\right) = \eta_j \log\left(\varepsilon_{j+1}\right) + \left(1-\eta_j\right)\log\left(\bar{\varepsilon}_j\right)$$

$$\varepsilon_{j+1} = \bar{\varepsilon}_{j+1}$$

where α_j is an actual acceptance probability for the jth iteration, δ is the desired average acceptance probability, μ is a freely chosen point that the iterated ε_j shrinks towards, γ is a free parameter that controls the shrinkage amount towards μ and j_0 is a free parameter that dampens early exploration.

Hoffman & Gelman (2011) introduces the variables $\eta_j = j^{-\kappa}$ with $\kappa < 1$ to give more recent iterates more weight. They show that this way of adapting stepsize guarantees that $\alpha \to \delta$. They recommend setting $\mu = \log\left(10\varepsilon_1\right)$ and $\delta \approx 0.6$.

NUTS tunes ε during a predetermined warm-up phase and fixes this value thereafter. Since the algorithm accepts or rejects $(\boldsymbol{\theta}^*, \boldsymbol{\rho}^*)$ from multiple candidates, an alternative statistic to the Metropolis acceptance probability must be defined. They do this by defining for each iteration the acceptance probability by

$$\alpha_j = \frac{1}{|B_j|}\sum_{\boldsymbol{\theta},\boldsymbol{\rho}\in B_j}\min\left\{1, \frac{p\left(\boldsymbol{\theta}^j, \boldsymbol{\rho}^j\right)}{p\left(\boldsymbol{\theta}^{j-1}, \boldsymbol{\rho}^{j,0}\right)}\right\}$$

The pseudocode for NUTS can be seen in algorithm 8.

Algorithm 8: No-U-Turn Sampler with Dual Averaging. One can easily change this pseudocode to one that runs until a certain number of samples are collected. For a more detailed pseudocode see Hoffman & Gelman (2011).

Input: Initial parameters $\boldsymbol{\theta}^{(0)}$

Input: A log-probability distribution $\mathcal{L}$

Input: Initial size of leapfrog-steps $\bar{\varepsilon_0}$

Input: Desired average of acceptance probability δ

Input: Aim point for values of the iterated ε_j values μ

Input: Parameter controlling shrinkage towards μ, γ

Input: Parameter that controls dampening of early exploration j_0

Input: Parameter that controls how much more weight are given to more recent iterates $\kappa < 1$

Input: Total number of total number of iterations J

Input: Total number of iterations for adapting size of leapfrog steps J^{adapt}

Output: Samples from the target distribution

for *j=0, . . . , J* **do**

Sample momentum $\boldsymbol{\rho}^{\text{init}} \sim N(0, \boldsymbol{I})$

Sample auxiliary variable

$u \sim \text{Uniform}\left(0, \exp\left(\mathcal{L}\left(\boldsymbol{\theta}^{(j)}\right) - \frac{1}{2}\boldsymbol{\rho}^{\text{init}^\top}\boldsymbol{I}^{-1}\boldsymbol{\rho}^{\text{init}}\right)\right)$

Generate $\mathbb{C}$ by using the doubling method with transition kernel T.

Compute acceptance probability: $\alpha_j = \frac{1}{|B_j|}\sum_{\theta,\boldsymbol{\rho}\in B_j} \min\left\{1, \frac{p\left(\boldsymbol{\theta}^{j+1},\boldsymbol{\rho}^{j+1}\right)}{p\left(\boldsymbol{\theta}^{j},\boldsymbol{\rho}^{\text{init}}\right)}\right\}$

Accept the proposal $(\boldsymbol{\theta}^*, \boldsymbol{\rho}^*)$ with probability α_j.

if $j \leq J^{adapt}$ **then**

$\log\left(\varepsilon_{j+1}\right) \leftarrow \mu - \frac{\sqrt{j}}{\gamma}\frac{1}{j+j_0}\sum_{i=1}^{j}\left(\delta - \alpha_i\right)$

$\log\left(\bar{\varepsilon}_{j+1}\right) \leftarrow j^{-\kappa}\log\left(\varepsilon_{j+1}\right) + \left(1 - j^{-\kappa}\right)\log\left(\bar{\varepsilon}_j\right)$

$\varepsilon_{j+1} \leftarrow \bar{\varepsilon}_{j+1}$

else

$\varepsilon_{j+1} = \varepsilon_{J^{\text{adapt}}}$

end

end

4.6 Priors

From section 4.1 and the example given in section 4.3 it is seen that Bayesian inference starts with a prior for the model parameters, which is supposed to embody ones prior beliefs about the assigned task. The prior is an important component and choosing a bad prior can affect the resulting posterior greatly. That said, the prior does have a diminishing effect on the posterior as the number of samples grow, as the likelihood will concentrate the posterior around a few highly likely parameters. One might think that if one has no qualified prior belief, then a weak prior, like a wide uniform distribution, can be a safe bet, but this can however have various unforeseen consequences as pointed out by Lemoine (2019) and Sarma & Kay (2020).

The prior component is however more than a dangerous element that threatens the quality of the posterior. The prior provides a principled mechanism for researchers to incorporate previous research and knowledge into the model. Priors can also be beneficial in small sample sizes as the prior acts to regularize and reduce the chances of overfitting. Although in neural networks the relationship between parameters and the problem can be very abstract and not as intuitive as in other machine learning models like linear regression models or support vector machines, so having qualified prior beliefs about the weights might not be so easy.

Neal (1996) stresses that even though it can seem like BNNs can be threatened by a lack of a suitable prior this is not the case, as much past work shows useful criteria for selecting suitable priors, even without full understanding of what the prior over the parameters will mean in terms of the output of the network. MacKay (1991) and MacKay (1992) has produced results, that Neal (1996) describes as at least reasonable, by giving the parameters Gaussian prior distributions. He lets the standard deviation of these distributions be selected as a hyperparameter, which allows the model to adapt to the data.

According to Neal (1996), our prior knowledge will often be too unspecific to fix the parameter values chosen for the prior distribution, even if we have complete insight into their effects on the prior. We may then wish to treat these values as unknown hyperparameters, giving them a higher-level broad prior distribution, which we call a hyper-prior. Neal (1996)

refers to these as Hierarchical models. One benefit of such models is that the appropriate degree of regularization for the task can be determined automatically from the data, see MacKay (1991) and MacKay (1992).

Chapter 5

Evaluation of Neural Network Models

This chapter examines the applications of non-Bayesian and Bayesian neural networks using two datasets. First a dataset on housing prices in Boston for regression and secondly a dataset on credit card client defaults for classification. We will evaluate the models by comparing and examining their accuracy as well as computational runtime.

All models are implemented in the programming language Python and the code can be found in the appendix. The specification on the computer that has performed the computation of the models are shown in appendix .5 along with the packages used and their versions.

In section 5.1 we examine the median prices on houses in specific areas of Boston based on a number of features shown in table 5.1. We aim to predict unknown median prices based on these features using regression. Section 5.2 examines defaulting payments on credit card clients in Taiwan. Our objective is in this case to predict the probability of defaults and describe how these can be used for classifying whether a client will default or not.

5.1 Predicting House Prices in Boston

The Boston housing data was originally introduced by Harrison & Rubinfeld (1978), who investigated the effect of air pollution on house prices. The dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Massachusetts. The sample contains 506 examples, where each example represents a unique area and has 14 features, which are shown in table 5.1. In this section we examine how to use the theory presented in the previous chapters for using non-Bayesian and Bayesian neural networks to predict the median value of owner-occupied homes in thousands.

The objective is to make good predictions in terms of a low mean squared error (MSE) on unseen data. For this reason we split the data, so that some data are only used as a test set for evaluating the model, while some other data are used as a training set for training the model. We also make another split to generate a validation set, which is used to calculate the loss for each training epoch for determining the stopping time of early stopping, see section 3.3. We perform the same splits for all of the neural networks, so that we ensure that they are trained and tested on the same data, meaning that we also have a validation set for the neural networks not using early stopping. This allows us to use the validation set to illustrate the validation loss and training loss for each epoch to examine possible overfitting in the non-Bayesian neural networks. First we split the data, so that 30% are used for test data and the remaining for training and validation. This remaining data are then split, so that 70 % of the remaining data are used only for training and the rest is used for the validation set. To avoid the possibility that the data are sorted in some undesired way we choose to shuffle the data randomly before splitting it.

A predictive model for house prices can be very useful in many applications. It could help a real estate developer determine the selling price of newly developed houses or it could help a buyer who wants to buy a house in a certain area. It can also be useful for a property investors to determine the price for an area to invest in.

Feature name	Feature description
crim	Per capita crime rate by town
zn	Proportion of residential land zoned for lots over 25,000 sq. ft
indus	Proportion of residential land zoned for lots over 25,000 sq. ft
chas	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
nox	Nitric oxide concentration (parts per 10 million)
rm	Average number of rooms per dwelling
age	Proportion of owner-occupied units built prior to 1940
dis	Weighted distances to five Boston employment centers
rad	Index of accessibility to radial highways
tax	Full-value property tax rate per $10,000$
ptratio	Pupil-teacher ratio by town
b	$1000(Bk - 0.63)^2$, where Bk is the proportion of people of African American descent by town
lstat	Percentage of lower status of the population
medv	Median value of owner-occupied homes in $1000s$

Table 5.1: Table of features in Boston Housing data. The dataset contains 506 examples each with 14 features. Data can be downloaded on: http://lib.stat.cmu.edu/datasets/boston.

5.1.1 Regression with Neural Networks

We perform regression with the neural networks listed in table 5.2. All of these are implemented using MSE as loss function, the ReLU function from equation 3.1.3 as activation function on every hidden layer and no activation function for the output layer. All of the networks have 10 neurons in each hidden layer and are trained with 300 epochs using ADAM described in section 2.6.6. We have selected these settings as they were found to provide acceptable results after experimenting with several other options. We reuse these settings for all the neural networks to make them more suitable for comparison.

The training and validation loss for each training epoch is shown for the networks in figure 5.1, figure 5.2 and figure 5.3. These figures show that the validation loss is almost always lower than our training loss for all of the networks, which might indicate that neither of the models overfit, but might also be due to chance. We examine a neural network with no hidden layers and weight decay, as this is equal to linear regression with weight decay. We include this to test if a neural network is beneficial compared to the simple approach of using linear regression. We see from table 5.2 that we get a smaller amount of test and train error when using one hidden layer even without regularization, which indicates that using a neural network for this problem is indeed useful.

When we use a neural network with one hidden layer and weight decay we get a slightly smaller test loss and slightly larger train loss, than we did when using one hidden layer and no regularization, indicating that we might have prevented some overfitting by introducing weight decay.

We also test regularizing the network of one hidden layer by using the method of early stopping, and see that this gives a slightly smaller test loss and larger train loss than the network regularized with weight decay. So early stopping might have prevented overfitting better than weight decay. Early stopping is, as mentioned in section 3.3, most useful when validation loss begins to increase at some point before ending the training. We see from figure 5.3 that this is not the case, but that the validation loss begins to oscillate slightly more after the stopping epoch. If test loss had the same behavior then this might explain why early stopping attained a smaller test loss than weight decay. We examined training

periods with more epochs, to see if validation loss would begin to increase at some later point, but both validation and training error continued to be around the same level as in epoch 300.

Model	Train MSE	Test MSE	Runtime (s)
No hidden layers & weight decay	81.798	60.074	10.64
1 hidden layer & no regularization	36.539	25.650	10.14
1 hidden layers & weight decay	37.103	25.542	10.81
1 hidden layers & early stopping	37.512	25.223	8.66

Table 5.2: Performance measurement for neural network models on Boston Housing data. Early stopping ran 255 epochs with patience 10 and $\delta_{\text{min}} = 0.1$. For weight decay on all networks we select $\alpha = 0.3$ as the regularization constant. The Python code used to implement these neural networks can be seen in appendix .6

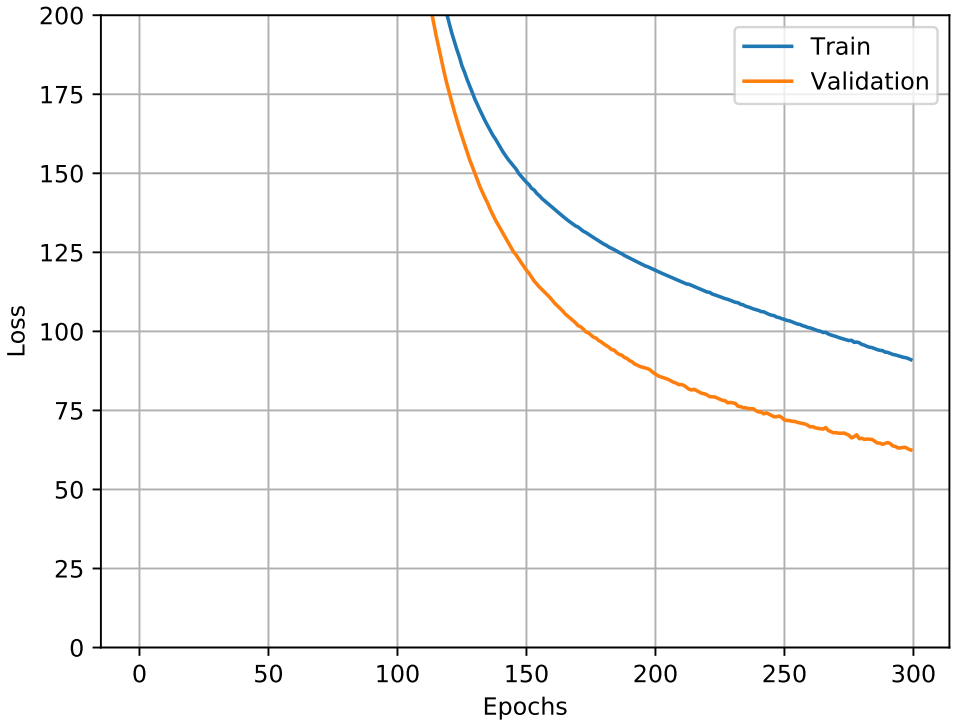


Figure 5.1: Loss on training and validation set during training epochs for the neural network with no hidden layers weight decay.

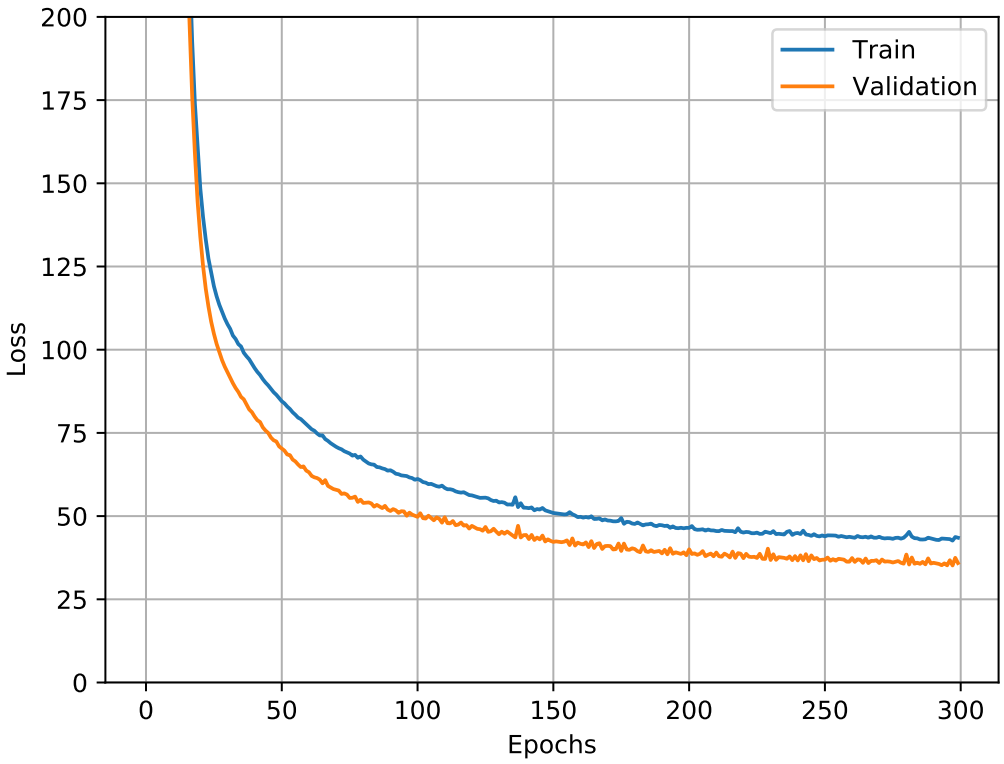


Figure 5.2: Loss on training and validation set during training epochs for the neural network with 1 hidden layers and weight decay.

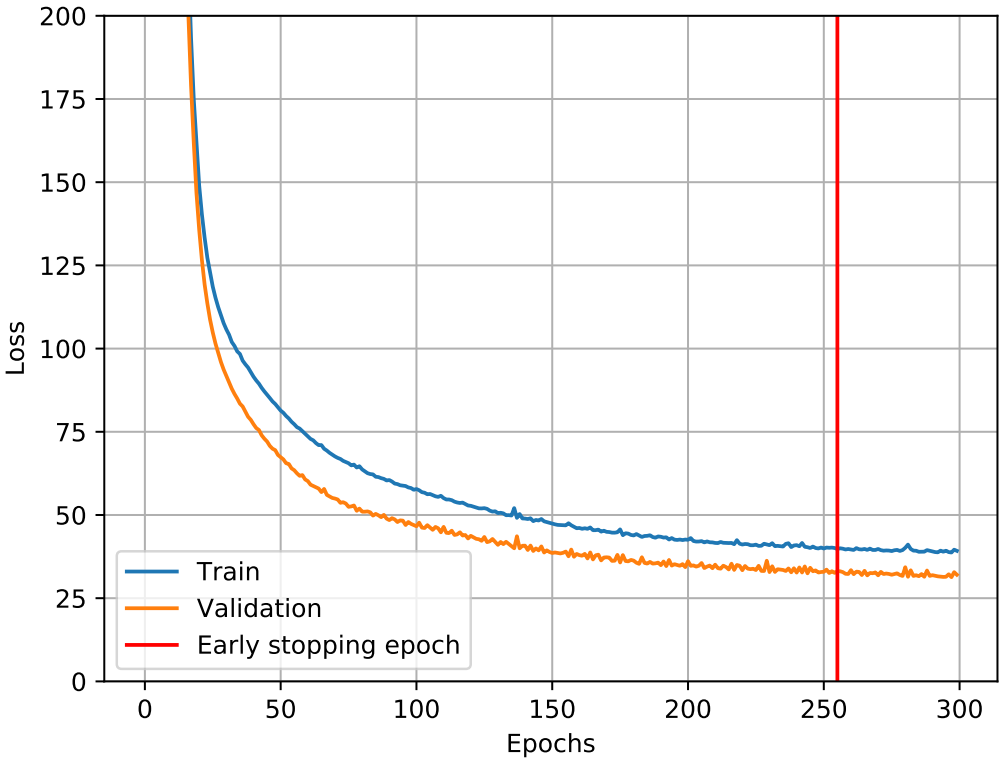


Figure 5.3: Loss on training and validation set during training epochs for the neural network with one hidden layer & no regularization. These losses are the same as for the network with 1 hidden layer and early stopping up to the stopping time on epoch 255 indicated by the red vertical line. It can be seen that validation loss is somewhat flattening and decreasing in oscillations around epoch 255, which might be what made the early stopping algorithm indicate no further improvements in validation loss, given the chosen patience of 10 epochs and $\delta_{\text{min}} = 0.1$, and stop the training.

5.1.2 Regression with Bayesian Neural Networks

We perform regression with the Bayesian neural networks listed in table 5.3. All of these networks are implemented with a Gaussian likelihood with mean equal to the network output and standard deviation σ equal to 1. We have chosen to run three kinds of BNNs. First a BNN with no hidden layers which corresponds to Bayesian linear regression. Then a BNN with one hidden layer, where we follow the example of MacKay (1991) and MacKay (1992) and choose a Gaussian prior on the parameter weights, which we choose more specifically to be $\mathcal{N}(0, 0.1)$. Lastly a hierarchical BNN with one hidden layer, with a $\mathcal{N}(\mu, \sigma)$ prior for the network weights. The parameter μ is chosen to follow a Cauchy(α, β) distribution and the standard deviation σ follows a Half-Normal(ν) distribution in order to avoid negative values. The hyper-prior distributions are parametric and therefore require that we specify the hyperparameters for these. After experimenting with different choices of parameters for the hyper-prior, we choose $\alpha = 0$, $\beta = 1$ and $\nu = 1$. All BNN models have 10 hidden neurons and use ReLU in the hidden layer for ease of comparison, and in order to evaluate on the same models as in section 5.1.1 but with a Bayesian approach.

We sample from the posterior distribution of the model parameters using the NUTS algorithm examined in section 4.5.5. We sample from three Markov Chains in parallel to get more independent samples, as described in section 4.4, where we for each chain draw 3000 samples from the posterior distribution. A burn-in phase of 1000 has been chosen for each chain. For the dual averaging algorithm we chose an acceptance probability δ of 90% , as it provided us with most acceptable results. All other inputs used for the NUTS algorithm were not chosen by us, as we used the default settings in `PyMC3`. Predictions are made by sampling from the posterior predictive distribution. We generate a total of 3000×3 samples from the posterior predictive distribution, and to make a single best guess in terms of minimizing MSE we make a prediction by taking the mean of the posterior predictive distribution.

From table 5.3 we see that both BNNs with one hidden layer outperform the baseline model with no hidden layers. As a BNN with no hidden layers corresponds to performing Bayesian linear regression, this indicates that using Bayesian inference in a neural network is more useful than using Bayesian inference in a simple linear regression, in this particular

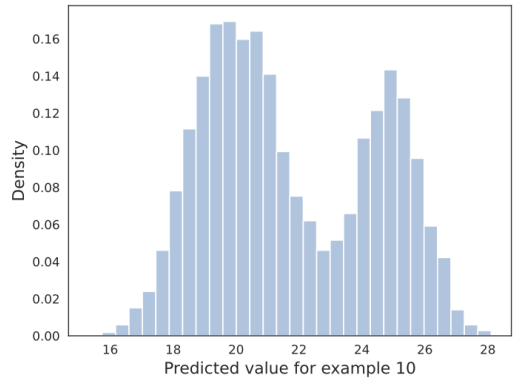
case. We also note that all the BNN models are outperforming the non-Bayesian neural networks in terms of MSE both on the training data, but also on the unseen test data. However, it takes about 140 times longer to run a BNN with one hidden layer than a similar non-Bayesian neural network. So even though the BNNs get a loss-wise better result in this case, it might not be so if one has a time budget lower than the runtime found in table 5.3.

However, the computational burden that comes with BNNs also comes with the benefit of producing a distribution for the predictions. This enables us to consider more possible outcomes when making predictions and to have a better idea of the uncertainty in our predictions. The posterior predictive distribution is shown as a histogram in figure 5.4 for different examples in the test dataset. Such a distribution can be useful in decision making. Consider the example of a property investor, who would like to invest in a certain area. If two areas, according to our model, have the same mean median price he might wish to invest in the area that has the most narrow posterior predictive distribution i.e. lowest standard deviation, so that he is least likely to be surprised by the actual median price.

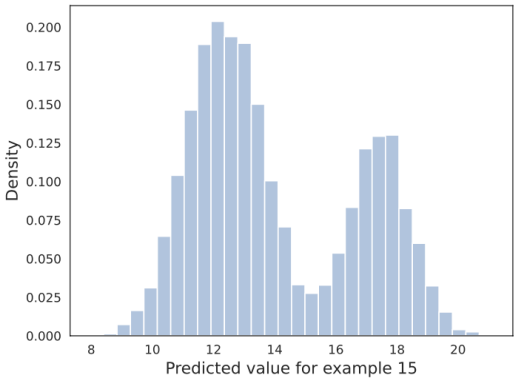
We have chosen to use the mean of the posterior predictive distribution for predictions, as this is usually the best measure for central tendency when the distribution is symmetric, but from figure 5.4 we can see that we might want to take some caution against doing this, as the posterior predictive distribution of the chosen examples show bimodal tendencies and are not symmetric. This shows how to critically evaluate our model when we have access to the distribution of a models predictions. With non-Bayesian neural networks, we only get a single point estimate for the model parameters, and do not factor in the full range of possibilities. Despite these examples we still use the mean for prediction as it provides us with acceptable results.

Model	Train MSE	Test MSE	Runtime (s)
No hidden layers	28.480	17.760	211.63
1 hidden layer	11.391	9.340	1496.69
Hierarchical with 1 hidden layer	14.135	10.869	1851.46

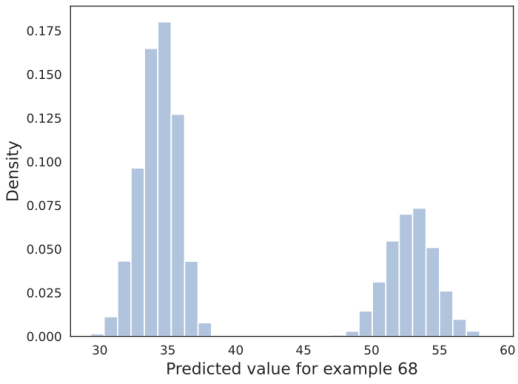
Table 5.3: Performance measurement for Bayesian neural network models on Boston housing data. The Python code used to implement these Bayesian neural networks can be seen in appendix .7.



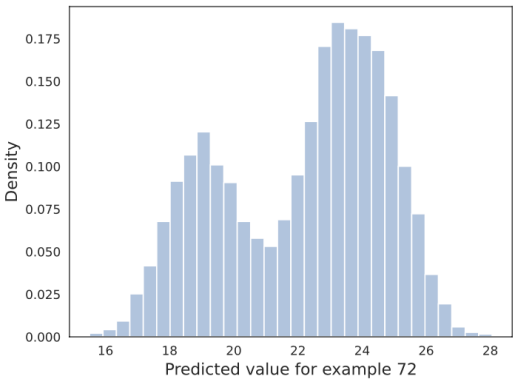
(a)



(b)



(c)



(d)

Figure 5.4: A histogram of predictions generated by samples from from the posterior predictive distribution for example 10, 15, 68 and 72 in the test dataset. The predictions are based on the 1 hidden layer BNN model

5.2 Predicting Defaults of Credit Card Clients

The default of credit card clients dataset contains information collected by a Taiwan bank in 2005, on 30.000 credit card clients. The data contains 25 variables, which are represented in table 5.4. Due to the computational complexity of the BNN model, we choose to subsample our dataset, such that it only contains 300 train examples and 100 test examples. We also perform the same splits of the data as in section 5.1, meaning that 30 % of the data are contained in a test set, and the remaining data are afterwards split into 70 % training data and 30 % validation data. The data are shuffled before splitting, to prevent that it should be sorted in an undesired way.

The default of credit card clients dataset has earlier been used by Yeh & hui Lien (2009) for predicting the probability of default for customers in Taiwan. The objective for our analysis is the same, to predict the probability of default on next months payment. This probability can then be used to predict defaults and expected losses. Since we are predicting a probability, all models will be using the cross-entropy loss from equation 2.2.2. A predictive model, that can predict the default probability of the next credit card payment, could be very useful for a banks risk management unit in order to detect which customers should be allowed to own a credit card and which should be denied.

The data does not contain an equal fraction of the two classes, since examples with the true label of default only constitutes 22.12%. Such a problem is often referred to as a class imbalance problem in binary classification and has been examined a lot through the years. Danquah (2020) suggest to use methods such as oversampling and undersampling for imbalanced class problems, but this will not be performed as we simply wish to evaluate the networks performance on classification. We simply note that a model can easily get an accuracy score of around 80% by just predicting the majority class for all examples without necessarily learning any patterns.

Feature name	Feature description
LIMIT_BAL	Amount of the given credit (New Taiwan dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.
SEX	Gender (1 = male; 2 = female)
EDUCATION	Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).
MARRIAGE	Marital status (1 = married; 2 = single; 3 = others)
AGE	Age (year)
PAY_0	Repayment status in September, 2005 -2=no consumption, -1=pay duly 0=the use of revolving credit, 1=payment delay for one month 2=payment delay for two months, 3=payment delay for three months ⋮ 8=payment delay for eight months, 9=payment delay for nine months and above
PAY_2	Repayment status in August, 2005 (scale same as above)
PAY_3	Repayment status in July, 2005 (scale same as above)
PAY_4	Repayment status in June, 2005 (scale same as above)
PAY_5	Repayment status in May, 2005 (scale same as above)
PAY_6	Repayment status in April, 2005 (scale same as above)
BILL_AMT1	Amount of bill statement in September, 2005 (NT dollar)
BILL_AMT2	Amount of bill statement in August, 2005 (NT dollar)
BILL_AMT3	Amount of bill statement in July, 2005 (NT dollar)
BILL_AMT4	Amount of bill statement in June, 2005 (NT dollar)
BILL_AMT5	Amount of bill statement in May, 2005 (NT dollar)
BILL_AMT6	Amount of bill statement in April, 2005 (NT dollar)
PAY_AMT1	Amount of previous payment in September, 2005 (NT dollar)
PAY_AMT2	Amount of previous payment in August, 2005 (NT dollar)
PAY_AMT3	Amount of previous payment in July, 2005 (NT dollar)
PAY_AMT4	Amount of previous payment in June, 2005 (NT dollar)
PAY_AMT5	Amount of previous payment in May, 2005 (NT dollar)
PAY_AMT6	Amount of previous payment in April, 2005 (NT dollar)
default payment next month	Default payment (1=yes, 0=no)

Table 5.4: Table of features in credit card default data. Data contains 30.000 examples each with 25 features. Data can be downloaded on:

https://archive.ics.uci.edu/ml/datasets.php

Model	Train cross-entropy loss	Test cross-entropy loss	Runtime (s)
No hidden layers & weight decay	47.841	67.646	38.08
1 hidden layer & no regularization	0.498	0.565	38.80
1 hidden layers & weight decay	0.513	0.558	28.44
1 hidden layers & early stopping	0.503	0.564	6.15

Table 5.5: Performance measurement for neural network models on Boston Housing data. Early stopping ran 194 epochs. The Python code used to implement these neural networks can be seen in appendix .8.

5.2.1 Classification with Neural Networks

We perform predictions on default probabilities of credit card clients using the networks in table 5.5. All of these networks use the tanh activation function from equation 3.1.2 in every hidden layer and a sigmoid activation from equation 3.1.1 for the output layer. All networks have 10 neurons in each hidden layer and are trained with 1000 epochs using ADAM from section 2.6.6. These settings were selected after experimenting with different other options and the settings are the same for all the neural networks so that we can compare them on their individual differences. The training and validation loss for each training epoch of the networks can be seen in figure 5.5, figure 5.6 and figure 5.7.

We examine a network with no hidden layers and weight decay as this correspond to logistic regression with weight decay to see if a complex model like neural networks are beneficial for this task instead of using simple logistic regression. We see from table 5.5 that no hidden layers i.e. logistic regression has a much larger train and test loss indicating that neural networks are indeed more useful for this task.

The neural network with one hidden layer and no regularization gets a smaller train loss but larger test loss than the other one hidden layer networks. This indicates that this network is overfitting, which is also apparent in figure 5.7. The NN with one hidden layer and weight decay is regularized and provides a larger training loss but smaller test loss indicating that weight decay has mitigated the overfitting. The NN with one hidden layer and early stopping has a slightly larger test loss than the NN with one hidden layer and weight decay, but it shows a small improvement in test loss compared to the network with no regularization

and a substantially shorter runtime. This is caused by sudden continually increase in the validation error as shown in figure 5.7. We see here that early stopping stops training at the exact time, that the validation error starts increasing, and this happens as we chose patience and δ_{min} to both be zero. These values were chosen by seeing that the validation error was smoothly increasing when training the network with no regularization.

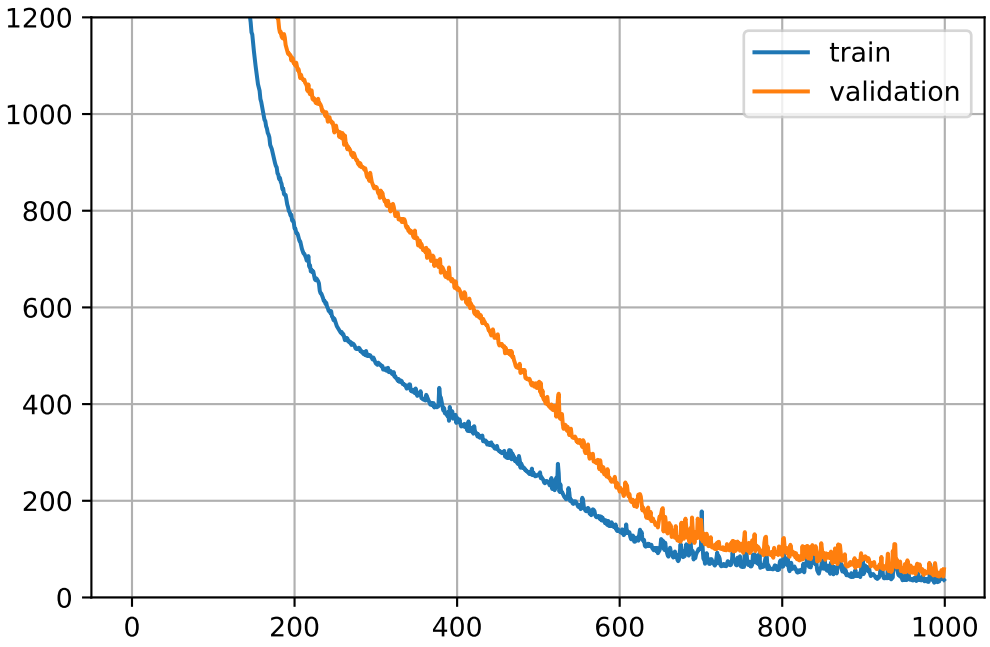


Figure 5.5: Loss on training and validation set during training epochs for the neural network with no hidden layers & weight decay.

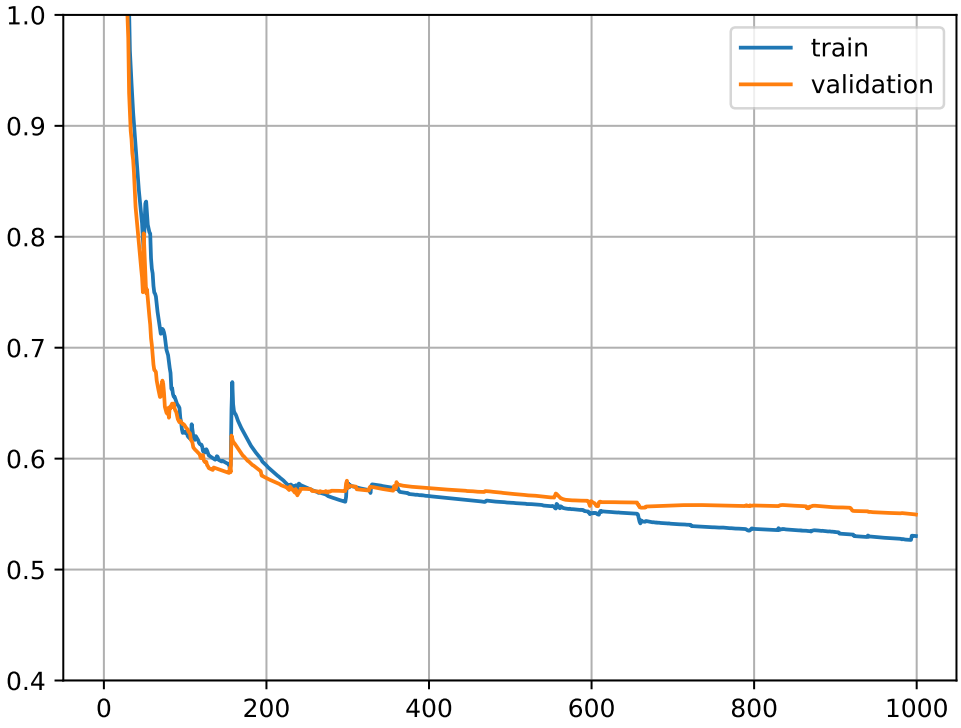


Figure 5.6: Loss on training and validation set during training epochs for the neural network with 1 hidden layers & weight decay.

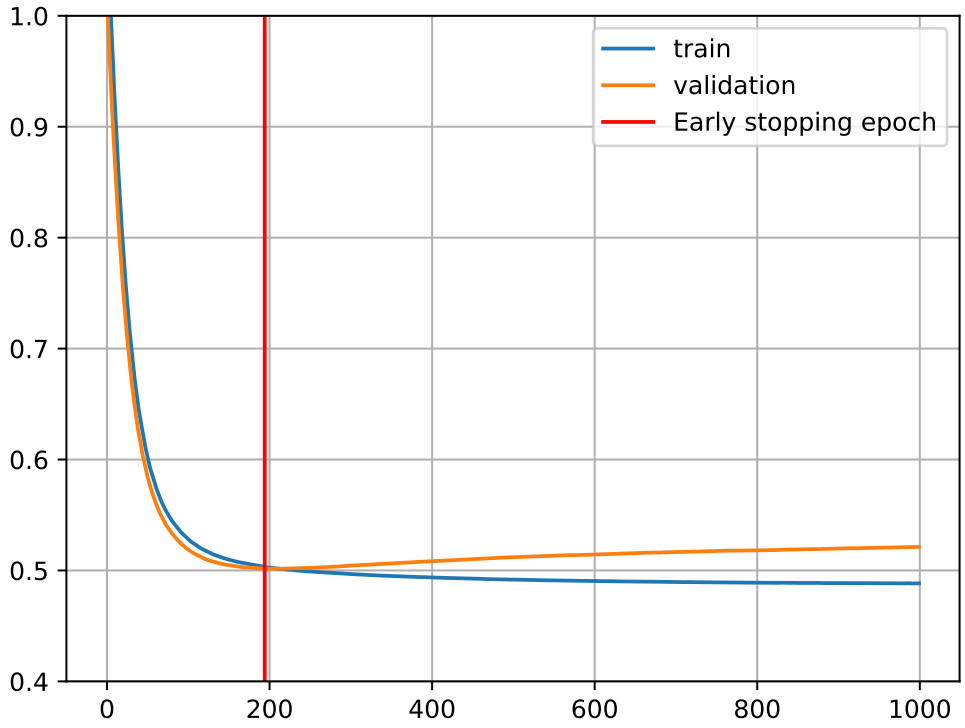


Figure 5.7: Loss on training and validation set during training epochs for the neural network with 1 hidden layers & no regularization. These losses are the same as for the network with 1 hidden layer and early stopping up to the stopping time on epoch 194, indicated by the red vertical line. It can be seen that validation loss is smoothly increasing, which is what made the early stopping algorithm indicate no further improvements in validation loss and stop the training, given the chosen patience of 0 epochs and $\delta_{\text{min}} = 0$.

5.2.2 Classification with Bayesian Neural Networks

In this section we predict default probabilities with the Bayesian neural networks listed in table 5.6. All these models are implemented with tanh activation function for the hidden layers and sigmoid for the output layer, as in the previous section with non-Bayesian neural network. All three models are implemented with a Bernoulli (p) likelihood, where the bias p is equal to the outputted probability of the network. The BNN with no hidden layers is used for comparing performance of the models to a simple Bayesian logistic regression. The BNN with one hidden layer has a $\mathcal{N}(0,1)$ prior distribution placed on the network parameters. The hierarchical BNN with one hidden layer, has a $\mathcal{N}(\mu,\sigma)$ prior for the network weights, where μ and σ are distributed by a Cauchy and a half-normal distribution respectively, as in the regression evaluation.

Sampling from the posterior distribution is performed using NUTS with three Markov chains, that runs in parallel, to avoid dependence between samples. For each chain we choose to draw 1500 samples, as running more than 1500 samples per chain resulted in the system exceeded its memory limit. Like before, we use a burn-in phase of 1000 samples for each chain, since we believe the Markov chains must have converged to the posterior by then. For the NUTS-algorithm we choose an acceptance probability δ of 90% and for all other inputs we use the default setting of `PyMC3`. Predictions are performed by sampling 1500×3 from the posterior predictive distribution and then take the mean across samples to make a single best guess for the default probability.

In table 5.6 we see that both the BNN with one hidden layer and the BNN as a hierarchical model are performing better than the BNN with no hidden layers. This indicates that it is useful to use a neural network for predicting credit default probabilities compared to using Bayesian logistic regression. We see that the runtimes of the BNNs are much higher than the non-Bayesian models, and since we get the same results in terms of cross-entropy loss, it is not beneficial to use BNNs instead of NNs if one is only interested in getting the lowest loss pr. runtime.

The benefits of the BNNs are instead found in the produced distribution of the predictive posterior. From a risk management perspective, this could be interesting to get an idea

Model	Train cross-entropy loss	Test cross-entropy loss	Runtime (s)
No hidden layers	0.774	0.685	20.332
1 hidden layer	0.523	0.539	1233.58
Hierarchical with 1 hidden layer	0.523	0.539	3640.28

Table 5.6: Performance measurement for Bayesian neural network models on credit card default data. The Python code used to implement these Bayesian neural networks can be seen in appendix .9.

of the uncertainty of the models prediction of default probability. The mean probability of defaulting might be 45% for two different clients, but the uncertainty in this quantity might be significantly different. This uncertainty can be examined by taking a look at the posterior predictive distribution. In figure 5.8 we have illustrated the posterior predictive distribution generated by the one hidden layer BNN for example 5, 11, 25 and 88 in the test dataset. We see that sometimes the model is quite confident about its predictions as in subfigure (c) and (d), but for other examples the uncertainty, expressed by the standard deviation of the distribution, is quite high like in subfigure (a) and (b). A bank could use this to decide not to give credit to a client if the uncertainty of the default probability is too high, like the client corresponding to subfigure (a) of figure 5.8.

In practice one might use the predicted default probabilities, that we use for our evaluation, to predict if the client will default or not. One can do this by classifying the most probable class, which means predicting default if the probability is more than 50%. This might however not be the most reasonable approach, if one prefers to wrongly classify one class more than the other. We might for example imagine that providing credit to a client that defaults next month would be much worse than denying a client that does not default next month. In this case it can make sense to choose another classification boundary than 50%.

This classification boundary can be determined by the loss of wrongly predicting certain classes and can be set so that the expected loss do not exceed a certain value. Choosing a proper classification boundary for this case will however not be pursued further, as we performed our evaluation using the default probabilities instead of the labels, which would

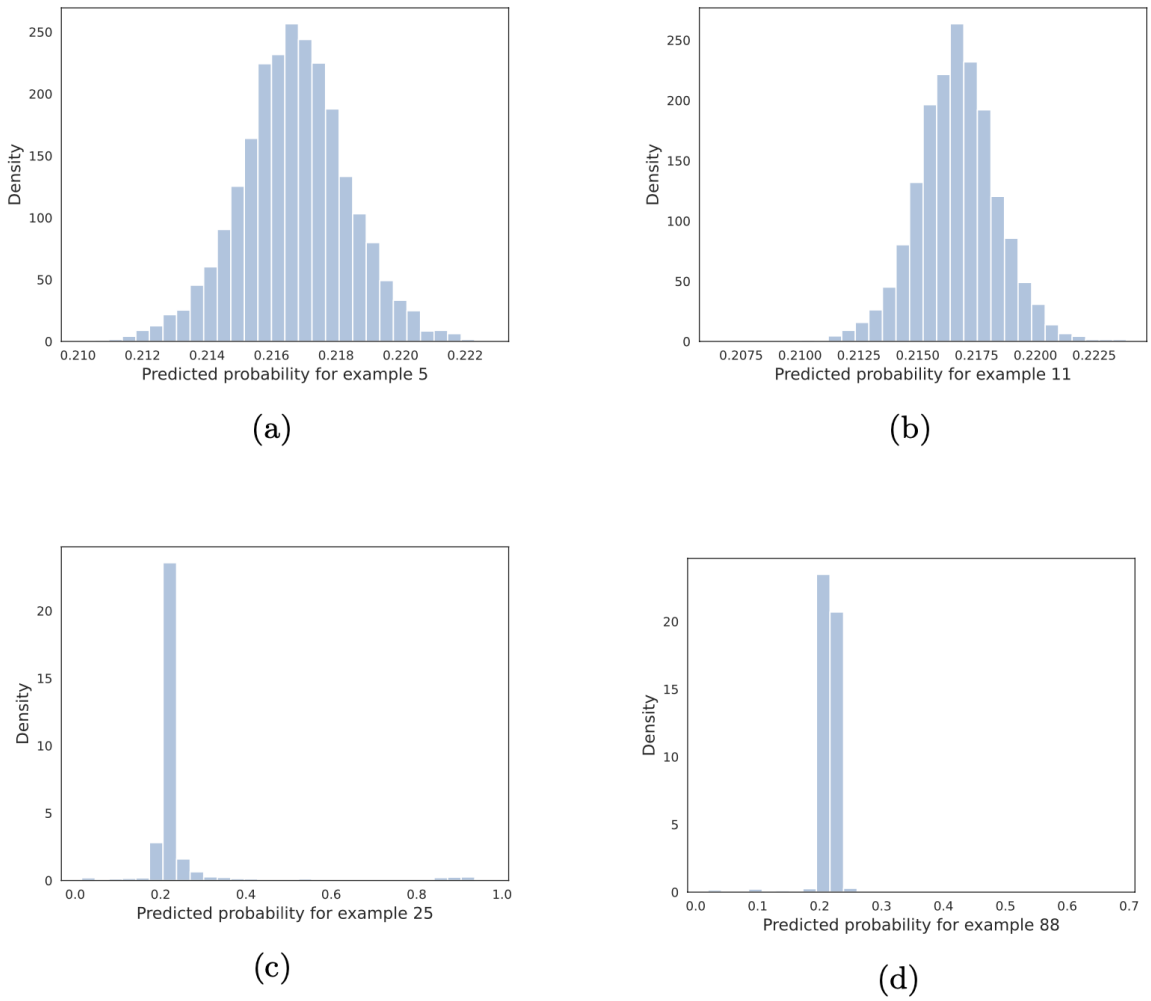


Figure 5.8: A histogram of predictions generated by sampling from the posterior predictive distribution for example 5, 11, 25 and 88 in the test dataset. The predictions are based on the 1 hidden layer BNN model.

depend on the choice of such classification boundary.

Chapter 6

Conclusion

In this thesis we have examined how Bayesian inference can be used in neural networks using Markov chain Monte Carlo sampling and how these and non-Bayesian neural networks perform regression and classification. The most essential difference between these two approaches of using neural networks is that Bayesian neural networks require a prior distribution for its weights and use sampling procedures to generate a distribution for its weights and predictions, while non-Bayesian neural networks use optimization algorithms, such as the gradient-based optimization algorithms described in section 2.6, to learn a single estimate of the weights that result in predictions that provide the least amount of loss.

The focal point of the examination of the Bayesian neural networks has been how to efficiently sample from the posterior distribution for the weights using Markov chain Monte Carlo methods. We analyzed this by first providing a simple example of a Bayesian neural network that took weights sampled from a prior and accepted them with a probability proportional to the likelihood of the produced results, a sort of rejections sampling illustrated by Neal (1996). This provided the motivation for examining more efficient sampling methods such as the Markov chain Monte Carlo methods. One such method was the Metropolis algorithm that has the inefficiency of exploring the distribution with a random walk behavior. For this reason we examined the Hamiltonian Monte Carlo, which combine the principles of the Metropolis algorithm with Hamiltonian dynamics for a better exploration of the distribution. This turns out to have the inefficiency of sometimes performing "U-turns", meaning that it has the possibility of returning to the previous sample-point of

the Markov chain and thus make highly correlated samples. The No-U-Turn Hamiltonian Monte Carlo is an extension to Hamiltonian Monte Carlo, which prevents such a U-turn and also make adaptive choices to the size and the number of steps taken by the Markov chain before sampling. We conclude our examination of Bayesian neural networks by covering the effect of the prior choice and by suggesting general schemes for choosing a prior distribution.

We end our thesis with an evaluation and illustration on how Bayesian and non-Bayesian neural networks work in practice. We do this with varying design choices to show how elements such as a hierarchical prior and different choice of regularization affects results. These different neural networks are applied to a regression task with the aim of predicting house prices and a classification task with the aim of predicting default probabilities of credit card clients. We furthermore show some of the benefits from the produced distributions of the predictive posterior from the Bayesian neural networks, and how to use these to evaluate models and make better predictions.

Chapter 7

Future work

In this thesis we examined Bayesian neural network exclusively using Markov chain Monte Carlo methods. These methods are often computationally complex, especially in neural network where the posterior is high-dimensional due to number of neural network weights. Therefore it could have been interesting to look at methods that could be more easily scaled to large distributions. Methods such as variational Inference (Blei et al. (2018)) has gained a lot of popularity due to its scalability to more complex models. Unlike Markov chain Monte Carlo methods, variational inference is not an exact method. Instead of sampling directly from the posterior the idea is to have a parametric distribution $q(\boldsymbol{\phi})$, called the variational distribution, to sample from instead. The parameters of the variational distribution are optimized in such a way that the variational distribution is as close as possible to the true posterior distribution in terms of a measure called the evidence lower bound. In this context it could have been interesting to look into the work by Kucukelbir et al. (2016), who has built an automatic differentiation variational inference algorithm, that can automatically optimize the parameters for the variational distribution.

Another interesting way to design Bayesian neural networks is proposed by Blundell et al. (2015) and is called Bayes-by-backprop, which is a variational inference methods combined with reparamitzation trick to ensure that backpropagation works as we described in section 4, but over the variational distribution parameters $\boldsymbol{\phi}$, which makes it possible to learn the variational distribution by using optimization algorithms like the ones described in section 2.6.

Another popular option worth examining is modelling uncertainty in neural networks by using dropout (Srivastava et al. (2014)) to approximate the variational distribution. Gal & Ghahramani (2016) do this by using a type of ensemble learning, where they generate random predictions for test examples by the dropout method and interpret these as coming from a distribution. They call this method Monte Carlo dropout and states, that it produces faster results than both MCMC methods and variational inference.

Bibliography

Abu-Mostafa, Y., Magdon-Ismail, M. & Lin, H. (2012), *Learning from Data: A Short Course*, AMLBook.com.

Bengio, Y. & Grandvalet, Y. (2004), 'No unbiased estimator of the variance of k-fold cross-validation.', *J. Mach. Learn. Res.* **5**, 1089–1105.
URL: *http://jmlr.org/papers/volume5/grandvalet04a/grandvalet04a.pdf*

Betancourt, M. (2017), 'A conceptual introduction to hamiltonian monte carlo'. arxiv:1701.02434.
URL: *http://arxiv.org/abs/1701.02434*

Bishop, C. (1995), 'Regularization and complexity control in feed-forward networks', **1**, 141–148.

Bishop, C. M. (1997), 'Bayesian Neural Networks', *Journal of the Brazilian Computer Society* **4**.
URL: *https://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65001997000200006*

Bishop, C. M. (2007), *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1 edn, Springer.

Blei, D. M., Kucukelbir, A. & McAuliffe, J. D. (2018), 'Variational inference: A review for statisticians'.
URL: *https://arxiv.org/abs/1601.00670*

Blundell, C., Cornebise, J., Kavukcuoglu, K. & Wierstra, D. (2015), 'Weight uncertainty in neural networks'. cite arxiv:1505.05424Comment: In Proceedings of the 32nd Interna-

tional Conference on Machine Learning (ICML 2015).
URL: http://arxiv.org/abs/1505.05424

Bottou, L. & Bousquet, O. (2008), The tradeoffs of large scale learning, *in* J. Platt, D. Koller, Y. Singer & S. Roweis, eds, 'Advances in Neural Information Processing Systems', Vol. 20, NIPS Foundation (http://books.nips.cc), pp. 161–168.
URL: http://leon.bottou.org/papers/bottou-bousquet-2008

Breiman, L. & Spector, P. (1992), 'Submodel selection and evaluation in regression. the x-random case', *International Statistical Review / Revue Internationale de Statistique* **60**(3), 291–319.
URL: http://www.jstor.org/stable/1403680

Buntine, W. L. & Weigend, A. (1991), 'Bayesian back-propagation', *Complex Syst.* **5**.

Cauchy, M. A. (1847), 'Méthode générale pour la résolution des systèmes d'équations simultanées', *Comptes Rendus Hebd. Seances Acad. Sci.* **383**(25), 536—-538.

Charnock, T., Perreault-Levasseur, L. & Lanusse, F. (2020), 'Bayesian neural networks'.
URL: https://arxiv.org/abs/2006.01490

Cybenko, G. (1989), 'Approximation by superpositions of a sigmoidal function.', *Math. Control. Signals Syst.* **2**(4), 303–314.

Danquah, R. A. (2020), 'Handling imbalanced data: A case study for binary class problems'.

Duane, S., Kennedy, A. D., Pendleton, B. J. & Roweth, D. (1987), 'Hybrid monte carlo', *Physics Letters B* **195**(2), 216 – 222.
URL: https://www.sciencedirect.com/science/article/abs/pii/037026938791197X

Duchi, J., Hazan, E. & Singer, Y. (2011), 'Adaptive subgradient methods for online learning and stochastic optimization', *Journal of Machine Learning Research* **12**(Jul), 2121–2159.

Feng, C. (2016), 'The markov-chain monte carlo interactive gallery'.
The MIT License (MIT). Copyright ©2016 Chi Feng: *Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish,*

distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. in no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

URL: https://chi-feng.github.io/mcmc-demo/

Funahashi, K.-I. (1989), 'On the approximate realization of continuous mappings by neural networks', *Neural Networks* **2**(3), 183–192.
URL: https://www.sciencedirect.com/science/article/pii/0893608089900038

Gal, Y. & Ghahramani, Z. (2016), 'Dropout as a bayesian approximation: Representing model uncertainty in deep learning'.
URL: https://arxiv.org/abs/1506.02142

Gamerman, D. & Lopes, H. (2006), *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition*, Chapman & Hall/CRC Texts in Statistical Science, Taylor & Francis.

Gelman, A., Carlin, J. B., Stern, H. S. & Rubin, D. B. (2004), *Bayesian Data Analysis*, 2nd ed. edn, Chapman and Hall/CRC.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press.
URL: http://www.deeplearningbook.org

Harrison, D. & Rubinfeld, D. L. (1978), 'Hedonic housing prices and the demand for clean air', *Journal of Environmental Economics and Management* **5**(1), 81–102.
URL: https://www.sciencedirect.com/science/article/pii/0095069678900062

Hastie, T., Tibshirani, R. & Friedman, J. (2001), *The Elements of Statistical Learning*, Springer Series in Statistics, Springer New York Inc., New York, NY, USA.

Hastings, W. K. (1970), 'Monte carlo sampling methods using markov chains and their applications', *Biometrika* **57**(1), 97–109.
URL: *http://biomet.oxfordjournals.org/cgi/content/abstract/57/1/97*

Hoffman, M. D. & Gelman, A. (2011), 'The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo'.
URL: *https://arxiv.org/abs/1111.4246*

Kingma, D. P. & Ba, J. (2015), 'Adam: A method for stochastic optimization.', *International Conference on Learning Representations* **1–13**.
URL: *https://arxiv.org/pdf/1412.6980.pdf*

Kohavi, R. (2001), 'A study of cross-validation and bootstrap for accuracy estimation and model selection', **14**.

Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A. & Blei, D. M. (2016), 'Automatic differentiation variational inference'.
URL: *https://arxiv.org/abs/1603.00788*

Lawler, G. (2006), *Introduction to Stochastic Processes, Second Edition*, Chapman & Hall/CRC Probability Series, Taylor & Francis.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. & Jackel, L. D. (1989), 'Backpropagation applied to handwritten zip code recognition', *Neural Computation* **1**, 541–551.

Lemoine, N. P. (2019), 'Moving beyond noninformative priors: why and how to choose weakly informative priors in bayesian analyses', *Oikos* **128**(7), 912–928.
URL: *https://onlinelibrary.wiley.com/doi/pdf/10.1111/oik.05985*

MacKay, D. J. (1991), 'Bayesian methods for adaptive models'.

MacKay, D. J. (1992), 'A practical bayesian framework for backpropagation networks'.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. & Teller, E. (1953), 'Equation of state calculations by fast computing machines', *The Journal of Chemical Physics* **21**(6), 1087–1092.

Neal, R. M. (1996), *Bayesian learning for neural networks*, Vol. 118, Springer Science & Business Media.

Neal, R. M. (2003), 'Slice sampling', *The Annals of Statistics* **31**(3), 705–767.

Neal, R. M. (2012), 'Mcmc using hamiltonian dynamics'.
URL: *http://arxiv.org/abs/1206.1901*

Nishio, M. & Arakawa, A. (2019), 'Performance of Hamiltonian Monte Carlo and No-U-Turn Sampler for estimating genetic parameters and breeding values', *Genetics Selection Evolution* **51**(1), 73.
URL: *https://hal.archives-ouvertes.fr/hal-02405175*

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning Representations by Back-propagating Errors', *Nature* **323**(6088), 533–536.

Sarma, A. & Kay, M. (2020), Prior setting in practice: Strategies and rationales used in choosing prior distributions for bayesian analysis, *in* 'Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems', CHI '20, Association for Computing Machinery, New York, NY, USA, p. 1–12.
URL: *https://doi.org/10.1145/3313831.3376377*

Sjöberg, J. & Ljung, L. (1992), 'Overtraining, regularization, and searching for minimum in neural networks', *IFAC Proceedings Volumes* **25**(14), 73–78. 4th IFAC Symposium on Adaptive Systems in Control and Signal Processing 1992, Grenoble, France, 1-3 July.
URL: *https://www.sciencedirect.com/science/article/pii/S1474667017507156*

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), 'Dropout: a simple way to prevent neural networks from overfitting.', *Journal of Machine Learning Research* **15**(1), 1929–1958.

Stigler, S. (1986), *The History of Statistics: The Measurement of Uncertainty Before 1900*, Belknap Series, Harvard University Press.

Sutton, R. S. (1986), Two problems with backpropagation and other steepest-descent learning procedures for networks, *in* 'Proceedings of the Eighth Annual Conference of the Cognitive Science Society', Hillsdale, NJ: Erlbaum.

Tieleman, T. & Hinton, G. (2012), 'Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude'.

Turkman, M., Paulino, C. & Müller, P. (2019), *Computational Bayesian Statistics: An Introduction*, Institute of Mathematical Statistics Textbooks, Cambridge University Press.

Wackerly, D. D., III, W. M. & Scheaffer, R. L. (2002), *Mathematical Statistics with Applications*, sixth edition edn, Duxbury Advanced Series.

Yeh, I. & hui Lien, C. (2009), 'The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients', *Expert Syst. Appl.* **36**, 2473–2480.

Zhou, Z.-H. (2012), *Ensemble Methods: Foundations and Algorithms*, Vol. 14, CRC Press.

Appendices

.1 Python code used for producing the over-underfitting example in figure 2.1

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score


def true_fun(X):
    return np.cos(1.5 * np.pi * X)


np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                             include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_features
   ),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)
```

```
    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                             scoring="neg_mean_squared_error", cv=
   10)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label=
   "Model")
    plt.plot(X_test, true_fun(X_test),
             label="True function", linestyle='dashed')
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE_cv = {:.2e}(+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))
plt.show()
```

.2 Python code for producing the activation functions in figure 3.2

```
import tensorflow as tf
import matplotlib.pyplot as plt

x = tf.linspace(-10., 10., 200)
y_sigmoid = tf.keras.activations.sigmoid(x)
y_tanh = tf.keras.activations.tanh(x)
y_relu = tf.keras.activations.relu(x)
y_elu = tf.keras.activations.elu(x)
y_step = x > 0
```

```
plt.plot(x, y_sigmoid, label='Sigmoid')
plt.plot(x, y_tanh, label=r'$tanh$')
plt.plot(x, y_relu, label='ReLU')
plt.plot(x, y_elu, label='ELU')
plt.plot(x, y_step, label='Step')
axes = plt.gca()
axes.set_ylim([-2, 2])
plt.xlabel('x')
plt.ylabel('g(x)')
plt.legend()
plt.savefig('act_func_fig.pdf')
plt.show()
```

.3 Python code for implementing the simple Bayesian neural network illustrated in figure 4.1

```
#import random as rn
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

# ------------------------------- Creating sin-data ---------------
    ----------------


def true_fun(x):
    return np.sin(3 * x)  # np.sin(1.5 * np.pi * x)


np.random.seed(42)
```

```
n_x = 6
x_train = np.sort(np.random.rand(n_x))
y_train = true_fun(x_train)


# ---------------------- Build and compile neural net ---------------
    ----------------

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, input_shape=([1, ]), activation='tanh'
    ),
    tf.keras.layers.Dense(1, activation='tanh')
])
model.summary()


# ---------------------- Sample weights through neural networks with
    acceptance-prop equal to likelihood ----------------------

n_NN = 10**5
weight_list = []
likelihood_list = []
sigma_k = 0.1  # sd of assumed gauss P(y \mid x), Neal's eq 1.8,
    Neal uses 0.1
tf.random.set_seed(42)
# Sample neural networks and save their likelihood and weights
for i in range(n_NN):
    print(i)
    # sample weights and set them to hidden layer "dense" and output
    layer "dense_1"
    for layer in model.layers:
        if layer.name == "dense":
            layer.set_weights([np.random.normal(0, 8, size=w.shape)
                               for w in layer.get_weights()])
        if layer.name == "dense_1":
```

```
            layer.set_weights([np.random.normal(
                0, 1/np.sqrt(16), size=w.shape) for w in layer.
    get_weights()])
     # save weights in list
     weight_list.append(model.get_weights())
     # Calculate gauss likelihood of y_train given weights and
    x_train, Neal's eq 1.11
     mean = model.predict(x_train)
     gauss = tfd.Normal(loc=mean, scale=sigma_k)
     likelihood = 1
     for j in range(n_x):
         likelihood *= gauss[j].prob(y_train[j])
     likelihood_list.append(likelihood)

# Normalize likelihood to max prob is 1
if max(likelihood_list) != 0:
     likelihood_list = likelihood_list/max(likelihood_list)

# Accept model with prob equal to normalized likelihood
accepted_weights = []
accepted_likelihood = []
for i in range(len(likelihood_list)):
     uniform_dist = tfd.Uniform(0, 1)
     if likelihood_list[i] >= uniform_dist.sample():
         accepted_weights.append(weight_list[i])
         accepted_likelihood.append(likelihood_list[i])


# ---------------------- Use sampled weights for predicting y's -----
    -----------------
x_pred = tf.linspace(0.0, 1, 200)
y_pred = []
for i in range(len(accepted_weights)):
     model.set_weights(accepted_weights[i])
     y_pred.append(model.predict(x_pred))
```

```
# mean y_pred
mean_y_pred = np.array(y_pred).mean(axis=0)

y_pred_std = np.array(y_pred).std(axis = 0)
# Lower std-line
lower_std = mean_y_pred - y_pred_std
# Upper std-line
upper_std = mean_y_pred + y_pred_std

# ---------------------- Plot of BNN results -----------------------
plt.scatter(x_train, y_train,
            edgecolor='b', s=40, label="Datapoint")
for i in range(len(y_pred)):
    plt.plot(x_pred, y_pred[i], color='k', linestyle='dashed')
plt.plot(x_pred, mean_y_pred, color='coral', label="Average
   prediction")
plt.fill_between(x_pred, lower_std.flatten() , upper_std.flatten() ,
    color='b', alpha=.1)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig('figure_simple_BNN.pdf')
plt.show()
```

.4 Python code for Metropolis implementation used for producing figure 4.2

```
# # ----------------------------- IMPORTS -------------------------
   -

import numpy as np
import scipy.stats as ss
```

```
import matplotlib.pyplot as plt

# # ------------------------------ Defining functions ---------------
    ------------
# Defining target probability
def p(x):
    sigma = np.array([[1, 0.6], [0.6, 1]])  # Covariance matrix
    return ss.multivariate_normal.pdf(x, cov=sigma)

# # ------------------------------ Sampling  ------------------------
    ---
samples = np.zeros((1000, 2))
np.random.seed(42)

x = np.array([7, 0])
for i in range(1000):
    samples[i] = x
    # Gaussian proposal for symmetry
    x_prime = np.random.multivariate_normal(mean=x, cov=np.eye(2),
    size=1).flatten()
    acceptance_prob = min(1, (p(x_prime) )/ (p(x)))
    u = np.random.uniform(0, 1)
    if u <= acceptance_prob:
        x = x_prime
    else:
        x = x

# # ------------------------------ Vizualising  --------------------
    ------

# For vizualising normal contours
X, Y = np.mgrid[-3:3:0.05, -3:3:0.05]
X, Y = np.mgrid[-3:3:0.05, -3:3:0.05]
XY = np.empty(X.shape + (2,))
XY[:,:,0] = X; XY[:,:,1] = Y
```

```
target_distribution = ss.multivariate_normal(mean=[0,0], cov=[[1,
    0.6],[0.6, 1]])



plt.subplot(2, 2, 1) # row 1, col 2 index 1
plt.contour(X, Y, target_distribution.pdf(XY),cmap=plt.cm.Blues)
plt.ylim(-3,5)
plt.xlim(-3,8)
plt.subplot(2, 2, 2) # index 2
plt.plot(samples[0:100,0], samples[0:100,1], 'ro-', color="navy",
    linewidth=.2, markersize=.7,label="First 100 samples")
plt.contour(X, Y, target_distribution.pdf(XY),cmap=plt.cm.Blues)
plt.legend(loc="upper right",fontsize=9)
plt.ylim(-3,5)
plt.xlim(-3,8)
plt.subplot(2, 2, 3) # index 3
plt.plot(samples[0:200,0], samples[0:200,1], 'ro-', color="navy",
    linewidth=.2, markersize=.7,label="First 200 samples")
plt.contour(X, Y, target_distribution.pdf(XY),cmap=plt.cm.Blues)
plt.legend(loc="upper right",fontsize=9)
plt.ylim(-3,5)
plt.xlim(-3,8)
plt.subplot(2, 2, 4) # index 4
plt.plot(samples[0:300,0], samples[0:300,1], 'ro-', color="navy",
    linewidth=.2, markersize=.7, label="First 300 samples")
plt.contour(X, Y, target_distribution.pdf(XY),cmap=plt.cm.Blues)
plt.legend(loc="upper right",fontsize=9)
plt.ylim(-3,5)
plt.xlim(-3,8)
plt.savefig("metro_example.pdf")
plt.show()
```

.5 Python packages and specification for computer doing the evaluations in chapter 5

The code ran on a virtual machine for the Linux distribution Ubuntu 20.04.2 LTS. The virtual machine used VMWare Workstation 16.1.1 on a native Windows 10 64-bit version 2004. The hardware used is

- GPU: NVidia GeForce GTX 970
- RAM allocated to virtual machine: 9.5 GB
- Processor: Intel® Core™ i5-6600K CPU @ 3.50GHz × 4
- A harddrive allocated only to this virtual machine with free space of 427 GB

The packages required for reproducing the evalutions are

- `PyMC3`==3.11.2
- `Theano`==1.1.2
- `Arviz`==0.11.2
- `Numpy`==1.19.5
- `tensorflow`==2.4.1
- `tensorflow_probability`==0.12.2
- `sklearn`==0.24.2
- `numpy`==1.19.5
- `seaborn`==0.11.1
- `matplotlib.pyplot`==3.4.1

.6 Python code for the neural networks in table 5.2

The network with with early stopping is performed using the code

```
import tensorflow as tf
import numpy as np
```

```
import matplotlib.pyplot as plt
from keras.datasets import boston_housing
import time


tf.random.set_seed(40)

# ------------------------------ Prepare data ---------------------
(X_train, y_train), (X_test, y_test) = boston_housing.load_data(seed
    =3030)

# ------------------------------ Neural Network ---------------------
n_hidden = 10

model = tf.keras.Sequential([
    tf.keras.Input((13, ), name='feature'),
    tf.keras.layers.Dense(n_hidden, activation=tf.nn.relu),
    tf.keras.layers.Dense(1)
])
model.summary()

# Early stopping
es = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', mode='min', patience=10, min_delta=0.1)

start_time = time.time()
# Compile, train, and evaluate.
model.compile(optimizer='adam',
              loss='mean_squared_error',
              metrics=['mse'])
history = model.fit(X_train, y_train, epochs=300,
                    validation_split=0.3, callbacks=[es])

print("The algorithm ran", len(history.history['loss']), "epochs")
```

```
# ---------------------------- Overfitting? --------------------
train_acc = model.evaluate(X_train, y_train, verbose=0)[-1]
test_acc = model.evaluate(X_test, y_test, verbose=0)[-1]
print("--- %s seconds ---" % (time.time() - start_time))
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='validation')
plt.legend()
plt.grid()
plt.show()
```

The networks not using early stopping and their visualization of train and validation loss in figure 5.1, figure 5.2 and figure 5.3 are produced using the code

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import boston_housing
import time
from keras.regularizers import l2

tf.random.set_seed(40)
# ---------------------------- Prepare data ---------------------
(X_train, y_train), (X_test, y_test) = boston_housing.load_data(seed
    =3030)



# ---------------------------- Neural Network --------------------
reg_const = 0.3
n_hidden = 10

model = tf.keras.Sequential([
    tf.keras.Input((13, ), name='feature'),
    tf.keras.layers.Dense(n_hidden, activation=tf.nn.relu,
```

```
    kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const)),
    tf.keras.layers.Dense(1, kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const))
])
model.summary()

start_time = time.time()
# Compile, train, and evaluate.
model.compile(optimizer='adam',
              loss='mean_squared_error',
              metrics=['mse'])
history = model.fit(X_train, y_train, epochs=300, validation_split=
    0.3)
model.evaluate(X_test, y_test)


# ------------------------------ Overfitting? --------------------
train_acc = model.evaluate(X_train, y_train, verbose=0)[-1]
test_acc = model.evaluate(X_test, y_test, verbose=0)[-1]
print("--- %s seconds ---" % (time.time() - start_time))
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.legend()
plt.grid()
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.ylim(0, 200)
plt.savefig('figure_Boston_NN_1hidden_wd_loss.pdf')
plt.show()
```

where the network with no hidden layers are produced by removing the line

```
tf.keras.layers.Dense(n_hidden, activation=tf.nn.relu,
```

```
    kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const))
```

and the network with 1 hidden layer and no regularization is produced by removing the regularization arguments `kernel_regularizer` and `bias_regularizer` in

```
tf.keras.layers.Dense(n_hidden, activation=tf.nn.relu,
    kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const)),
     tf.keras.layers.Dense(1, kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const))
```

.7 Python code for the Bayesian neural networks in table 5.3

The Bayesian neural network with hierarchical model is implemented by the following code

```
# # ------------------------------ IMPORTS -----------------------
import sys
import time
from keras.datasets import boston_housing
from sklearn import metrics
import numpy as np
import pymc3 as pm
import theano
import arviz as az
from arviz.utils import Numba
import theano.tensor as tt
Numba.disable_numba()
Numba.numba_flag
floatX = theano.config.floatX
import seaborn as sns
sns.set_style("white")
import tensorflow as tf
```

```
# Ignore warnings - NUTS provide many runtimeWarning
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)

tf.random.set_seed(42)
# # ----------------------------- Loading Boston data --------------
(X_train, y_train), (X_test, y_test) = boston_housing.load_data(seed
    =3030)

#pad Xs with 1's to add bias
ones_train=np.ones(X_train.shape[0])
ones_test=np.ones(X_test.shape[0])
X_train=np.insert(X_train,0,ones_train,axis=1)
X_test=np.insert(X_test,0,ones_test,axis=1)


# # ----------------------------- Implementing a BNN function -----

def construct_bnn(ann_input, ann_output, n_hidden):
    # Initialize random weights between each layer
    init_1 = np.random.randn(X_train.shape[1], n_hidden).astype(
    floatX)*.1
    init_out = np.random.randn(n_hidden,1).astype(floatX)*.1
    with pm.Model() as bayesian_neural_network:
        ann_input = pm.Data("ann_input", X_train)
        ann_output = pm.Data("ann_output", y_train)

    # prior on hyper parameters for weight 1
        #mu1 = pm.Normal('mu1',shape=(X_train.shape[1], n_hidden),
    mu=0, sigma=1)
        mu1 = pm.Cauchy('mu1',shape=(X_train.shape[1], n_hidden),
    alpha=0, beta=1)
        sigma1 = pm.HalfNormal('sigma1',shape=(X_train.shape[1],
    n_hidden), sigma=1)
```

```
    # Input -> Layer 1
        weights_1 = pm.Normal('w_1', mu=mu1, sd=sigma1,
                              shape=(X_train.shape[1], n_hidden),
                              testval=init_1)
        acts_1 = pm.Deterministic('activations_1', tt.nnet.relu(tt.
    dot(ann_input, weights_1)))

    # prior on hyper parameters for weight_out
        mu_out = pm.Cauchy('mu_out',shape=(n_hidden, 1), alpha=0,
    beta=1)
        sigma_out = pm.HalfNormal('sigma_out',shape=(n_hidden, 1),
    sigma=1)

    # Layer 1 -> Output Layer
        weights_out = pm.Normal('w_out', mu=mu_out, sd=sigma_out,
                                shape=(n_hidden, 1),
                                testval=init_out)
        acts_out = pm.Deterministic('activations_out',tt.dot(acts_1,
     weights_out))


    #Define likelihood
        out = pm.Normal('out', mu=acts_out[:,0], sd=1, observed=
    ann_output)

    return bayesian_neural_network


# # ---------------- Sampling from posterior ---------
# Start time
tic = time.perf_counter() # for timing
bayesian_neural_network_NUTS = construct_bnn(X_train, y_train,
    n_hidden=10)
```

```
# Sample from the posterior using the NUTS samplper
with bayesian_neural_network_NUTS:
    trace = pm.sample(draws=3000, tune=1000, chains=3,target_accept=
    .90)


# # ------------------ Making predictions on training data ---------
ppc1=pm.sample_posterior_predictive(trace, model=
    bayesian_neural_network_NUTS)

# Taking the mean over all samples to generate a prediction
y_train_pred = ppc1['out'].mean(axis=0)


# Replace shared variables with testing set
pm.set_data(new_data={"ann_input": X_test, "ann_output": y_test},
    model=bayesian_neural_network_NUTS)



# # ----------------- Making predictions on test data ------
ppc2 = pm.sample_posterior_predictive(trace, model=
    bayesian_neural_network_NUTS)

# Taking the mean over all samples to generate a prediction
y_test_pred = ppc2['out'].mean(axis=0)

# End time
toc = time.perf_counter()
print(f"Run time {toc - tic:0.4f} seconds")

# Printing the performance measures
print('MSE (NUTS) on training data:', metrics.mean_squared_error(
    y_train, y_train_pred))
print('MSE (NUTS) on test data:', metrics.mean_squared_error(y_test,
```

```
    y_test_pred))
```

The Bayesian neural network with one hidden layer is implemented by the following code

```
# # ------------------------------ IMPORTS -------------------
import sys
import time
from keras.datasets import boston_housing
from sklearn import metrics
import numpy as np
import pymc3 as pm
import theano
import arviz as az
from arviz.utils import Numba
import theano.tensor as tt
Numba.disable_numba()
Numba.numba_flag
floatX = theano.config.floatX
# seaborn for vizualzing
import seaborn as sns
sns.set_style("white")
import matplotlib.pyplot as plt
import tensorflow as tf
# # ------------------------------ Print versions -------------


print("Running on Python version %s" % sys.version)
print(f"Running on PyMC3 version{pm.__version__}")
print("Running on Theano version %s" % theano.__version__)
print("Running on Arviz version %s" % az.__version__)
print("Running on Numpy version %s" % np.__version__)

# Ignore warnings - NUTS provide many runtimeWarning
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)


tf.random.set_seed(42)
```

```
# # ------------------------------ Loading Boston data ------------
(X_train, y_train), (X_test, y_test) = boston_housing.load_data(seed
    =3030)

#pad Xs with 1's to add bias
ones_train=np.ones(X_train.shape[0])
ones_test=np.ones(X_test.shape[0])
X_train=np.insert(X_train,0,ones_train,axis=1)
X_test=np.insert(X_test,0,ones_test,axis=1)



# # ------------------------------ Implementing a BNN function ------

def construct_bnn(ann_input, ann_output, n_hidden, prior_std):
    # Initialize random weights between each layer
    init_1 = np.random.randn(X_train.shape[1], n_hidden).astype(
   floatX)*prior_std
    init_out = np.random.randn(n_hidden,1).astype(floatX)*prior_std

    with pm.Model() as bayesian_neural_network:
        ann_input = pm.Data("ann_input", X_train)
        ann_output = pm.Data("ann_output", y_train)

    # Input -> Layer 1
        weights_1 = pm.Normal('w_1', mu=0, sd=prior_std,
                              shape=(X_train.shape[1], n_hidden),
                              testval=init_1)
        acts_1 = tt.nnet.relu(tt.dot(ann_input, weights_1))

    # Layer 1 -> Output Layer
        weights_out = pm.Normal('w_out', mu=0, sd=prior_std,
                                shape=(n_hidden, 1),
                                testval=init_out)
        acts_out = tt.dot(acts_1, weights_out)
```

```
    #Define likelihood
        out = pm.Normal('out', mu=acts_out[:,0], sd=1, observed=
    ann_output)

    return bayesian_neural_network


# # ------------------------------ Sampling from posterior -------
# Start time
tic = time.perf_counter() # for timing
bayesian_neural_network_NUTS = construct_bnn(X_train, y_train,
    n_hidden=10, prior_std=.1)

# Sample from the posterior using the NUTS samplper
with bayesian_neural_network_NUTS:
    trace = pm.sample(draws=3000, tune=1000, chains=3,target_accept=
    .9, random_seed=42)


# # ------------------- Making predictions on training data --------
ppc1=pm.sample_posterior_predictive(trace, model=
    bayesian_neural_network_NUTS, random_seed=42)

# Taking the mean over all samples to generate a prediction
y_train_pred = ppc1['out'].mean(axis=0)


# Replace shared variables with testing set
pm.set_data(new_data={"ann_input": X_test, "ann_output": y_test},
    model=bayesian_neural_network_NUTS)



# # ------------------- Making predictions on test data ------
```

```
ppc2 = pm.sample_posterior_predictive(trace, model=
    bayesian_neural_network_NUTS, random_seed=42)

# Taking the mean over all samples to generate a prediction
y_test_pred = ppc2['out'].mean(axis=0)

# End time
toc = time.perf_counter()
print(f"Run time {toc - tic:0.4f} seconds")

# Printing the performance measures
print('MSE (NUTS) on training data:', metrics.mean_squared_error(
    y_train, y_train_pred))
print('MSE (NUTS) on test data:', metrics.mean_squared_error(y_test,
     y_test_pred))


# -------------------------------- Plots --------------------
# Vizualize uncertainty
# Define examples for which you want to examine the posterior
    predictive:
example_vec=np.array
    ([1,2,4,9,10,11,15,16,22,24,27,28,30,44,55,62,68,72,84,93])
for example in example_vec:
     plt_hist_array=np.array(ppc2['out'])
     plt.hist(plt_hist_array[:,example], density=1, color="
    lightsteelblue", bins=30)
     plt.xlabel(f"Predicted value for example {example}",fontsize=13)
     plt.ylabel("Density",fontsize=13)
     plt.savefig(f'Python_code/Boston_BNN_1hidden_postpred_{example}.
    pdf')
     plt.show()
```

where the network with no hidden layers is implemented by replacing the lines

```
# Input -> Layer 1
```

```
    weights_1 = pm.Normal('w_1', mu=0, sd=prior_std,
                          shape=(X_train.shape[1], n_hidden),
                          testval=init_1)
    acts_1 = tt.nnet.relu(tt.dot(ann_input, weights_1))

# Layer 1 -> Output Layer
    weights_out = pm.Normal('w_out', mu=0, sd=prior_std,
                          shape=(n_hidden, 1),
                          testval=init_out)
    acts_out = tt.dot(acts_1, weights_out)
```

with

```
 # Input layer -> Output layer
        weights_out = pm.Normal('w_out', mu=0, sd=prior_std,
        shape=(X_train.shape[1], 1), testval=init_out)
    acts_out = pm.Deterministic(
        'activations_out', tt.dot(ann_input, weights_out))
```

.8 Python code for the neural networks in table 5.5

The neural network with early stopping is performed using the code

```
 import time
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
import seaborn as sns

tf.random.set_seed(40)
# ------------------------------ Prepare data --------------------
credit_data = pd.read_csv(
```

```
    "Python_code/data/UCI_Credit_Card.csv", encoding="utf-8",
    index_col=0)
credit_data.head()

# Data to numpy
data = np.array(credit_data)

# Extract labels
data_X = data[:, 0:23]
data_y = data[:, 23]



# # -------------------------- Subsamling credit data -------
X_train, X_test, y_train, y_test = train_test_split(
    data_X, data_y, test_size=0.30, random_state=3030)

N = 300
N_test = 100
X_train = X_train[0:N, :]
y_train = y_train[0:N]
X_test = X_test[0:N_test, :]
y_test = y_test[0:N_test]



# ------------------------- Neural Network ---------------

model = tf.keras.Sequential([
    tf.keras.Input((23, ), name='feature'),
    tf.keras.layers.Dense(10, activation=tf.nn.tanh),
    tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)
])
model.summary()

# Early stopping
es = tf.keras.callbacks.EarlyStopping(
```

```
    monitor='val_loss', mode='min', patience=0, min_delta=0)

start_time = time.time()

# Compile, train, and evaluate.
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['binary_crossentropy'])
history = model.fit(X_train, y_train,  validation_split=0.3,
                    epochs=1000, callbacks=[es])
print("The algorithm ran", len(history.history['loss']), "epochs")

print("--- %s seconds ---" % (time.time() - start_time))


# ------------------------ Overfitting? ----------------
train_acc = model.evaluate(X_train, y_train, verbose=0)[-1]
test_acc = model.evaluate(X_test, y_test, verbose=0)[-1]
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

# taking mean of summed cross-entropy loss
train_loss = np.array(history.history['loss'])
val_loss = np.array(history.history['val_loss'])

plt.plot(train_loss, label='train')
plt.plot(val_loss, label='validation')
plt.legend()
plt.grid()
plt.show()
```

The networks not using early stopping and their visualization of train and validation loss in figure 5.5, 5.6 and figure 5.7 are produced using the code

```
from keras.regularizers import l2
import time
import tensorflow as tf
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
import seaborn as sns
start_time = time.time()
tf.random.set_seed(40)

# ------------------------------ Prepare data ----------------
credit_data = pd.read_csv(
    "Python_code/data/UCI_Credit_Card.csv", encoding="utf-8",
   index_col=0)
credit_data.head()

# Data to numpy
data = np.array(credit_data)

# Extract labels
data_X = data[:, 0:23]
data_y = data[:, 23]

# # ------------------------------ Subsamling credit data ------
X_train, X_test, y_train, y_test = train_test_split(
    data_X, data_y, test_size=0.30, random_state=3030)

N = 300
N_test = 100
X_train = X_train[0:N, :]
y_train = y_train[0:N]
X_test = X_test[0:N_test, :]
y_test = y_test[0:N_test]

# ------------------------------ Neural Network ----------------
reg_const = 0.1
n_hidden = 10
```

```
model = tf.keras.Sequential([
    tf.keras.Input((23, ), name='feature'),
    tf.keras.layers.Dense(n_hidden, activation=tf.nn.tanh,
   kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const)),
    tf.keras.layers.Dense(1, activation=tf.nn.sigmoid,
   kernel_regularizer=l2(
        reg_const), bias_regularizer=l2(reg_const))
])
model.summary()

# Compile, train, and evaluate.
val_ratio = 0.3
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['binary_crossentropy'])
history = model.fit(X_train, y_train, epochs=1000,
                    validation_split=val_ratio)

model.evaluate(X_test, y_test)

print("--- %s seconds ---" % (time.time() - start_time))


# ------------------------------ Overfitting? ----------------

train_acc = model.evaluate(X_train, y_train, verbose=0)[-1]
test_acc = model.evaluate(X_test, y_test, verbose=0)[-1]
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

# taking mean of summed cross-entropy loss
train_loss = np.array(history.history['loss'])
val_loss = np.array(history.history['val_loss'])
```

```
plt.plot(train_loss, label='train')
plt.plot(val_loss, label='validation')
plt.legend()
plt.grid()
plt.ylim(0.4, 1)
plt.savefig('Python_code/figure_Credit_NN_1hidden_wd_loss.pdf')
plt.show()
```

where the network with no hidden layers are produced by removing the line

```
 tf.keras.layers.Dense(n_hidden, activation=tf.nn.tanh,
    kernel_regularizer=l2(
         reg_const), bias_regularizer=l2(reg_const)),
```

and the network with 1 hidden layer and no regularization is produced by removing the regularization arguments `kernel_regularizer` and `bias_regularizer` in

```
tf.keras.layers.Dense(n_hidden, activation=tf.nn.tanh,
    kernel_regularizer=l2(
         reg_const), bias_regularizer=l2(reg_const)),
tf.keras.layers.Dense(1, activation=tf.nn.sigmoid,
    kernel_regularizer=l2(
         reg_const), bias_regularizer=l2(reg_const))
```

.9 Python code for the Bayesian neural networks in table 5.6

The Bayesian neural network with hierarchical model is implemented by the following code

```
# # ------------------------------ IMPORTS -----------------
import warnings
from sklearn.metrics import accuracy_score, log_loss
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import sys
```

```
import time
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
import pymc3 as pm
import theano
import arviz as az
from arviz.utils import Numba
import theano.tensor as tt
from scipy.stats import mode
Numba.disable_numba()
Numba.numba_flag
floatX = theano.config.floatX
sns.set_style("white")



# # ------------------------------ Print versions -----------------
print("Running on Python version %s" % sys.version)
print(f"Running on PyMC3 version{pm.__version__}")
print("Running on Theano version %s" % theano.__version__)
print("Running on Arviz version %s" % az.__version__)
print("Running on Numpy version %s" % np.__version__)

# Ignore warnings - NUTS provide many runtimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

tf.random.set_seed(42)



# # ------------------------------ Loading credit data ---
credit_data = pd.read_csv("Python_code/data/UCI_Credit_Card.csv",
                          encoding="utf-8", index_col=0, delimiter="
    ,")
credit_data.head()
# Data to numpy
```

```
data = np.array(credit_data)
# seperating labels from features
data_X = data[:, 0:23]
data_y = data[:, 23]


# # ----------------------------- Subsamling credit data --------
X_train, X_test, y_train, y_test = train_test_split(
    data_X, data_y, test_size=0.30, random_state=3030)

N = 300
N_test = 100
X_train = X_train[0:N, :]
y_train = y_train[0:N]
X_test = X_test[0:N_test, :]
y_test = y_test[0:N_test]


# pad Xs with 1's to add bias
ones_train = np.ones(X_train.shape[0])
ones_test = np.ones(X_test.shape[0])
X_train = np.insert(X_train, 0, ones_train, axis=1)
X_test = np.insert(X_test, 0, ones_test, axis=1)

# # ----------------------------- Implementing a BNN function -----


def construct_bnn(ann_input, ann_output, n_hidden):

    with pm.Model() as bayesian_neural_network:
        ann_input = pm.Data("ann_input", X_train)
        ann_output = pm.Data("ann_output", y_train)

        # prior on hyper parameters for weight 1
        mu1 = pm.Cauchy('mu1', shape=(
```

```
            X_train.shape[1], n_hidden), alpha=0, beta=1)
        sigma1 = pm.HalfNormal('sigma1', shape=(
            X_train.shape[1], n_hidden), sigma=1)

        # Weights from input to hidden layer
        weights_in_1 = pm.Normal(
            "w_in_1", mu1, sigma1, shape=(X_train.shape[1], n_hidden
))

        # prior on hyper parameters for weight_out
        mu_out = pm.Cauchy('mu_out', shape=(n_hidden, 1), alpha=0,
beta=1)
        sigma_out = pm.HalfNormal('sigma_out', shape=(n_hidden, 1),
sigma=1)
        # Weights from hidden layer to output
        weights_1_out = pm.Normal(
            "weights_out", mu_out, sigma=sigma_out, shape=(n_hidden,
 1))

        # Build neural-network using tanh activation function
        act_1 = pm.math.tanh(pm.math.dot(ann_input, weights_in_1))

        output = pm.Deterministic(
            "output", pm.math.sigmoid(tt.dot(act_1, weights_1_out)))

        # Binary classification -> Bernoulli likelihood
        out = pm.Bernoulli(
            "out",
            output,
            observed=ann_output,
            total_size=y_train.shape[0],
        )

    return bayesian_neural_network
```

```
# # ---------------------------- Sampling from posterior --------
tic = time.time()  # for timing
bayesian_neural_network_NUTS = construct_bnn(X_train, y_train,
    n_hidden=10)

# Sample from the posterior using the NUTS samplper
draws = 1500
tune = 10**3
chains = 3
target_accept = .9
with bayesian_neural_network_NUTS:
    trace = pm.sample(draws=draws, tune=tune, chains=chains,
                      target_accept=target_accept)


y_train_pred = (trace["output"]).mean(axis=0)


# Replace shared variables with testing set
pm.set_data(new_data={"ann_input": X_test, "ann_output": y_test},
            model=bayesian_neural_network_NUTS)

ppc2 = pm.sample_posterior_predictive(
    trace, var_names=["output"], model=bayesian_neural_network_NUTS)
y_test_pred = (ppc2["output"]).mean(axis=0)

# y_test_pred = np.append(y_test_pred,1-y_test_pred,axis=1)

# end time
toc = time.time()
print(f"Running MCMC completed in {toc - tic:} seconds")

# Printing the performance measures
print('Cross-entropy loss on train data = {}'.format(log_loss(
```

```
    y_train, y_train_pred)))
print('Cross-entropy loss on test data = {}'.format(log_loss(y_test,
     y_test_pred)))
```

The Bayesian neural network with one hidden layer is implemented by the following code

```
# # ----------------------------- IMPORTS -----------------
import warnings
from sklearn.metrics import accuracy_score, log_loss
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import sys
import time
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
import pymc3 as pm
import theano
import arviz as az
from arviz.utils import Numba
import theano.tensor as tt
from scipy.stats import mode
Numba.disable_numba()
Numba.numba_flag
floatX = theano.config.floatX
sns.set_style("white")



# # ----------------------------- Print versions -----------------
print("Running on Python version %s" % sys.version)
print(f"Running on PyMC3 version{pm.__version__}")
print("Running on Theano version %s" % theano.__version__)
print("Running on Arviz version %s" % az.__version__)
print("Running on Numpy version %s" % np.__version__)
```

```
# Ignore warnings - NUTS provide many runtimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

tf.random.set_seed(42)


# # ----------------------------- Loading credit data ------
credit_data = pd.read_csv("Python_code/data/UCI_Credit_Card.csv",
                          encoding="utf-8", index_col=0, delimiter="
    ,")
credit_data.head()
# Data to numpy
data = np.array(credit_data)
# seperating labels from features
data_X = data[:, 0:23]
data_y = data[:, 23]


# # ----------------------------- Subsamling credit data -----
X_train, X_test, y_train, y_test = train_test_split(
    data_X, data_y, test_size=0.30, random_state=3030)

N = 300
N_test = 100
X_train = X_train[0:N, :]
y_train = y_train[0:N]
X_test = X_test[0:N_test, :]
y_test = y_test[0:N_test]


# pad Xs with 1's to add bias
ones_train = np.ones(X_train.shape[0])
ones_test = np.ones(X_test.shape[0])
X_train = np.insert(X_train, 0, ones_train, axis=1)
X_test = np.insert(X_test, 0, ones_test, axis=1)
```

```
# # ------------------------------ Implementing a BNN function -----
def construct_bnn(ann_input, ann_output, n_hidden, prior_std):

    with pm.Model() as bayesian_neural_network:
        ann_input = pm.Data("ann_input", X_train)
        ann_output = pm.Data("ann_output", y_train)

        # Weights from input to hidden layer
        weights_in_1 = pm.Normal(
            "w_in_1", 0, sigma=prior_std, shape=(X_train.shape[1],
   n_hidden))

        # Weights from hidden layer to output
        weights_1_out = pm.Normal(
            "weights_out", 0, sigma=prior_std, shape=(n_hidden, 1))

        # Build neural-network using tanh activation function
        act_1 = pm.math.tanh(pm.math.dot(ann_input, weights_in_1))
        output = pm.Deterministic(
            "output", pm.math.sigmoid(tt.dot(act_1, weights_1_out)))

        # Binary classification -> Bernoulli likelihood
        out = pm.Bernoulli(
            "out",
            output,
            observed=ann_output,
            total_size=y_train.shape[0],  # IMPORTANT for
   minibatches
        )

    return bayesian_neural_network
```

```
# # ------------------ Sampling from posterior -----
tic = time.time()  # for timing
bayesian_neural_network_NUTS = construct_bnn(
    X_train, y_train, n_hidden=10, prior_std=1)

# Sample from the posterior using the NUTS samplper
draws = 1500
tune = 10**3
chains = 3
target_accept = .9
with bayesian_neural_network_NUTS:
    trace = pm.sample(draws=draws, tune=tune, chains=chains,
                      target_accept=target_accept)


y_train_pred = (trace["output"]).mean(axis=0)

# Replace shared variables with testing set
pm.set_data(new_data={"ann_input": X_test, "ann_output": y_test},
            model=bayesian_neural_network_NUTS)

ppc2 = pm.sample_posterior_predictive(
    trace, var_names=["output"], model=bayesian_neural_network_NUTS)
y_test_pred = (ppc2["output"]).mean(axis=0)


# end time
toc = time.time()
print(f"Running MCMC completed in {toc - tic:} seconds")

# Printing the performance measures
print('Cross-entropy loss on train data = {}'.format(log_loss(
   y_train, y_train_pred)))
print('Cross-entropy loss on test data = {}'.format(log_loss(y_test,
    y_test_pred)))
```

```
# Vizualize uncertainty
# Define examples for which you want to examine the posterior
   predictive:
example_vec = np.array([5, 11, 25, 88])
for example in example_vec:
    plt_hist_array = np.array(ppc2['output'])
    plt.hist(plt_hist_array[:, example], density=1,
             color="lightsteelblue", bins=30)
    plt.xlabel(f"Predicted probability for example {example}",
   fontsize=13)
    plt.ylabel("Density", fontsize=13)
    plt.savefig(f'Python_code/Credit_BNN_1hidden_postpred_{example}.
   pdf')
    plt.show()
```

where the network with no hidden layers is implemented by replacing the lines

```
 # Weights from input to hidden layer
        weights_in_1 = pm.Normal(
            "w_in_1", 0, sigma=prior_std, shape=(X_train.shape[1],
   n_hidden))

# Weights from hidden layer to output
    weights_1_out = pm.Normal(
            "weights_out", 0, sigma=prior_std, shape=(n_hidden, 1))

# Build neural-network using tanh activation function
    act_1 = pm.math.tanh(pm.math.dot(ann_input, weights_in_1))
    output = pm.Deterministic(
            "output", pm.math.sigmoid(tt.dot(act_1, weights_1_out)))
```

with

```
 # Weights from hidden layer to output
        weights_in_out = pm.Normal("weights_out", 0, sigma=prior_std
```

```
, shape=(X_train.shape[1],1))

    # Build neural-network using tanh activation function
    output = pm.Deterministic("output", pm.math.sigmoid(tt.dot(
ann_input, weights_in_out)))
```