BagDrop

Computationally Feasible Approximate Bagging
with Neural Networks

Friso Kingma

TUDelft Delft
University of
Technology

BagDrop
Computationally Feasible Approximate Bagging with Neural Networks

BACHELOR GRADUATION THESIS

Friso Kingma

4320557

September 8, 2017

Project duration:	February, 2017 – Juli 9, 2017	
Thesis committee:	Prof. dr. M. Loog,	TU Delft, supervisor
	Dr. ir. F.H. van der Meulen,	TU Delft, supervisor
	Dr. ir. B. van den Dries,	TU Delft

Bachelor of Applied Mathematics
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology

Preface & Abstract

1-1 Preface

This thesis was not possible without the help and support of many. However, there are a few that must be mentioned. First of all, I would like to thank my supervisors for their support and patience. They have helped me over a long period of time. Secondly, I would like to thank my brother for providing the research proposal, endless support and positive feedback. He has introduced me to Machine Learning and convinced me to take the educated leap of faith.

1-2 Abstract

Overfitting is a common problem when learning models from noisy observational data. This problem is especially present in very flexible models, such as Neural Networks, which can easily fit to spurious patterns in the data that are not indicative of true underlying patterns. One technique that conquers the problem of overfitting is Bagging, an ensemble method. However, Bagging can be a slow technique, since its computational cost scales linearly with the size of the ensemble. We propose a Dropout-inspired method, BagDrop, as a solution to the problem of computationally high cost of Bagging. We conduct experiments on a regression problem with fully-connected Neural Networks. Our results show that BagDrop does well in terms of generalization performance and computational cost. Our encouraging results provide a proof-of-concept that indicates a promising direction for future research.

Table of Contents

Introduction

2-1 Introduction

In the field of *machine learning* we want to obtain new knowledge, so that we can use this knowledge to create software or hardware that can learn theirselves. Specifically, machine learning explores the study and construction of algorithms that can learn from and make predictions on data. Such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions through building a *model* from the data available.

In case of *supervised learning*, we typically aim to learn a conditional model: to make predictions, given the value of some other variable. We train this model by some observed, hopefully representational, samples. An optimal scenario will allow for the obtained predictor to always correctly determine the value, given some other value. This requires the learning algorithm to generalize from the observed sample to future situations in a "reasonable" way. Otherwise, we will get false predictions.

One problem is that of image classification: given some image, could the algorithm tell us what is seen on the image? Through the yearly ImageNet competition (Russakovsky et al., 2015), it has become clear that Neural Networks (Goodfellow et al., 2016), given a very large amount of labeled images, are extraordinarily good at solving this image classification task.

2-2 Problem Description & Research Questions

However, when we want to challenge the problem of supervised learning, we are restricted by our observed sample. This observed sample is usually not fully representative to future data, due to the presence of undesirable noise. This noise may be unique for every observed sample. Therefore, when fitting a model to obtain a predictor, we do not want the model to adapt to the data too much. Especially when using Neural Networks, which are very flexible models, we could capture every sample's unique noise. This way, the obtained predictor would only predict reasonable values when fed with the observed sample and generalize very poorly on future samples. The problem of adaptation of supervised learning models to noise in the observed sample is called *overfitting*.

Luckily, there are many techniques that conquer the problem of *overfitting*. Many researchers have investigated the technique of combining the predictions of multiple predictors to produce a single final predictor (Breiman, 1996) (Clemen, 1989) (Perrone, 1993) (Wolpert, 1992). One technique to create and combine multiple predictors is Bagging (Breiman, 1996). Bagging is based on resampling from the observed sample to obtain bootstrap samples (Efron and Tibshirani, 1994), and using this bootstrap samples to train the model. The resulting predictors are averaged to obtain an aggregated predictor. However, Bagging could be a very slow technique,

especially when using Neural Networks. A more recent technique that creates and combines multiple Neural Network architectures is Dropout (Srivastava et al., 2014). The created architectures are called Dropout masks. Dropout uses the special architecture of a Neural Network in a very fast and clever way. That brings us to the first research question:

> **Research Question 1**
>
> How can we apply Bagging in a Dropout way, such that we retain the speed of Dropout, while still having similar generalization properties as Bagging?

We propose **BagDrop** as answer to this research question. In BagDrop, every Dropout mask is trained by its own bootstrap sample and combined as Dropout prescribes. But how does the BagDrop procedure perform? Ideally, the generalization performance of BagDrop is similar to Bagging. This brings us to the second research question:

> **Research Question 2**
>
> How does the BagDrop procedure perform in comparison to Bagging, in terms of generalization performance?

Furthermore, the computational cost of BagDrop must be more or less the same as Dropout. This brings us to the third and last research question:

> **Research Question 3**
>
> How does the BagDrop procedure perform in comparison to Dropout, in terms of computational cost?

To get a good understanding of how BagDrop works, why BagDrop could perform well in terms of generalization performance and computational cost and how BagDrop is used in Neural Networks, we will give a very comprehensive introduction to probabilistic models, supervised learning, the underlying theory of Neural Networks, how to use Neural Networks, overfitting in Neural Networks, what Bagging is, how to apply Bagging, why Bagging could perform well in terms of generalization performance, what Dropout is, how to apply Dropout and why Dropout is favorable to solve the computational cost problem of Bagging.

To answer the first research question, we will give a understanding of how BagDrop works and argue why it could perform similar to Bagging in terms of generalization performance and to Dropout in terms of computational cost. Afterwards, we will carry out a proof-of-concept to answer the second and third research question. We will investigate the performance of BagDrop empirically with a simulated toy-dataset. We show that BagDrop does have better generalization performance than Bagging and more or less the same computational cost as Dropout in this setting. Finally, we will draw conclusions and start a discussion.

In the following section, a short summary of every chapter and section is given. Moreover, there is a summary of the chapter at the beginning of every chapter.

BagDrop

2-3   Thesis Outline

Chapter (3), (4) and (5) are meant to equip the reader with the background information that is necessary to understand BagDrop and the rest of the thesis.

In chapter (3) we will give a detailed explanation of probabilistic models and how to interpret them. Afterwards, we will introduce the concepts classification and regression. Then, some simple linear models are introduced, linear regression and logistic regression. At the end of chapter (3) we will introduce Neural Networks. Finally, we will give a detailed example of a Neural Network.

In chapter (4) we will explain how to train/fit models on observed data. We will introduce the squared error loss function and introduce the principle of maximum likelihood estimation. Afterwards, the likelihood based loss function for neural networks is given and how to optimize the parameters of the model. Stochastic gradient descent and the Adam optimizer are also introduced.

Chapter (5) has two parts. The first part introduces the concept of overfitting and the second part will explain techniques to prevent overfitting. First we will define generalization (performance) and the bias-variance trade-off. Afterwards, overfitting with Neural Networks is explained and why Neural Networks are sensitive to overfitting. We will define regularization and give regularization techniques that are important for our thesis: Bagging and Dropout.

Chapter (6) introduces BagDrop as answer to the first research question. In chapter (7) we will give answer to the second and third research question.

In chapter (8) we will draw conclusions and start a discussion.

2-4   Notation

Given a data matrix $\underline{\mathbf{x}} \in \mathbb{R}^{N \times P}$ in which each row $\mathbf{x}_i$ with $i \in \{1, .., N\}$ defines a vector of observed features (also called inputs).

$$\underline{\mathbf{x}} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ . \\ . \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,P} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,P} \end{pmatrix} \tag{2-1}$$

Each feature vector $\mathbf{x}_i$ is assigned to an observed target variable y_i, where $y_i \in \mathbb{R}^d$ in the case of regression and $y_i \in \{0, 1, ..., K-1\}$ in the case of classification with K classes. We define an observed *training set* D as a set of observations of both inputs and outputs $D = \{(\mathbf{x}_i, y_i)\}$ with $i \in \{1, ..., N\}$, and an observed test sample T and observed validations sample O are analogously defined. We will denote a random training sample with S and a random validation sample with V.

We use uppercase letters such as X and Y when referring to random variables. If X is a vector, its components are denoted by subscripts X_j. We will denote the distribution of X with $Pr(X)$, the joint distribution of X and Y with $Pr(X, Y)$ and the conditional distribution of Y given X with $Pr(Y|X)$. Furthermore, we will denote the chance of X taking on value $\mathbf{x}$ or Y taking on value y with $Pr(X = \mathbf{x})$ and $Pr(Y = y)$ respectively.

2-5 Literature Used

This thesis is heavily based on the following books

- *Pattern Recognition and Machine Learning* (Bishop, 2006)

- *The Elements of Statistical Learning* (Friedman et al., 2001)

- *Applied Predictive Modelling* (Kuhn and Johnson, 2013)

- *Deep Learning* (Goodfellow et al., 2016)

- *Mathematical Statistics and Data Analysis* (Rice, 2006)

Model

This chapter we will give a detailed explanation of probabilistic models and how to interpret them. Afterwards, we will introduce the concepts classification and regression. Then, some simple linear models are introduced: linear regression and logistic regression. This knowledge is needed to understand Neural Networks. At the end of chapter we will introduce Neural Networks. Finally, we will give a detailed example of a Neural Network.

3-1 Probabilistic Models

We are interested to learn probabilistic models of high-dimensional data. Perhaps the most complete probabilistic model is the joint distribution $Pr(X, Y)$ over the random variables X and Y. Since we are just interested in the target variables, we can restrict ourselves to researching Y given some X. Note that we often do not know how $Y|X$ is really distributed. Therefore, we will denote *the unknown underlying process* that produces $Y|X$ as $Pr^*(Y|X)$.

We attempt to approximate this underlying process with a chosen model $Pr_\theta(Y|X)$ with parameters θ. In this context, learning is the process of finding the parameters θ such that the probability distribution function given by the model, $Pr_\theta(Y|X)$, approximates the true distribution function of the data $Pr^*(Y|X)$. Such that for any observed $(\mathbf{x}, y)$

$$Pr_\theta(Y = y|X = \mathbf{x}) \approx Pr^*(Y = y|X = \mathbf{x}) \tag{3-1}$$

We will explain two types target variables Y. Namely, the continuous type and the categorical type. In case we have to deal with categorical Y, we want to know which $Y = y$ belongs to an observed feature vector $X = \mathbf{x}$. This is known as classification. We are often not particularly interested in the value of Y given some $X = \mathbf{x}$, but we will focus on the *chance* of $Y = y$ given some $X = \mathbf{x}$. Therefore, we can model $Pr^*(Y|X)$ directly. However, when we have to deal with a continuous target variable Y, we are more interested in Y given some X. In particular, we are interested in $\mathbb{E}^*(Y|X)$. Therefore, we will model this expectation.

With modelling and learning, we have to keep three things in mind.

1. We wish to have our model be sufficiently flexible to be able to adapt to the data, such that we have a chance of obtaining a sufficiently accurate model.

2. At the same time, our model is restricted by our dataset, which is often not fully representative to future data. So we do not want to adapt to the data too much, otherwise our model will not be approximately representative to the underlying process.

3. Moreover, ideally, taking (1) and (2) into account must not slow down our learning process.

3-2   Simple Linear Models

3-2-1   Regression

Let $X \in \mathbb{R}^P$ be a feature vector and let $Y \in \mathbb{R}^d$ be the corresponding target variable. Suppose that our data arose from a statistical model

$$Y = \mu_\theta(X) + \epsilon \tag{3-2}$$

where the random noise ϵ is independent and identically distributed (i.i.d.) and $\mathbb{E}(\epsilon) = 0$. Note that for this model, $\mu(X) = \mathbb{E}_\theta(Y|X)$, and in fact the conditional distribution $Pr_\theta(Y|X)$ depends on X through the conditional mean $\mu_\theta(\mathbf{x})$. If, for once, we assume that the random noise is normally distributed $\epsilon \sim N(0, \sigma^2)$, then

$$Y|X \sim N(\mu_\theta(X), \sigma^2) \tag{3-3}$$

The simplest linear model for regression is one that involves a linear combination of the features

$$\mu_\theta(X) = \langle X, \theta \rangle \tag{3-4}$$

where $X \in \mathbb{R}^{P+1}$, $X = (X_0, ..., X_P)^T$, $\theta = (W_0, ..., W_P)^T$ and $X_0 = 1$. This is simply known as *linear regression*. The model is linear in the parameters. This imposes significant limitations on the model. We therefore extend the class of models by considering linear combinations of fixed (nonlinear) *basis functions* of the features $\boldsymbol{\phi}$.

$$\mu_\theta(X) = \langle \boldsymbol{\phi}(X), \theta \rangle \tag{3-5}$$

The vector θ is called the parameter vector of our model. By using nonlinear basis functions we allow the model $q_\theta(X)$ to be a nonlinear function of the feature vector X. Models of the form (3-5) are called linear models, because this model is linear in the parameters.

3-2-2   Classification

So far we have concentrated on the continuous target variable $Y \in \mathbb{R}^d$. In the case of classification $Pr^*(Y|X)$ is modelled directly. For example, for two-class data, it reasonable that the data arise from independent binary trials, with the probability of one particular outcome being $q_\theta(X)$, and the other $1 - q_\theta(X)$. Then the conditional distribution becomes

$$Y|X \sim \text{Bern}(q_\theta(X)) \tag{3-6}$$

For classification problems, we wish to predict discrete class labels. More generally we wish to model probabilities that lie in the range $[0, 1]$. To achieve this, we consider a generalization of (3-4) in which we transform the linear function of W using a nonlinear function so that

$$q_\theta(X) = f(\langle X, W \rangle) \tag{3-7}$$

BagDrop

In the statistics literature $f^{-1}(\cdot)$ is known as the *link function*, whereas its inverse $f(\cdot)$ is called a *activation function* in the machine learning literature. We could use the sigmoid function for $f(\cdot)$ as activation function, which has the form

$$\sigma : \mathbb{R} \to [0, 1]$$
$$\sigma(\alpha) = \frac{1}{1 + \exp(-\alpha)} \tag{3-8}$$

Now, we turn to the case of a two-class target variable $Y \in \{1, 2\}$. The conditional probability of one of the two classes can be written as a sigmoid function acting on a linear function of the basis functions ϕ so that (3-7) becomes

$$Pr_\theta(Y = 1|X) = \sigma(\langle \phi(X), \theta \rangle) \tag{3-9}$$

This is simply known as *logistic regression*. If we turn to a K-class target variable $Y \in \{1, 2, ..., K\}$. We want probabilities for the K classes that sum to one and at the same time remain in $[0, 1]$. Then we get the following model

$$Pr_\theta(Y = k|X) = \frac{\exp\left(\langle \sigma(X), W_k \rangle\right)}{\sum_{l=1}^{K} \exp\left(\langle \sigma(X), W_l \rangle\right)} \tag{3-10}$$

where $k \in \{1, 2, ..., K - 1\}$ and $W_k = \{W_{k0}, ..., W_{kP}\}^T$.

3-3 Neural Networks

In order to get more flexible models than described by (3-4) and (3-9), it is an option to adapt the basis functions to the data. An approach is to fix the number of basis functions in advance but allow them to be adaptive, in other words to use parametric forms for the basis functions in which the parameters are adapted during training. One model of this type is the feed-forward *neural network*, also known as the multilayer perceptron.

Generally, the linear model for *regression* and *classification* are based on a linear combination of fixed nonlinear basis function ϕ as described by

$$\mu_\theta(X) = \langle \phi(X), \theta \rangle$$
$$q_\theta(X) = \sigma(\langle \phi(X), \theta \rangle) \tag{3-11}$$

We will denote a neural network function with parameters θ with

$$\text{NeuralNet}_\theta(X) = f(\langle \phi(X), \theta \rangle), \tag{3-12}$$

where $f(\cdot)$ is a nonlinear activation function in the case of classification and the identity in the case of regression. In the context of neural networks, we define the activation function $f(\cdot)$ as the *final activation function*.
The goal is to extend (3-11) by making the basis function $\phi(X)$ depend on parameters and

then make these parameters adjustable. The basis functions will be adjustable along with the parameters θ, during training. The parameters are called *weights* in the context of neural networks. In neural networks the basis functions follow the same form as (3-11).

For example, we shall construct a two stage neural network. First, derived features $\{Z_m\}_{m=1}^M$ are created from linear combinations of the input features $\{X_p\}_{p=1}^P$.

$$
\begin{aligned}
Z_1 &= \sigma\Big(\langle X, W_1^{(1)}\rangle\Big) \\
Z_2 &= \sigma\Big(\langle X, W_2^{(1)}\rangle\Big) \\
&\vdots \\
Z_M &= \sigma\Big(\langle X, W_M^{(1)}\rangle\Big)
\end{aligned}
\tag{3-13}
$$

where $W_m^{(1)} = \{W_{m0}^{(1)}, ..., W_{mP}^{(1)}\}$, $m \in \{1, ..., M\}$ and $\sigma(\cdot)$ some activation function. Then, final derived features $\{\mathrm{NeuralNet}_k(X)\}_{k=1}^K$ are created from linear combinations of the features $\{Z_m\}_{m=1}^M$.

$$
\begin{aligned}
\mathrm{NeuralNet}_1(X) &= f_1\Big(\langle Z, W_1^{(2)}\rangle\Big) \\
\mathrm{NeuralNet}_2(X) &= f_2\Big(\langle Z, W_2^{(2)}\rangle\Big) \\
&\vdots \\
\mathrm{NeuralNet}_K(X) &= f_K\Big(\langle Z, W_K^{(2)}\rangle\Big)
\end{aligned}
\tag{3-14}
$$

where $W_k^{(2)} = \{W_{k0}^{(2)}, ..., W_{kM}^{(2)}\}$, $k \in \{1, ..., K\}$, $Z = (Z_0, ..., Z_M)^T$, $Z_0 = 1$ and $f(\cdot)$ some final activation function. The activation function $\sigma(\cdot)$ is usually chosen to be the sigmoid (3-8). But we could also use the so called exponential linear unit (ELU) or the rectified linear unit (ReLU) as activation functions, which are respectively

$$
\begin{aligned}
\mathrm{ELU}(\alpha) &= \begin{cases} \exp(\alpha) - 1 & \text{if } x < 0 \\ \alpha & \text{if } x \geq 0 \end{cases} \\
\mathrm{ReLU}(\alpha) &= \begin{cases} 0 & \text{if } x < 0 \\ \alpha & \text{if } x \geq 0 \end{cases}
\end{aligned}
\tag{3-15}
$$

The final activation function allows a final transformation of the $\langle Z, W_k^{(2)}\rangle$. For regression we typically choose $K = 1$ and the identity function as final activation function

$$
\mathrm{NeuralNet}(X) = f\Big(\langle Z, W^{(2)}\rangle\Big) = \langle Z, W^{(2)}\rangle.
\tag{3-16}
$$

However, neural networks can handle K continuous target variables. For K-class classification we can choose the softmax function as the final activation function

$$
\mathrm{NeuralNet}_k(X) = f_k\Big(\langle Z, W_k^{(2)}\rangle\Big) = \frac{\exp\Big(\langle Z, W_k^{(2)}\rangle\Big)}{\sum_{l=1}^K \exp\Big(\langle Z, W_k^{(2)}\rangle\Big)}.
\tag{3-17}
$$

A simple calculation shows that this is exactly the transformation used in logistic regression (3-10), and produces positive estimates that sum to one.

The units in the middle of the network, computing the derived features Z_m, are called *hidden units* because the values Z_m are not directly observed. The features are called input units and the last units NeuralNet$_k$ are called the output units. We will define the hidden units and the output units as layers, so this model will be a 2 layer neural network. Picture 5-3 shows a diagram of the constructed neural network, but in general there can be more than one hidden layer.

Figure 3-1: Diagram of a 2 layer neural network
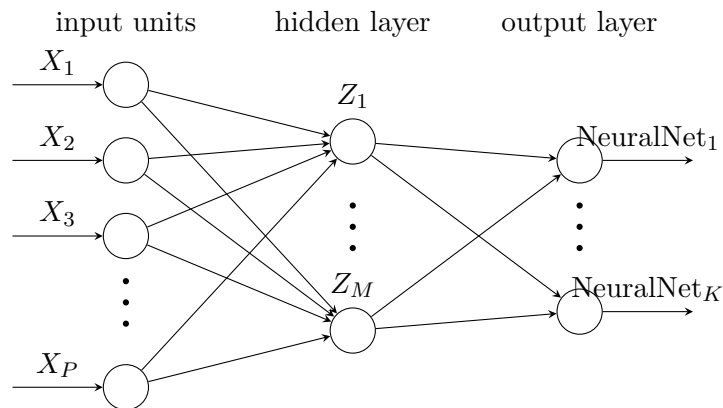
3-3-1   Notation of Neural Networks, Regression and Classification

In the following parts of the thesis, we will often talk about regression and classification without taking into account the underlying model, such as neural networks. In that case, we will denote regression with $\mu_\theta(X)$ and classification with $q_\theta(X)$. Besides, we will also talk about neural networks, regardless which problem we want to solve (classification or regression). We will denote neural networks with NeuralNet$_\theta(X)$. If NeuralNet$_\theta(X)$ is a vector of outputs, its components are denoted by NeuralNet$_{\theta,k}(X)$. Furthermore, we will denote $M_\theta(X)$ when we talk about an arbitrary model, regardless the problem we want to solve (classification or regression) and regardless the underlying model.

Training

This chapter we will explain how to train/fit (Neural Network) models on training data. We will introduce the squared error loss function and introduce the principle of maximum likelihood estimation. Afterwards, the likelihood based loss function for neural networks is given and how to optimize the parameters of the model. Stochastic gradient descent and the Adam optimizer are also introduced.

4-1 Training

The neural network model has unknown parameters, often called *weights*, and we seek values for them that make the model fit the training data well. We will discuss a more general case of a neural network in which we have L layers. We denote the complete set of weights by θ, which consists of

$$
\begin{aligned}
W_m^{(1)} &= \{W_{m0}^{(1)}, ..., W_{mP}^{(1)}\}, m \in \{1, ..., M\} \\
&\vdots \\
W_k^{(L)} &= \{W_{k0}^{(L)}, ..., W_{kM}^{(L)}\}, k \in \{1, ..., K\}
\end{aligned}
\tag{4-1}
$$

A simple approach to the problem of determining the network parameters is to minimize a sum-of-squares loss function. Given a training set of feature vectors $\mathbf{x}_i$ where $i \in \{1, ..., N\}$, together with a corresponding set of target variables y_i, we minimize the loss function

$$
L(\theta) = \frac{1}{2} \sum_{i=1}^{N} (\text{NeuralNet}_\theta(\mathbf{x}_i) - y_i)^2
\tag{4-2}
$$

where $q(\cdot)$ is our model and $\theta = \{W^{(1)}, ..., W^{(L)}\}$ a set weights.

A more general principle for estimation is *maximum likelihood estimation*. We will see that in a certain setting the maximum likelihood loss function is actually the same as a sum-of-squares loss function. Suppose we have a random sample Y with observed value y_i with $i \in \{1, ..., N\}$ from a distribution $Pr_\theta(Y)$ indexed by some parameters θ. The log-probability of the observed sample is

$$
L(\theta) = \sum_{i=1}^{N} \log Pr_\theta(Y = y_i)
\tag{4-3}
$$

The principle of maximum likelihood assumes that the most reasonable values for θ are those for which the probability of the observed sample is largest.

4-1-1   Regression

We consider the case $K = 1$. Suppose, again, that our data arose from the statistical model $Y = \mu_\theta(X) + \epsilon$, with $\epsilon \sim N(0, \sigma^2)$, then we get the following conditional likelihood

$$Y|X \sim N(\mu_\theta(X), \sigma^2) \tag{4-4}$$

The log-likelihood of the data is

$$L(\theta) = -\frac{N}{2}\log(2\pi) - N\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - \mu_\theta(\mathbf{x}_i))^2 \tag{4-5}$$

Although the additional assumption of normality seems restrictive, the results are the same as sum-of-squares. The only term in (7-10) involving θ is the last, which is the sum-of-squares (4-2) up to a scalar negative multiplier.

4-1-2   Classification

Models for classification problems can also be fit by maximum likelihood estimation, using conditional likelihood of Y given X. We denote $q_{\theta,k}(\mathbf{x}_i) = Pr(Y = k|X = \mathbf{x}_i)$. Since $Pr(Y|X)$ completely specifies the conditional distribution, the multinomial distribution is appropriate. We first consider the general case of K classes, where $k \in \{0, .., K-1\}$. The log-likelihood (also called cross-entropy) of the data is

$$L(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \ \log \ q_{\theta,k}(\mathbf{x}_i) \tag{4-6}$$

Now, we turn to the case of $K = 2$. It is convenient code the two class k via a 0/1 target variable Y. Let $Pr(Y = 1|X = \mathbf{x}) = q_{\theta,1}(\mathbf{x})$ and $Pr(Y = 0|X = x) = 1 - q_{\theta,1}(\mathbf{x})$. Now the loss function becomes

$$L(\theta) = -\sum_{i=1}^{N} y_{i1}\log q_{\theta,1}(\mathbf{x}_i) - (1 - y_{i1})\log(1 - q_{\theta,1}(\mathbf{x}_i)) \tag{4-7}$$

4-1-3   Log-likelihood for Neural Networks

The log-likelihood based loss is standard for Neural Networks (Goodfellow et al., 2016). Therefore, we shall construct the log-likelihood for a Neural Network with 1 hidden layer. We consider the case $K = 1$. Suppose, again, that our data arose from the statistical model $Y = \mu_\theta(X) + \epsilon$, with $\epsilon \sim N(0, \sigma^2)$. In this case the Neural Networks model is used

$$\mu_\theta(X) = \text{NeuralNet}_\theta(X) \tag{4-8}$$

The 1 hidden layer Neural Network is visualized in (4-1) and has the following form

$$\text{NeuralNet}_\theta(X) = \sum_{j=1}^{M} W_j^{(2)}\sigma\left(\sum_{i=1}^{P} W_{ij}^{(1)}X_i\right) \tag{4-9}$$

BagDrop

Therefore, the log-likelihood will for this Neural Network has the form

$$L(\theta) = -\frac{N}{2}\log(2\pi) - N\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y_i - \sum_{j=1}^{M}W_j^{(2)}\sigma\left(\sum_{i=1}^{P}W_{ij}^{(1)}\mathbf{x}_i\right)\right)^2 \tag{4-10}$$

Figure 4-1: Diagram of a 1 hidden layer Neural Network with 1 output
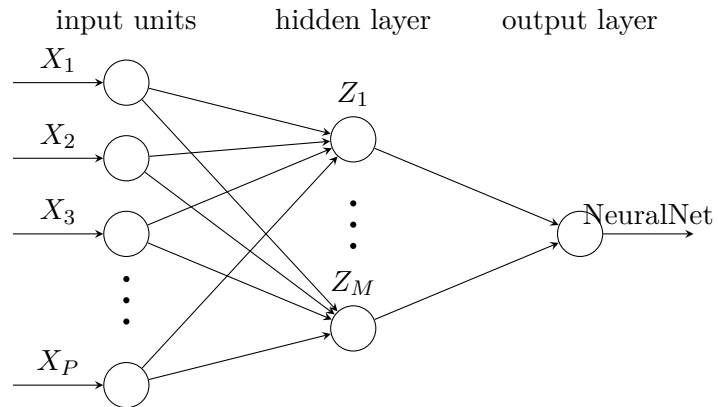
4-2   Parameter Optimization

The non-linearity function of the network function $\text{NeuralNet}_\theta(X)$ causes the loss function $L(\theta)$ to be non-convex. So we turn next to the task of finding a weight vector θ which minimizes the loss function. Our goal is to find a vector θ such that $L(\theta)$ takes its smallest value. However, because the network function is non-linear, we cannot always simply solve $\nabla L(\theta) = 0$ analytically to find a stationary point. Because the loss function has a highly nonlinear dependence on the weights and bias parameters, there will be many points in weight space at which the gradient (almost) vanishes. So it will be necessary to compare several local minima to get a good result. We clearly cannot find an analytical solution of $\nabla L(\theta) = 0$, therefore we call upon iterative numerical algorithms.

These algorithms are all based on the following update rule for the weight vector θ

$$\theta^{(\tau+1)} = \theta^{(\tau)} + \lambda \Delta \theta^{(\tau)} \tag{4-11}$$

where we choose some initial value $\theta^{(0)}$, τ label the iteration step, λ is the learning rate, discussed below and the weight vector update $\Delta \theta^{(\tau)}$ depends on the chosen algorithm. An example of determining the weight vector update is the use of gradient information.

4-2-1   Gradient Descent

The simplest approach to using gradient information is to use a so-called *batch* method, in which we use the whole data set at once, where we take a step in the direction of the negative gradient, such that

$$\theta^{\tau+1} = \theta^\tau - \lambda \nabla L(\theta^{(\tau)}) \tag{4-12}$$

where the parameter $\lambda > 0$ is known as the *learning rate*. At each step the weight vector is moved in the direction of the greatest rate of decrease of the loss function. This method is known as *gradient descent*. We can also make an update to the weight vector based on a sample (or a *mini-batch* of samples) from the training sample at the time. This is known as *stochastic gradient descent*. The derivation of stochastic gradient descent can be found in appendix (D-1). The approximation of the gradient of the loss is denoted by $\theta^{(\tau+1)}$, so that

$$\theta^{(\tau+1)} = \theta^{(\tau)} - \lambda \nabla_\theta \hat{L}(\theta) \tag{4-13}$$

A training epoch refers to one sweep through the entire training set. The learning rate λ for batch methods is usually taken to be a constant but can also be linearly annealed over the training epochs. This means that with every epoch, the learning rate is decreased by a certain value.
Furthermore, there exist many variations to Stochastic Gradient Descent, one of which is Adam (Adaptive Moment Estimation) (Kingma and Ba, 2015). The authors show empirically that Adam works well in practice and compares favorably to other Stochastic Gradient Descent based algorithms.

Generalization Performance & Overfitting

This chapter has two parts. The first part introduces the concept of overfitting and the second part will explain techniques that prevent overfitting. First we will define generalization (performance) and the bias-variance trade-off. Afterwards, overfitting with Neural Networks is explained and why Neural Networks are sensitive to overfitting. We will define regularization and why we should use regularization techniques in Neural Networks. Afterwards, we will give regularization techniques that are important for our thesis: Bagging and Dropout.

5-1 Generalization

Generalization is a measure of how accurately an algorithm is able to predict outcomes for future unseen data. Because learning algorithms are evaluated on finite samples, the evaluation of a learning algorithm may be sensitive to sampling error. Overfitting occurs when the learned function becomes too sensitive to the noise in the sample. As a result, the learned function $M_\theta(\cdot)$ performs well on the training data but does not perform well on unseen data. Thus to avoid bad generalization performance, we have to reduce overfitting.

Before we jump into overfitting, we have to define generalization more explicitly. First, consider we have a target variable Y and a vector of inputs X. We fit the model $M_\theta(X)$ with a training sample S and obtain our predictor $M_{\theta|S}(X)$. For neural networks in particular, this is explained in the previous sections. We also defined the loss function $L(\theta)$ and we showed how to minimize it by optimizing the parameters θ. In the following, we will look at the loss $L(Y, M_{\theta|S}(X))$, where we focus on the measure of the difference between Y and $M_{\theta|S}(X)$ instead of the parameters θ being optimized. The test error, also referred to as the *generalization error*, is the prediction error over an independent test sample (X, Y), where X and Y are drawn randomly from their joint distribution $Pr^*(X, Y)$, and *not* from the training sample S. The generalization error of some input point X is

$$Err_S(X) = \mathbb{E}\big(L(Y, M_{\theta|S}(X))\big) \tag{5-1}$$

More about the generalization error can be found in appendix (D-2).

5-2   Overfitting of Neural Networks

A neural network model has the advantage of being very flexible. The flexibility strongly relies on the number of hidden layers L or hidden units M. This way it can learn the general patterns of the data very well. One big disadvantage of a neural network model is that it may also learn the characteristics of each sample's unique noise. The neural networks model is generally overparametrized, and the optimization problem is nonconvex and unstable unless certain guidelines are followed. *Regularization* is the usage of methods that prevent overfitting.

Furthermore, we might expect that in a maximum likelihood setting there will be an optimum value of M that gives the best generalization performance, corresponding to the optimum balance between under-fitting and over-fitting. This can be seen in an **example**. Consider a regression problem in which we want to predict a sine-function from a training sample. The simulation of the training- and test-data is as follows

$$
\begin{aligned}
(X, Y) &\sim \left(R, sin\left(\frac{2\pi R}{4}\right) + \epsilon\right) \\
R &\sim \text{Uniform}(0, 4) \\
\epsilon &\sim N(0, 0.05)
\end{aligned}
\tag{5-2}
$$

We sampled 10 times to obtain the training-data and 400 times to obtain the test-data. We used a Neural Network with an architecture of 3 hidden layers and number of hidden units per layer ranging from 1 to 15. The used training epochs were 5000, 10000, 20000. The log-likelihood with respect to the test-data on these models are displayed in figure (5-1).

As the figure indicates, there is a trade-off between the model complexities. The optimal number of hidden units per layer is between 5 and 7. This means that the optimal model holds between 60 and 121 weights that need to be optimized. A lower number of hidden units M can result in bad log-likelihoods, but increasing the number of hidden units too much can result in worse log-likelihoods. The goal of getting a model that is sufficiently flexible to capture the particular characteristics of the data is at odds with the goal of finding a function that does not overfit. This is known as the bias-variance trade-off. The bias-variance decomposition can be found in appendix (D-3). The generalization error, however, is not a simple function of M due to the presence of local minima in the loss function. Furthermore, finding the optimal number of hidden units is a very time consuming task. So it is most common to put down a *reasonably large number of units* and train them with *regularization*. A standard regularization method is stopping the optimization algorithm when there is improvement for some amount of time. This is called *early stopping* and a description can be found in appendix (D-4). However, we are researching Bagging and Dropout as regularization methods. These are explained in the following sections.
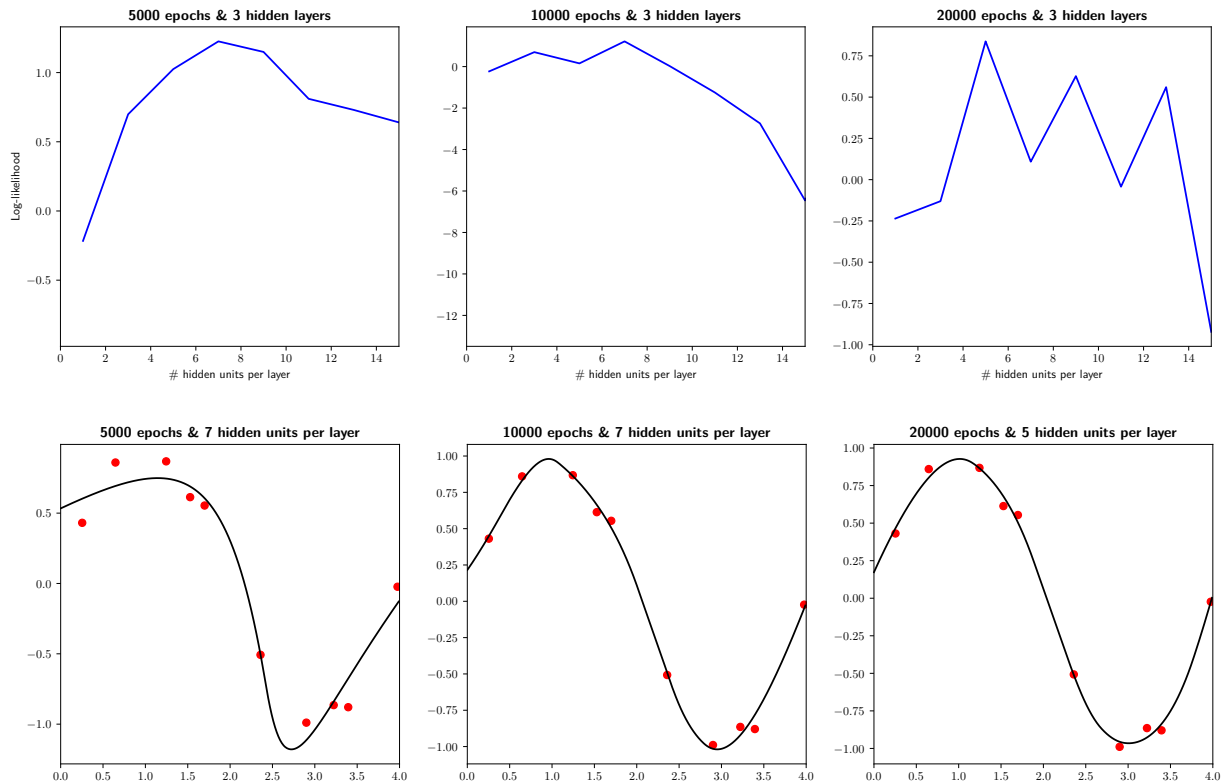
Figure 5-1: First row of graphs: example of the loglikelihoods of 3 hidden layer Neural Networks trained on 10 data points drawn from a sinusoidal data set (described by equation (5-2)). The log-likelihoods are with respect the test data. The test data was drawn from the same sinusoidal data set. The number of hidden units per layer of the models range from 1 to 15. The graphs show the log-likelihoods versus the number of hidden units per layer, for 5000 epochs (left), 10000 epochs (middle) and 20000 epochs (right). Between 5 and 7 hidden units per layer holds the optimal model *for this problem*. We can see a trade-off between the model complexities. **Second row of graphs**: example of the obtained predictors, trained with 5000 epochs (left), 10000 epochs (middle) and 20000 epochs (right). The obtained Neural Networks hold the optimal number of hidden units per layer *for this problem*, which is 7 (left), 7 (middle) and 5 (right) respectively. We can clearly see the predictor improve to an acceptable sine-function as the number of epochs increase. However, searching for the optimal number of hidden units is a time consuming task.

5-3 Bagging

An approach to prevent overfitting is via bootstrap aggregation, which averages the predictions of multiple networks which are trained by randomly perturbed versions of the training data. Bagging creates and combines all these multiple networks.

5-3-1 What is Bagging and how do we apply it?

If we want to know how Bagging works, we first have to know how we create bootstrap samples. We will first describe bootstrap sampling and afterwards we will explain what Bagging is and how to apply Bagging.

The Bootstrap

Bootstrapping (Efron and Tibshirani, 1994) is a general approach to statistical inference based on building a sampling distribution for a statistic by resampling from the data at hand. In the context of neural networks, the deterministic algorithm of obtaining weights from a training sample S can be seen as a statistic.

The combination of a feature vector X along with a corresponding target value Y is called an observed tuple (X, Y). Now consider a random training sample $S = \{(X_i, Y_i)\}$ with $i \in \{1, ..., N\}$ as independent random variables with joint distribution $Pr^*(X, Y)$.

Now suppose that we are interested in some statistic $T = t(S)$ as an estimate of the corresponding population parameter γ. It is important to keep in mind that S is random, therefore $t(S)$ is random, hence $t(S)$ has a (sampling) distribution. The sampling distribution is determined by N and $Pr^*(X, Y)$. We would like to know this sampling distribution, but we are faced with two problems:

1. We don't know $Pr^*(X, Y)$.

2. Even if we knew $Pr^*(X, Y)$, finding the distribution of T explicitly would exceed our analytic capabilities.

We will first look at the second problem. If we now suppose that we knew $Pr^*(X, Y)$. We can skip the part where we have to go through incredibly complicated analytic calculations and approximate the probability distribution of T through simulation. We generate many samples, say B, of size N from $Pr^*(X, Y)$, and with each sample we calculate the value of T. Now, the empirical distribution of the resulting $T^{(1)}, .., T^{(B)}$ is an approximation to the distribution of T. If we take B larger, these approximations will be more accurate.

This is all possible under the condition that we know $Pr^*(X, Y)$, but we don't. So what do we do next? The essential idea of nonparametric bootstrap is as follows. We denote $Pr_N^*(X, Y)$ as the empirical distribution created by the elements of S. We view $Pr_N^*(X, Y)$ as an approximation to $Pr^*(X, Y)$. That is, $Pr_N^*(X, Y)$ would be used in place of $Pr^*(X, Y)$ in the previous paragraph. How do we go about sampling from $Pr_N^*(X, Y)$? $Pr_N^*(X, Y)$ is a discrete probability distribution that gives probability $1/N$ to each element of S. We proceed to draw a sample of size N from the elements in S. Call the resulting bootstrap sample $S^{(1)} = \{(X_i, Y_i)^{(1)}\}$ with $i \in \{1, ..., N\}$. It is necessary to sample with replacement, because we would otherwise simply reproduce the original sample S. Now we can select B bootstrap samples and we denote the b-th bootstrap sample with $S^{(b)} = \{(X_i, Y_i)^{(b)}\}$. Next, we compute the statistic T for each of the bootstrap samples; that is $T^{(b)} = t(S^{(b)})$ for all $b \in \{1, ..., B\}$. The whole process of bootstrap sampling is visualized in figure 5-2.

Bootstrap sampling
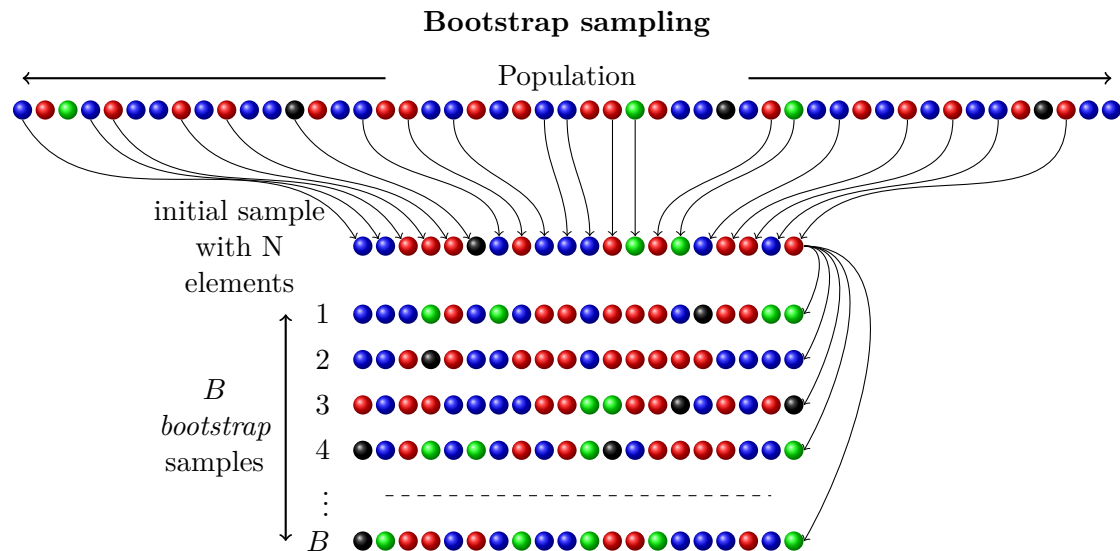
Figure 5-2: A schematic illustration of bootstrap sampling. Suppose that our training sample is sampled from a population. A bootstrap sample is obtained by sampling from the training sample. We repeat bootstrap sampling until we get B bootstrap samples. Every dot represents a single datapoint (X, Y). The first row of dots represent the population and the second row represent the training sample. The third to seventh row represent the bootstrap samples.

Bagging

Bagging (Breiman, 1996) is short for "Bootstrap AGGregatING". With making predictions from the data, we fit one single model to the data. In Bagging, instead of making predictions from a single fit of the data, we take bootstrap samples of the data and fit the model to each sample. Afterwards, the predictions are averaged over all the fitted models to get the bagged prediction.

Neural networks are very flexible models and predictions of neural networks typically have low bias but high variance. How do we reduce the variance of predictions? Suppose again that we have a training sample $S = \{(X_i, Y_i)\}$ with $i \in \{1, ..., N\}$ as independent random variables with joint distribution $Pr^*(X, Y)$. Suppose that the statistic is in fact

$$t(S) = \text{NeuralNet}_{\theta|S}(\mathbf{x}) \tag{5-3}$$

where $\theta|S$ denotes the weights optimized based on training sample S and $\mathbf{x}$ a fixed observed value.

The bootstrap aggregation procedure is as follows. Take B repeated bootstrap samples $\{S^{(b)}\}$ with $b \in \{1, .., B\}$ from the training sample S and form the learned function from this bootstrap samples $\text{NeuralNet}_{\theta|S^{(b)}}(\mathbf{x})$. The bootstrap samples are drawn from the empirical distribution $Pr_N^*(X, Y)$, which gives probability $1/N$ to each element of S. The bagged estimate is defined by

$$\overline{\text{NeuralNet}_\theta(\mathbf{x})} = \frac{1}{B} \sum_{b=1}^{B} \text{NeuralNet}_{\theta|S^{(b)}}(\mathbf{x}) \tag{5-4}$$

This expression is a Monte Carlo estimate of the true Bagging estimate, approaching it as $B \to \infty$.

BagDrop

5-3-2   Why could Bagging work?

Bagging can reduce the variance of unstable procedures (Bauer and Kohavi, 1999) leading to improved prediction. A simple argument shows why Bagging helps under squared-error loss. Assume that we have a regression problem with continuous target variable Y. Furthermore, assume that

$$Y|X \sim N(\mu_\theta(X), \sigma^2) \tag{5-5}$$

Consider the ideal aggregate estimator

$$\overline{\mu_{\theta|S}(\mathbf{x})} = \mathbb{E}(\mu_{\theta|S}(\mathbf{x})) \tag{5-6}$$

where $S = \{(X_i, Y_i)\}$ with $i \in \{1, .., N\}$ is a random sample from the joint distribution $Pr^*(X, Y)$ and $\mathbf{x}$ a fixed observed value. Note that $\overline{\mu_{\theta|S}(\mathbf{x})}$ is a Bagging estimate, drawing bootstrap samples from the actual population rather than the training sample.

Take a y, just like $\mathbf{x}$, to be a fixed observed output value. Then

$$
\begin{aligned}
\mathbb{E}(y - \mu_{\theta|S}(\mathbf{x}))^2 &= y^2 - 2y\mathbb{E}(\mu_{\theta|S}(\mathbf{x})) + \mathbb{E}(\mu_{\theta|S}(\mathbf{x})^2) \\
&\geq y^2 - 2y\mathbb{E}(\mu_{\theta|S}(\mathbf{x})) + \mathbb{E}(\mu_{\theta|S}(\mathbf{x}))^2 \\
&= y^2 - 2y\,\overline{\mu_{\theta|S}(\mathbf{x})} + \overline{\mu_{\theta|S}(\mathbf{x})}^2 \\
&= (y - \overline{\mu_{\theta|S}(\mathbf{x})})^2
\end{aligned}
\tag{5-7}
$$

where the inequality $\mathbb{E}(X) \geq (\mathbb{E}(X))^2$ is used. Therefore, true population aggregation never increases mean squared error. How much depends on the difference

$$\mathbb{E}(\mu_{\theta|S}(\mathbf{x}))^2 - \mathbb{E}(\mu_{\theta|S}(\mathbf{x})^2) \tag{5-8}$$

The more more $\mu_{\theta|S}(\mathbf{x})$ varies with sample S, the more improvement aggregation may produce. Therefore, Bagging can only give improvement to unstable procedures, like neural networks.

Furthermore, more recent empirical evidence (Grandvalet, 2004) suggests that Bagging equalizes training points $(X, Y) \in S$ in such a way that highly influential points are down-weighted. Leverage points are isolated in the feature space. Leverage points can be part of the undesirable noise in the training data we talked about in section 5-1. In much procedures, such as neural networks, leverage points are badly influential. Neural Networks have the potential to overfit on these points.

Since in bootstrap sampling all the elements of the training sample have the same probability $1/N$ to be selected, it seems paradoxical that Bagging has the ability of reducing the influence of specific training points. The following explanation gives an intuitive insight. Leverage points are isolated in the feature space while others are located in more dense regions of the input space. To remove the influence of a leverage point it is enough to just remove that specific point from the training sample. But when we want to remove the influence of a training point located in a more dense region in the feature space, we must in general remove a group of training points. The probability that a group of training points $\{(X_1, Y_1), .., (X_k, Y_k)\} \in S$ is ignored by a bootstrap sample $S^{(b)}$ is

$$Pr(\{(X_1, Y_1), .., (X_k, Y_k)\} \notin S^{(b)}) = (1 - k/N)^N \tag{5-9}$$

where N is the size of the training sample S. This probability decays with increasing group size k. Therefore, the probability that influence of isolated training points is removed is larger than the probability that influence of a large group of training points is removed.

There is also a paper (Poggio et al., 2002) that tries to derive certain stability statements about Bagging when using special resampling schemes. Poggio et al. studied the relationship between stability and Bagging. They showed that there is a Bagging scheme, where each expert is trained on a disjoint subset of the training data, providing strong stability to ensembles of non-strongly stable experts, and therefore providing the same order of convergence for the generalization error as Tikhonov regularization. Thus, at least asymptotically, Bagging strongly stable experts would not improve generalization ability of the individual member, since regularization would provide the exact same effect.

5-4   Dropout

Dropout (Srivastava et al., 2014) is a relatively new tool that prevents overfitting and provides a way of combining many different neural network architectures. The technique describes a way of dropping out hidden units during training, hence the name "Dropout". In this context, dropping out means temporarily removing units from the network, along with the incoming and outgoing connections. To understand why Dropout could be favorable to solve the problem of computational cost of Bagging, we need to know what Dropout is and how to apply it.

5-4-1   What is Dropout?

The procedure is as follows. The units have a probability p of being dropped out and p is called the *Dropout rate*. Srivastava et al. investigate the choice of the Dropout rate and come to the conclusion that a Dropout rate of $p = 0.5$ is empirically the best choice in their experiments. However, this does not necessarily apply to all supervised learning problems with a Neural Network as model. But, the choice of $p = 0.5$ seems to be an intuitive choice when we think in terms of the Dropout masks $p = 0.5$ produces. When we have to deal with a relatively small amount of features, choosing the Dropout rate too small will possibly result in removing all the feature variables. Moreover, if we drop out the output units, we will lose the output.

Figure 5-3: Neural Network where the layers are fully connected
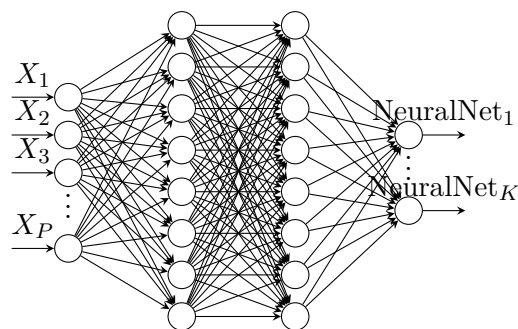
Figure 5-4: After applying Dropout
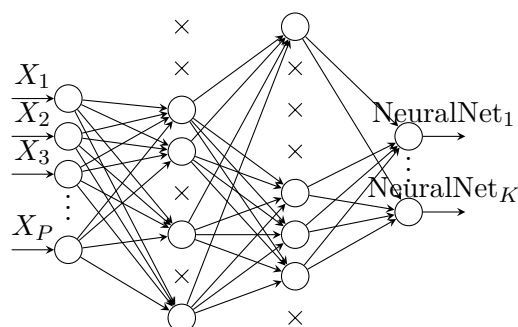
Assume that our network consists of a total of H hidden units. We now interpret the neural network as a collection of 2^H possible "thinned networks", called *droput masks*. In stochastic gradient descent, we choose a mini-batch of training data, then randomly choose one mask and take a step in the direction of the negative gradient with respect to the chosen mask. Then training a neural network with Dropout can be seen as training a collection of 2^H networks with extensive weight sharing, where each network in the collection gets trained very rarely, if at all. At test time however, the idea of Srivastava et al. is to use a single neural network without Dropout. The weights of this single network are scaled-down versions of the trained weights. If

an unit has a probability of p to be dropped out during training, the outgoing weights of that unit are multiplied by p if testing the neural network on a test set. This ensures that for any hidden unit the expected outpout is the same as the actual output at test time. If we apply this scaling, it is possible to get one single network out of the 2^H networks with shared weights.

5-4-2 How do we apply Dropout?

The formal description of Dropout on a neural network is as follows. Consider a neural network with L layers. Now let $l \in \{0, ..., L-1\}$. Let $\mathbf{a}^{(l)}$ be the vector of inputs into layers into layer l, $\mathbf{z}^{(l)}$ denote the vector of outputs from layer l. Then $\mathbf{z}^{(0)} = \mathbf{x}$ denotes the feature vector. Furthermore, $W^{(l)}$ denotes the weights and biases at layer l. Now, the standard neural network model description is

$$
\begin{aligned}
a_i^{l+1} &= \langle \theta_i^{(l+1)}, \mathbf{z}^{(l)} \rangle \\
z_i^{(l+1)} &= h\left(a_i^{(l+1)}\right)
\end{aligned}
\tag{5-10}
$$

where $h(\cdot)$ is the activation function, $l \in \{0, ..., L-1\}$ and hidden units indexed by i. Now with Dropout, the description is as follows

$$
\begin{aligned}
r_j^l &\sim \text{Bernoulli}(p) \\
\hat{\mathbf{z}}^{(l)} &= \langle \mathbf{r}^{(l)}, \mathbf{z}^{(l)} \rangle \\
a_i^{(l+1)} &= \langle \theta_i^{(l+1)}, \hat{\mathbf{z}}^l \rangle \\
y_i^{(l+1)} &= h\left(a_i^{(l+1)}\right)
\end{aligned}
\tag{5-11}
$$

For any layer l, $\mathbf{r}^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability p of being 1.

Now we can train Dropout neural networks using stochastic gradient descent. We can use Dropout in several ways. Srivastava et al. describes a way in which mini-batches are used. For every training case in the mini-batch, he creates a Dropout mask by dropping out units. Forward and backpropagation for that training case are done only on this Dropout mask. The gradients for each parameter are accumulated over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient zero for that parameter.

5-4-3 Why could Dropout help to solve our problem?

Dropout gives a very simple way of combining a collection of models. Combining several models is most helpful when the individual models are different from each other and in order to make neural networks different, they should either have different architectures or be trained on different data. We described the latter in sections (5-3-1) and (5-3-1). But we came to the conclusion that training several models is very time consuming. Dropout can solve that problem.

Moreover, Dropout also addresses the issue of training several *different* architectures jointly. If we want to use different architectures for every bootstrap sample in Bagging, we have to find to optimal learning rate, batch size, training epochs etc. for *every* architecture, which is very time consuming. Therefore, the clever way of training different architectures in Dropout can be seen as an additional advantage.

Research Question 1: BagDrop

This chapter describes a possible answer to the first research question. We propose the new BagDrop procedure and argue why it could solve the computational cost problem of Bagging.

6-1 BagDrop

Bagging is a very simple technique that can reduce variance in predictions in Neural Networks. However, Bagging has the major disadvantage of computational cost. This cost is linear in the number of models used, and the results get better when we use more models. The computational cost may stop people from using it in practice when using Neural Networks as base model. Furthermore, Dropout is a relatively new tool that combines several Neural Network architectures in a very fast and clever way. This brings us back to the first research question:

> **Research Question 1**
>
> How can we apply Bagging in a Dropout way, such that we retain the speed of Dropout, while still having similar generalization properties as Bagging?

We have to come up with a better idea than traditional Bagging. What should we do next? We propose **BagDrop** as answer to the first research question. Bagdrop provides a way of combining the methods of Bagging and Dropout, that hopefully has similar generalization performance as Bagging and more or less the same computational cost as Dropout. In BagDrop, we sample a Dropout mask for every bootstrap sample. Every Dropout mask is trained by its own bootstrap sample and combined as Dropout prescribes. But how do we apply BagDrop? This is explained in the following section.

6-1-1 How do we apply BagDrop?

The BagDrop procedure is as follows. Assume that we have a training sample $S = \{Z_i\}$ with $i \in \{1, ..., N\}$ as independent random variables and $Z_i = (X_i, Y_i)$. Take B repeated bootstrap samples $\{S^{(b)}\}$ with $b \in \{1, .., B\}$ from the training sample S. Now, for every bootstrap sample, sample a Dropout mask $M^{(b)}$ for the hidden units. We call the combination of one Dropout mask and one bootstrap sample a model tuple and denote it by $(S^{(b)}, M^{(b)})$.

Now, we have to derive a way to apply stochastic gradient descent. We have to minimize the expected loss on all data and all Dropout masks. To approximate the gradient of the expected loss we can use Monte Carlo methods. Assume that there exists some sort of distribution

underlying X, Y, M. If we want to take K independent samples from this distribution we have to keep in mind that the training samples are dependent on the Dropout masks $X, Y|M$. Therefore, first sample a Dropout mask $m \sim M$ from the ensemble and afterwards, we sample a training sample $\mathbf{x}, y \sim X, Y|M$ from the corresponding bootstrap sample. Then we get a set of samples $\{\mathbf{x}^{(k)}, y^{(k)}, m^{(k)}\}$ with $k \in \{1, ..., K\}$. We can approximate the gradient of the loss with

$$\nabla_\theta \hat{L}(\theta) = \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta L(\theta, \mathbf{x}^{(k)}, y^{(k)}, m^{(k)}) \tag{6-1}$$

Once we have above value, we can take a step towards the negative gradient

$$\theta^{(\tau+1)} = \theta^{(\tau)} - \lambda \nabla_\theta \hat{L}(\theta) \tag{6-2}$$

Repeat this process until one reaches a certain desirable value of the weights θ.

6-1-2   Why could BagDrop solve our problem?

Just as in the procedure of Dropout, every mask defines a certain "thinned network". Every training step, a batch of masks is trained jointly, with each mask having it's own corresponding bootstrap sample. This way, it is possible to combine several neural network architectures (or "thinned networks") and train them jointly. From a Bagging perspective, with BagDrop, we can apply a special case of Bagging in a much faster way than traditional Bagging. The computational cost of BagDrop stays approximately the same with the amount of bootstrap samples. In comparison, the computational cost of traditional Bagging grows linear with the amount of bootstrap samples.

Research Question 2 & 3: Generalization Performance & Computational Cost

In this chapter we will investigate the second and third research question. To answer the second and third research question, we will conduct an empirical research. The research consists of two elements. First, we will conduct a proof-of-concept of BagDrop with a toy-dataset. The explanation of the method, experiments and analysis of the **proof-of-concept** are very comprehensive. We do this to give the reader an elaborate understanding of the experiments conducted.

7-1 Proof-of-Concept

7-1-1 Method

Before we start experimentation, we need to have a good idea what it is we are studying, how the data is collected, and how we plan to analyze it. We will look at a problem in which overfitting occurs. We described Bagging in chapter 5 as possible regularization technique. However, Bagging has the major disadvantage of computational cost. We described BagDrop in chapter (6) as possible candidate to solve this problem. But, does Bagdrop perform similar as Bagging in terms of generalization performance? And how does BagDrop perform in terms of computational cost? This brings us back to the second and third research question:

> **Research Question 2**
>
> How does the BagDrop procedure perform in comparison to Bagging, in terms of generalization performance?

> **Research Question 3**
>
> How does the BagDrop procedure perform in comparison to Dropout, in terms of computational cost?

How do we answer these research questions? We need to construct a measure of generalization performance and a measure of computational cost. But, before we do that, we have to explain how the *data* is collected and how our *model* is defined. We will explain the data and our

model. One may notice that we used the same data and model as in the example in section (5-2) and figure (5-1). However, we made two small adjustments; number of epochs and size of the network. The data, model and adjustments are clarified in the following two sections. Afterwards, we explain our measures of generalization performance and computational cost. Finally, we will give our experimental design and explain how we will analyze the results of the experiments.

Data: Toy-dataset

With limited computational capability, it was initially unpractical to look at classification and regression problems from large datasets. So we began with simulating sine data to form a very small dataset. The following is *the same dataset* we used in the example in section (5-2). We sampled from

$$
\begin{aligned}
(X, Y) &\sim \left(R, sin\left(\frac{2\pi R}{4}\right) + \epsilon\right) \\
R &\sim \text{Uniform}(0, 4) \\
\epsilon &\sim N(0, 0.05)
\end{aligned}
\tag{7-1}
$$

to obtain the observed training sample $D = \{(x_1, y_1), .., (x_{10}, y_{10})\}$. Larger datasets would be unpractical due to the computational cost of Bagging. Then, we sampled a test sample of size 400 from the same distribution to obtain $T = \{(x_1, y_1), .., (x_{400}, y_{400})\}$. This resulted in the following graph (figure (7-1)) of the observed training and test data.
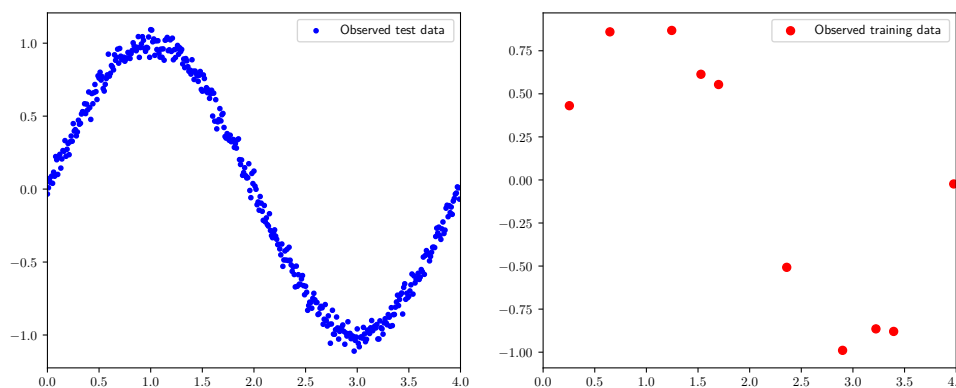
Figure 7-1: Observed test data (left) and observed training data (right) drawn from sinusoidal data described by the equation (7-1).

Model & Hyperparameters

We will use a full probabilistic setting. Suppose that our data arose from the statistical model

$$
Y|X \sim N(\mu_\theta(X), \sigma^2)
\tag{7-2}
$$

We are specifically interested in the mean $\mu_\theta(X)$ and the variance σ^2. We will use a neural network as described in (3-3) to predict the mean

BagDrop

$$\mu_\theta(X) = \text{NeuralNet}_\theta(X) \tag{7-3}$$

The log likelihood of the data is

$$L(\theta, \sigma^2) = -\frac{N}{2}\log(2\pi) - N\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - \text{NeuralNet}_\theta(\mathbf{x}_i))^2 \tag{7-4}$$

We want to maximize this log-likelihood in order to get the best fit of our parameters to the data. Therefore, log-likelihood (7-4) is defined as our loss fucntion. Note that, besides the network weights θ, the standard deviation σ is also modelled as an optimization parameter. The standard-deviation must satisfy $\sigma \geq 0$ and can therefore be represented in terms of exponentials during training.

$$\sigma = \exp(\alpha) \tag{7-5}$$

We will use stochastic gradient descent as optimization algorithm as described in section (4-2-1). Particularly, we will use *Adam* (Kingma and Ba, 2015). We shortly mentioned early stopping as standard regularization technique in section (5-2). However, we first want to investigate BagDrop isolated from other regularization methods. Therefore, we will be researching BagDrop *without early stopping*. But, because early stopping is standard for Neural Networks, the results and analysis of the experiments with early stopping can be found in appendix (B-2) and (C-2-3) respectively.

Until now, all the properties of our model were not dependent upon the training data. With our training data, we wish to fit the *parameters* of our model. However, there is another kind of parameters that cannot be directly learned from the regular training process. These parameters express "higher-level" properties of our model complexity or how fast it should learn. They are called **hyperparameters**. Hyperparameters are determined and fixed before the actual training process begins. In our case, the hyperparameters, with corresponding section containing background theory, are

- Number of hidden layers (section (3-3))

- Number of hidden units per layer (section (3-3))

- The activation function (section (3-3))

- The final activation function (section (3-3))

- The learning rate (section (4-2-1))

- Number of training epochs (section (4-2-1))

- Batch size (section (4-2-1))

How do we choose our hyperparameters? Choosing the final activation function is straightforward. We are considering a regression problem with 1 output, so the final activation function is the *identity*.

Furthermore, we are investigating regularization techniques, so we put down a reasonably large number of hidden units as we explained in (5-2). We chose a network with 3 hidden layers and 50 hidden units per layer. Furthermore, we used a linearly annealed learning rate as described in (4-2-1). After repeatedly re-configuring the activation function and the learning rates we

came to a learning rate that starts at $\lambda_{\text{start}} = 0.002$ and ends at $\lambda_{\text{end}} = 0.00001$ and the ELU activation function as described in section (3-3). Finally, we chose a batch size of 10 and 50000 training epochs to make sure we do not underfit.

The hyperparameters clearly produce an overfitted predictor. the hyperparameters are summarized in table (7-1) and the predictions are displayed in figure (7-2).

Table 7-1: Hyperparameters

Hyperparameter	Selected
Hidden layers (L)	3
Hidden units per layer	50
Activation function	ELU
Learning rate start (λ_{start})	0.002
Learning rate end (λ_{end})	0.00001
Training epochs	50000
Batch size	10
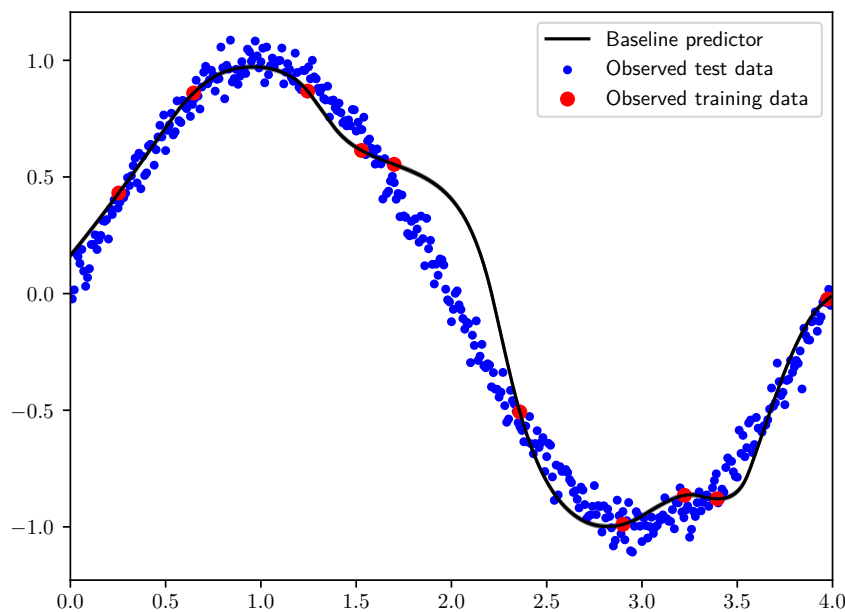
Figure 7-2: Predictions of overfitted predictor based on the hyperparameters described in table (7-1)

Research Question 1: Measure of generalization performance

How do we measure generalization performance? This can be done in different ways. After training, we obtain trained parameters $\hat{\theta}$ and $\hat{\sigma^2}$. The log-likelihood of the trained parameters with respect to the test data is a measure of generalization performance

$$L(\theta, \sigma^2) = -\frac{N}{2}\log(2\pi) - N\log\hat{\sigma} - \frac{1}{2\hat{\sigma^2}}\sum_{i=1}^{N}(y_i - \text{NeuralNet}_{\hat{\theta}}(\mathbf{x}_i))^2 \tag{7-6}$$

BagDrop

We also want a measure of the generalization performance of an ensemble of predictions. Therefore, assume an ensemble $(\{\text{NeuralNet}_{\hat{\theta}_1}(X), \hat{\sigma_1^2}), .., (\{\text{NeuralNet}_{\hat{\theta}_B}(X), \hat{\sigma_B^2})\}$ of B predicted means and variances. Every ensemble component $b \in \{1, .., B\}$ produces its own normal density function $N(\text{NeuralNet}_{\hat{\theta}_1}(X), \hat{\sigma_1^2})$. The average of the B density functions forms a Gaussian mixture density function

$$\frac{1}{B} \sum_{b=1}^{B} N(\text{NeuralNet}_{\hat{\theta}_b}(X), \hat{\sigma_b^2}) \tag{7-7}$$

If P is a random variable with this underlying Gaussian mixture density function, the mean and variance are

$$\mathbb{E}(P|X) = \frac{1}{B} \sum_{b=1}^{B} \text{NeuralNet}_{\hat{\theta}_b}(X)$$

$$\text{Var}(P|X) = \frac{1}{B} \sum_{b=1}^{B} \left(\left(\mathbb{E}(P|X) - \text{NeuralNet}_{\hat{\theta}_b}\right)^2 + \hat{\sigma_b^2} \right) \tag{7-8}$$

Note that the log-likelihood of the Gaussian mixture density with respect to the *training data* is

$$L(\theta, \sigma^2) = -\frac{1}{B} \sum_{b=1}^{B} \sum_{i=1}^{N} \log \left(N(\text{NeuralNet}_{\hat{\theta}_b}(X), \hat{\sigma_b^2}) \right) \tag{7-9}$$

The actual maximizer of the log-likelihood occurs when we put a spike of infinite height at any one data point, that is, $\text{NeuralNet}_{\hat{\theta}_b}(\mathbf{x}_i) = y_i$ for some i and $\sigma_b^2 = 0$. This gives infinite likelihood, but is not a very useful solution. Therefore, we could assume that the Gaussian mixture density is approximately a normal density with mean $\frac{1}{B} \sum_{b=1}^{B} \text{NeuralNet}_{\hat{\theta}_b}(X)$ and variance $\frac{1}{B} \sum_{b=1}^{B} \left(\left(\mathbb{E}(P|X) - \text{NeuralNet}_{\hat{\theta}_b}\right)^2 + \hat{\sigma_b^2} \right)$. Then, the log-likelihood of the trained parameters with respect to the test data is

$$L(\theta, \sigma^2) = -\frac{N}{2} \log(2\pi) - N \log \sqrt{\text{Var}(P|X = \mathbf{x})} - \frac{1}{2\text{Var}(P|X = \mathbf{x})} \sum_{i=1}^{N} (y_i - \mathbb{E}(P|X = \mathbf{x}))^2 \tag{7-10}$$

This will be our measure of generalization performance of an ensemble of size B.

Research Question 2: Measure of computational cost

How do we measure computational cost? Compared to Dropout, BagDrop requires some extra calculations througout the whole algorithm. The question is whether this will be a problem in practice. Therefore, we will have two measures; run-time of one experiment and run-time of one epoch (in the same experiment). This way we can differentiate between the time BagDrop needs in one epoch and the run-time of one whole experiment.

BagDrop

Experimental Design

We are comparing algorithms, therefore, the learning algorithm is the factor of interest. We have a dataset (described in section (7-1-1)) and model (described in section (7-1-1)) in which overfitting emerges as problem. We will call this problem the *baseline*. This will be our starting point. We will run the baseline, without additional technques, in the first set of experiments. Afterwards, we run the baseline with the additional regularization techniques *Bagging*, *Dropout* and *BagDrop* in the second, third and fourth sets of experiments respectively. Every algorithm also has its own hyperparameters. In the following list, the algorithms and hyperparameters are stated with corresponding background theory section.

- **Bagging:** We will apply the Bagging procedure as described in section (5-3-1). We will use an ensemble size of 20. Bigger ensemble sizes were unpractical due to computational limitations.

- **Dropout:** We will apply the traditional dropout procedure as described in section (5-4). We will use a Dropout rate of $p = 0.5$.

- **BagDrop:** We will apply the BagDrop procedure as described in section (6-1-1). We will use an ensemble size of 100 and a Dropout rate of $p = 0.5$

However, these algorithms may not be the only factors of interest. There is one algorithm that could have the same results in terms of generalization performance and computational cost as BagDrop. Namely, the BagDrop algorithm *without a bootstrap sample* for every Dropout mask. We will call this special version of the BagDrop algorithm *Quasi-Dropout*. We will run Quasi-Dropout in the fifth set of experiments. Furthermore, we also want to know whether a non-sophisticated combination of Bagging and Dropout has more or less the same generalization performance as BagDrop. This means that we apply Bagging and train the bootstrap samples one-by-one such as Bagging prescribes. Then, for every bootstrap sample, apply Dropout. Because we basically use Quasi-Dropout in BagDrop, we will use Bagging + Quasi-Dropout (instead of traditional Dropout) in the sixth set of experiments to get a good comparison. In the following list, the two additional algorithms are stated with corresponding background theory section.

- **Quasi-Dropout:** We will apply the Quasi-Dropout procedure as described in appendix (D-5). This procedure has the potential to have the nearly same effect as BagDrop. So it is important that the ensemble used in Quasi-Dropout is of the same size as in BagDrop. That is, an ensemble size of 100 and a Dropout rate of $p = 0.5$.

- **Bagging + Quasi-Dropout:** We will apply the Bagging procedure as described in section (5-3-1), and with every bootstrap sample, we will apply Quasi-Dropout as described in appendix (D-5). This gives a good comparison to the new procedure BagDrop. We used an ensemble size of 20 for Bagging and 100 for Quasi-Dropout. We used a Dropout rate of $p = 0.5$.

If we denote the log-likelihood of a method with c_{method}, then ideally, the log-likelihoods will be ordered in the following way

$$c_{baseline} < \ c_{Quasi-Dropout} < c_{Bagging} \approx c_{Bagging+Dropout} \approx c_{BagDrop} \qquad (7\text{-}11)$$

If we denote the run-time of each experiment and each epoch with t_{method}, then ideally, the run-times will follow the following equality

BagDrop

$$t_{Dropout} \approx t_{Quasi-Dropout} \approx t_{BagDrop} \tag{7-12}$$

For *every* method, we will replicate 20 experiments. Every experiment produces a log-likelihood as a measure of generalization performance. We derive a mean and standard-deviation of every experiment to get a summarized idea of the performance. Furthermore, every experiment produces a run-time of that particular experiment and a mean run-time of all epochs.

To determine whether the log-likelihoods of the algorithms are significantly different we will have to do some statistical analysis of the results. In the following section, we will explain how we will do this.

Analysis: Kruskal-Wallis Test and more

First, we will make a *boxplot* of the *log-likelihood results* and analyze this. Afterwards we will do a more sophisticated test. We are using a repeated-measures experimental design, so we can use the non-parametric *Kruskal-Wallis test* (Kruskal and Wallis, 1952) to compare the resulting log-likelihoods. The reason why we will not use other more common tests like the *t*-test (Student, 1908) is that these make very strong assumptions regadering the underlying distribution of the data. The Kruskal-Wallis test relies on ranks and is less demanding with respect to the implied data distribution conditions.

A full description of the Kruskal-Wallis test can be found in appendix (D-6). The Kruskal-Wallis test produces a *p*-value. The null hypothesis is denoted by H_0. In our analysis, we will reject H_0 with significance level $\alpha = 0.05$. This means, we will reject the hypothesis that the algorithms are the same if the *p*-value drops below $\alpha = 0.05$

When H_0 is rejected, we know that the log-likelihoods of the algorithms are significantly different. However, we still do not know which algorithm is better. Therefore, we should do a so called *post-hoc* test for comparing all pairs of algorithms. One of the possible tests is the Nemenyi test (Nemenyi, 1962). In the Nemenyi test, two algorithms differ significantly from eachother if the corresponding ranks differ at least

$$cd = q_a \sqrt{\frac{k(k+1)}{6E}} \tag{7-13}$$

where the critical values q_a are based on the Studentized range statistic divided by $\sqrt{2}$ and can be found in look-up tables (Nemenyi, 1962).

However, it may happen that the Kruskal-Wallis test results in the rejection of H_0, but the post-hoc test fails to find any significant pairwise difference. Then we can only make the general statement that there is a difference between some algorithms.

Finally, we will compare the results of the *run-times* without a significance test and state some striking features.

7-1-2 Experiments

The results of the generalization performance of the experiments can be found in apendix (B-1) and the computational cost in appendix (B-3) but are summarized in this section. Table (7-2) summarizes the results of the measure of generalization performance. For every technique, we have taken the mean and standard deviation of the log-likelihoods of the 20 experiments. Furthermore, table (7-3) summarizes the results of the measure of computational cost of the techniques. For every technique for every experiment we measure a mean epoch run-time. These epoch mean run-times are averaged. The results of the averaged epoch mean run-times can be found in the second column. For every technique for every experiment we measure the total run-time of one experiment. The average experiment run-times can be found in the third column. Finally, examples of the obtained predictors are displayed in figure (7-3).

Table 7-2: Results of the *generalization performance* of the techniques tested. For every method, we replicated 20 experiments. The generalization performance is measured in terms of log-likelihood regarding the test set. The mean and standard deviation of the log-likelihoods of the experiments can be found in the second and third column respectively.

	Mean	Standard deviation
Baseline	-197.102	65.149
Bagging	0.366	0.478
Dropout	0.016	0.254
BagDrop	0.804	0.038
Quasi-Dropout	0.205	0.254
Bagging + Quasi-Dropout	0.298	0.146

Table 7-3: Results of the *computational cost* of the techniques tested. For every method, we replicated 20 experiments. The computational cost is measured in terms of run-time. The mean of the run-times of the experiments can be found in the second column. The average of the means of the run-times of the epochs can be found in the second column. Notation: ms = milliseconds and sec = seconds

	Epochs average	Experiments average
Baseline	1.044 ms	61.227 sec
Bagging	1.168 ms	1190.940 sec
Dropout	1.246 ms	71.997 sec
BagDrop	1.501 ms	88.726 sec
Quasi-Dropout	1.497 ms	87.767 sec
Bagging + Quasi-Dropout	1.168 ms	1191.480 sec
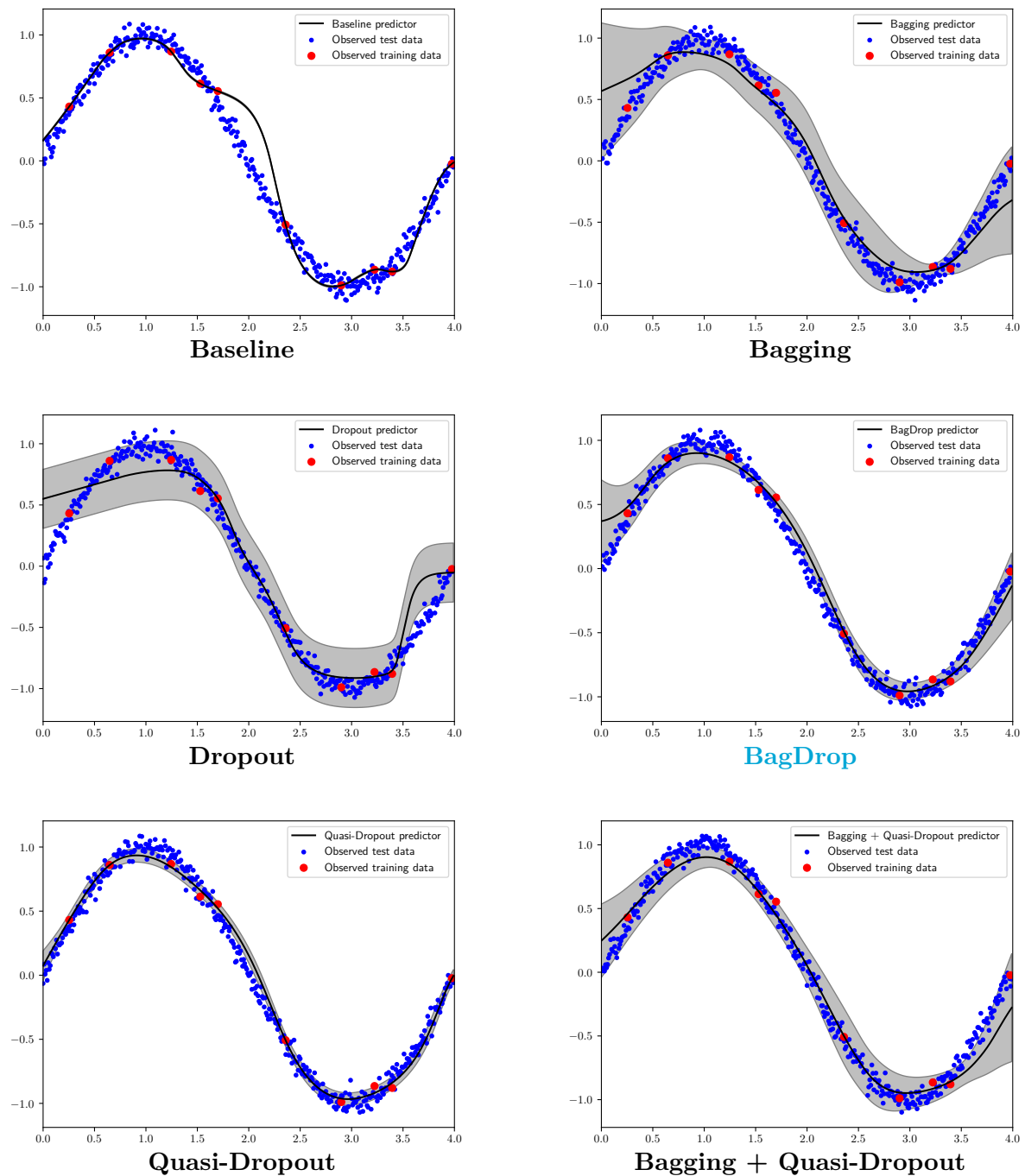
Figure 7-3: Examples of the obtained predictors using the different regularization techniques. The blue dots represent the observed test data. The red dots represent the observed training data. The lines represent the predictions of the obtained predictors and the grey surface represents the obtained variance of predictions. In case of the baseline and Dropout, we do not have an ensemble of predictions. Therefore, the obtained (mean of a) prediction is the obtained network function $\mathsf{NeuralNet}_{\hat{\theta}}(\mathbf{x}_i)$ and the obtained variance of a prediction is $\hat{\sigma}^2$. In case of Bagging, BagDrop, Quasi-Dropout and Bagging + Quasi-Dropout we had to deal with an ensemble of predictions. Therefore, the obtained (mean of a) prediction is $\frac{1}{B}\sum_{b=1}^{B} \mathsf{NeuralNet}_{\hat{\theta}_b}(X)$ and the obtained variance of a predictions is $\frac{1}{B}\sum_{b=1}^{B}\left(\left(\mathbb{E}(P|X) - \mathsf{NeuralNet}_{\hat{\theta}_b}\right)^2 + \hat{\sigma}_b^2\right)$.

7-1-3   Analysis

Generalization Performance

The results of the experiments are visualized in a boxplot in figure 7-4.
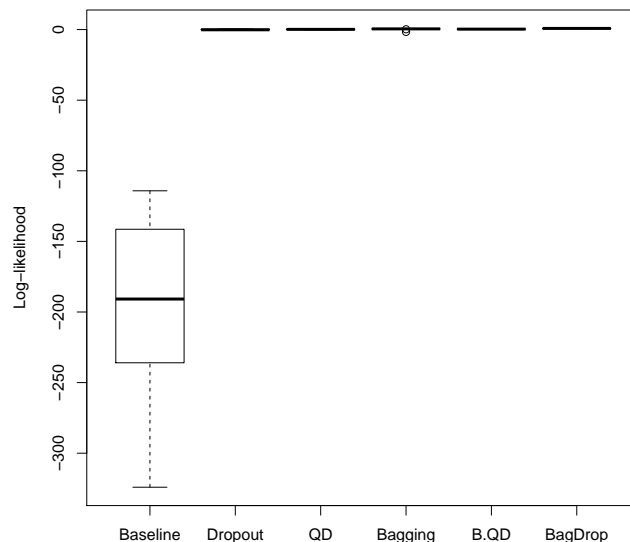
Figure 7-4: Boxplot of the generalization performance of the experiments. We can find the log-likelihood with respect to the test data on the vertical axis and the methods on the horizontal axis.

From visual inspection we can see that the baseline experiment has clearly significantly different log-likelihood compared to the baseline with additional techniques. Therefore, it would be meaningful to analyze the results without the baseline. However, to conduct a complete analysis we will start with analyzing the results *including* the baseline experiment.

As the Kruskal-Wallis test indicates significance ($\chi_5^2 = 94.431, p = 2.2e - 16 < 0.05$) for the experiments, it is meaningful to conduct multiple comparisons in order to identify differences between the procedures. The Nemenyi test p-values can be found in table (7-4).

Table 7-4: Nemenyi test pairwise p-value table for experiments without Early Stopping

	Baseline	Dropout	Quasi-Dropout	Bagging	Bagging + QD
Dropout	0.04674	-	-	-	-
Quasi-Dropout	0.00142	0.91341	-	-	-
Bagging	8.2e-08	0.03657	0.37525	-	-
Bagging + QD	0.00024	0.70448	0.99829	0.65049	-
BagDrop	5.9e-14	3.6e-07	7.4e-05	0.09242	0.00049

According to the Nemenyi post-hoc test for multiple joint samples without the baseline experiment, BagDrop log-likelihood differs highly significant ($p < 0.01$) from the baseline, Dropout, Quasi-Dropout and Bagging+Quasi-Dropout and BagDrop differs not significantly ($p > 0.05$) from Bagging. Please refer to appendix (C-2-1) for The results of the Kruskal-Wallis test and Nemenyi test.

The results of the experiments excluding the baseline are visualized in figure (7-5).

From visual inspection we can see that the BagDrop procedure differs significantly from the other methods. The Kruskal-Wallis test also indicates significance ($\chi_4^2 = 64.469, p = 3.329e - 13 < 0.05$). Therefore, it is meaningful to conduct multiple comparisons in order to identify differences
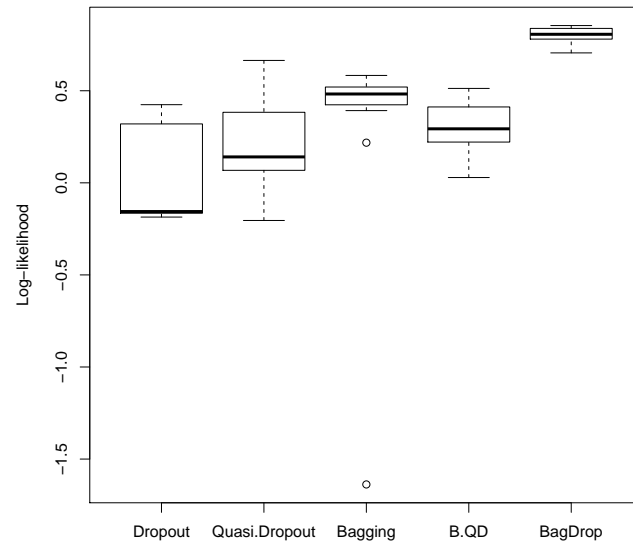
Figure 7-5: Boxplot of the generalization performance of the experiments without the baseline experiment. We can find the log-likelihood with respect to the test data on the vertical axis and the methods on the horizontal axis.

between the procedures without the baseline. The Nemenyi test p-values can be found in table (7-5).

Table 7-5: Nemenyi test pairwise p-value table for experiments without Early Stopping and without the baseline

	Dropout	Quasi-Dropout	Bagging	Bagging + QD
Quasi-Dropout	0.7514	-	-	-
Bagging	0.0042	0.1448	-	-
Bagging + QD	0.4339	0.9874	0.3735	-
BagDrop	4.1e-10	6.6e-07	0.0166	9.5e-06

According to the Nemenyi post-hoc test for multiple joint samples without the baseline experiment, BagDrop log-likelihood differs highly significant ($p < 0.01$) from Dropout, Quasi-Dropout and Bagging+Quasi-Dropout and BagDrop differs significantly ($p < 0.05$) from Bagging.

To conclude, in the experimental design, section (7-1-1), we described the ideal situation

$$c_{baseline} < \ c_{Quasi-Dropout} < c_{Bagging} \approx c_{Bagging+Dropout} \approx c_{BagDrop} \qquad (7\text{-}14)$$

where c_{method} indicates the log-likelihood of a method. If we look at the results, we can say that this (in)equality in fact *does* hold regarding this particular problem. Quasi-Dropout does perform better than the baseline experiment and the methods Bagging, Bagging + Quasi-Dropout and BagDrop perform better than Quasi-Dropout. Therefore, our hypothesis is right. We can see good improvement of generalization performance using the BagDrop procedure. In fact, BagDrop is even performing *better* than the other methods; the mean log-likelihood is higher and the standard deviation is much smaller. Thus, if we look back at the second research question:

> **Research Question 2**
>
> How does the BagDrop procedure perform in comparison to Bagging, in terms of generalization performance?

BagDrop

we can answer this question with: **the BagDrop procedure does perform better in comparison to Bagging, in terms of generalization performance.** Nonetheless, we have to keep in mind that the described and tested setting remains a **proof-of-concept.** Before we can (more) generally state that BagDrop performs well in terms of generalization performance, we have to either test it on a large collection of different real-life supervised learning problems, or mathematically proof it.

Computational Cost

Looking at the results of the run-time experiments in table (7-3), we can clearly see that BagDrop improves on Bagging with a factor of 14 regarding experiment run-time. It must be noted that BagDrop has a slightly higher experiment run-time than Dropout, due the additional operations that has to be done in BagDrop. Part of which is due to the additional operations in the optimization algorithm. We can see the increase in operations in the optimization algorithm by comparing the epoch run-time of BagDrop and Dropout, which differ by 0.255 ms. This means that, on average, more operations have to be done during one epoch. This is not surprising if we look at the script in appendix (A). Still, the experiment and epoch run-time only slightly differ from Dropout. Therefore, we can state that the run-time of BagDrop is more ore less the same as Dropout.

However, which method has the optimal computational cost? The baseline experiment clearly has the lowest epoch and experiment run-time, which is not very surprising if we look at the script in appendix (A). We could argue that the baseline must be used in practice, without additional regularization techniques, when it comes to computational cost. However, we must not forget that finding the optimal hidden layers and/or hidden units per layer is a very time consuming task. We would prefer setting out a reasonably large number of hidden units and/or layers and use regularization as we described in section (5-2).

To conclude, in the experimental design, section (7-1-1), we described the ideal situation for the computational cost t_{method} of a method:

$$t_{Dropout} \approx t_{Quasi-Dropout} \approx t_{BagDrop} \tag{7-15}$$

where t_{method} indicates the run-time of a method. If we look at the results, we can say that this equality does hold regarding this particular problem. Dropout, Quasi-Dropout and BagDrop all perform more or less the same, in terms of the run-time. Therefore, our hypothesis is right. Moreover, BagDrop performs much better compared to Bagging and Bagging + Quasi-Dropout. Thus, if we look back at the third and last research question:

> **Research Question 3**
>
> How does the BagDrop procedure perform in comparison to Dropout, in terms of computational cost?

we can answer this question with: **the BagDrop procedure does perform similar in comparison to Dropout, in terms of computational cost.** Again, we have to keep in mind that the described and tested setting remains a **proof-of-concept**.

Conclusion & Discussion

8-1 Conclusion

Neural Networks are very flexible models but can be sensitive to noise in the training data. When certain guidelines are not followed, overfitting may occur. Bagging is a regularization technique that helps prevent overfitting. However, Bagging has the disadvantage of computational cost. We showed how to apply Bagging in a Dropout-inspired way to obtain a new method called BagDrop to conquer the computational cost problem. As a proof-of-concept, we compared BagDrop to Dropout, Bagging and derived techniques in a simple experimental set-up. BagDrop does well with respect to generalization performance, relative to Bagging and alternatives. Furthermore, the computational cost of BagDrop is much lower than Bagging. These observations together may be interesting for future research.

8-2 Discussion

8-2-1 Scaling up to Real-life Problems

We have to keep in mind that the described and tested setting remains a **proof-of-concept.** Before we can (more) generally state that BagDrop performs well in terms of generalization performance, we have to either test it on a large collection of different real-life supervised learning problems, or mathematically proof it. There exist many big datasets that can be used to test BagDrop. One of them is MNIST, which is a popular handwritten digits database. MNIST can be used to test BagDrop on high dimensional data. Figure (8-1) shows a random sample from the MNIST database.

8-2-2 Bagging

In contrast to what many people think, it is not mathematically proven that Bagging reduces variance of predictions. Quite the reverse; papers suggest situations in which Bagging only makes performance worse. In the paper of Breiman (Breiman, 1996) he already suggests that Bagging increases variance when "stable" models, like linear regression, are used. The paper of Grandvalet (Grandvalet, 2004) suggest more situations. Therefore, we have to be very careful when using Bagging.

8-2-3 Dependence of Dropout masks

The BagDrop procedure combines several neural network architectures that all share parameters. Parameter sharing, however, breaks the assumption behind Bagging that the ensemble

Figure 8-1: Random sample from the MNIST database

components are independent. The predictions of models in the ensemble may be more similar to eachother compared to regular Bagging, resulting in lower variance in the ensemble. One possible fix is to sample the bootstrap samples in a different way. We could also use the (Bayesian) Bootstrap, which will be shortly explained in the following section.

8-2-4 The Bayesian Bootstrap

We could use the Bayesian Bootstrap (Rubin et al., 1981). If we compare the Bayesian Bootstrap with the Bootstrap (5-3-1), a bootstrap sample is obtained by sampling with replacement from the training set D. Now define a Bayesian Bootstrap sample D^b with $b \in \{1, .., B\}$ with N categories. The Bayesian Bootstrap can be expressed as sampling the category proportions from a multinomial distribution.

$$p\hat{\mathbf{d}}^* \sim \text{Mult}(p, \hat{\mathbf{d}}) \tag{8-1}$$

The probability that an observed sample $(\mathbf{x}, y) \in D$ falls in category j is d_j. The prior for $\mathbf{d} = \{d_1, .., d_N\}$ is a symmetric Dirichlet distribution with parameter $\alpha = (\alpha_1, .., \alpha_N)$. One possible fix to increase the difference between the dropout masks in the ensemble is by modifying $\mathbf{d}$ to a concentrated distribution. Ideas for $\mathbf{d}$ are:

1. Using $\mathbf{d} = \text{Dir}(\alpha_1 = \alpha_2 = .. = \alpha_N = \alpha)$, with $\alpha < 1$, increasing its contractedness to smaller number of datapoints, compared to $\alpha = 1$ (used in the regular Bayesian Bootstrap)

2. Assigning each datapoint at random to a single model. This corresponds to a fairly concentrated $\mathbf{d}$. When used in combination with Dropout rate $1/2$, this has the effect of assigning half the dataset (at random) to each hidden unit. This makes intuitive sense since if you squint a little, this can be interpreted as a form of bootstrap at the neuron level. Perhaps this intuition can be translated into something rigorous.

Bibliography

Bauer, E. and Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1):105–139.

Bishop, C. M. (2006). *Pattern recognition and machine learning*, volume 1. Springer New York.

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Clemen, R. T. (1989). Combining forecasts: A review and annotated bibliography. *International journal of forecasting*, 5(4):559–583.

Efron, B. and Tibshirani, R. J. (1994). *An introduction to the bootstrap*. CRC press.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Grandvalet, Y. (2004). Bagging equalizes influence. *Machine Learning*, 55(3):251–270.

Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations 2015*.

Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621.

Kuhn, M. and Johnson, K. (2013). *Applied predictive modeling*, volume 810. Springer.

Nemenyi, P. (1962). Distribution-free multiple comparisons. In *Biometrics*, volume 18, page 263. INTERNATIONAL BIOMETRIC SOC 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210.

Perrone, M. P. (1993). *Improving regression estimation: Averaging methods for variance reduction with extensions to general convex measure optimization*. PhD thesis, Brown University Providence, RI.

Poggio, T., Rifkin, R., Mukherjee, S., and Rakhlin, A. (2002). Bagging regularizes. Technical report, DTIC Document.

Rice, J. (2006). *Mathematical statistics and data analysis*. Nelson Education.

Rubin, D. B. et al. (1981). The bayesian bootstrap. *The annals of statistics*, 9(1):130–134.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252.

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

Student (1908). The probable error of a mean. *Biometrika*, pages 1–25.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

Appendices

Appendix A: Experiments (in Python with Tensorflow)

A-1   Appendix A.1: Simulation Dataset Proof-of-Concept

```python
'''
Created on 4 mei 2017

@author: Friso Kingma
'''

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
from tensorflow.contrib import learn

from matplotlib import pyplot as pl
import numpy as np
import math as mth
import sys
import os
import progressbar
from matplotlib.delaunay.testfuncs import saddle
import csv

def sine(x, period = 4):
    return mth.sin(((2*np.pi)/period)*(x))

def polynomial(x):
    return (np.power(x,3)+3*np.power(x,2)-6*x-8)/4

def block(x):
    if x > 1:
        return 10
    else:
        return -10

#Set train data
sin_x_data = []
sin_y_data = []
values = 10
```

```
39     starting_point = 0
40     end_point = 4.0
41
42     sample_interval = np.random.uniform(starting_point,end_point,values)
43
44     step_size = abs(starting_point-end_point)/values
45     for i in range(values):
46             point = starting_point + i*step_size
47             sin_x_data.append([0, sample_interval[i]])
48             sin_y_data.append(sine(sample_interval[i]))
49
50
51     # Add some NOISE
52     sin_y_data += np.random.normal(0,0.05,values)
53
54     #reshape training data for plot
55     sin_x_data_plot = []
56     for i in range(np.shape(sin_x_data)[0]):
57         sin_x_data_plot.append(sin_x_data[i][1])
58     sin_x_data_plot = np.array(sin_x_data_plot)
59
60     #plot distribution of ensemble
61     pl.figure()
62     pl.scatter(sin_x_data_plot, sin_y_data, c="darkorange", label="data")
63     pl.show()
64
65     print(sin_y_data)
66     with open('data.csv', 'w') as csvfile:
67         writer = csv.writer(csvfile, delimiter=' ', quotechar='|', ...
               quoting=csv.QUOTE_MINIMAL)
68         for i in range(values):
69             writer.writerow([sin_x_data[i][1],sin_y_data[i]])
```

A-2   Appendix A.1: Experiments Proof-of-Concept

```python
'''
Created on 4 mei 2017

@author: Friso Kingma
'''

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
from tensorflow.contrib import learn

from matplotlib import pyplot as pl
import numpy as np
import math as mth
import sys
import os
import progressbar
from matplotlib.delaunay.testfuncs import saddle
import csv
import argparse, json
import time

from matplotlib import rc

rc('text', usetex=True)

pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [
        r"\usepackage[utf8]{inputenc}",
        r"\usepackage[T1]{fontenc}",
        r"\usepackage{cmbright}",
        r"\usepackage{amsmath}"
        ],
}
pl.rcParams.update(pgf_with_pdflatex)

#command-line arguments
parser = argparse.ArgumentParser()
parser.add_argument('--alpha0', type=float, default=0.002, ...
    help='learning rate start')
parser.add_argument('--alpha1', type=float, default=0.00001, ...
    help='learning rate stop')
parser.add_argument('--beta1', type=float, default=0, help='beta1 ...
    (adam parameter)')
parser.add_argument('--epochs', type=int, default=50000, ...
    help='number of training epochs')
parser.add_argument('--batch_size', type=int, default=10, ...
    help='batch size')

# must be equal to num_masks, or at least one of them should be 1
parser.add_argument('--models', type=int, default=1, help='number ...
```

BagDrop

```
            of bootstrap_samples for new method. ')
50    parser.add_argument('--num_masks', type=int, default=1, ...
          help='number of dropout masks, for new method. If only applying ...
          dropout.')

52    parser.add_argument('--keep_prob', type=float, default=1.0, ...
          help='keep probability of dropout')
53    parser.add_argument('--units', type=int, default=50, help='number ...
          of units per hidden layer')
54    parser.add_argument('--num_layers', type=int, default=3, ...
          help='number of hidden layers') #should be at least 3 layers
55    parser.add_argument('--nonlinearity', type=str, default='elu', ...
          help='relu / elu / sigmoid') #all work, but elu works best

57    parser.add_argument('--min_std', type=float, default=0.001, ...
          help='minimum value of standard deviation')

59    #only bagging hyperparameters
60    parser.add_argument('--bagging_models', type=int, default=20, ...
          help='Total number of neural networks trained, for bagging, ...
          this should be more than 1')

62    # Number of experiments
63    parser.add_argument('--experiments', type=int, default=1, ...
          help='Total experiments')

65    # Plotting options
66    parser.add_argument('--plot_default_mask', type=int, default=0, ...
          help='Whether to plot heuristic prediction with default mask')

68    # Early stopping options
69    parser.add_argument('--early_stopping', type=int, default=1, ...
          help='Whether to do early stopping')
70    parser.add_argument('--early_stopping_default_mask', type=int, ...
          default=1, help='Whether to calculate log-likelihood of default ...
          mask at early stopping')

72    # Training set and validation set options
73    parser.add_argument('--datafile_train', type=str, ...
          default='data1.csv', help='Which data file to load')
74    parser.add_argument('--datafile_validate', type=str, ...
          default='data3.csv', help='Which data file to load')

76    args = parser.parse_args()
77    print('input args:\n', json.dumps(vars(args), indent=4, ...
          separators=(',', ':')))  # pretty print args



81    def sine(x, period = 4):
82        return mth.sin(((2*np.pi)/period)*(x))

84    def polynomial(x):
85        return (np.power(x,3)+3*np.power(x,2)-6*x-8)/4

87    def block(x):
88        if x > 1:
89            return 10
```

```
 90            else :
 91                return -10
 92
 93      #Set train data
 94      sin_x_data = []
 95      sin_y_data = []
 96      with open(args.datafile_train , 'r') as csvfile:
 97          reader = csv.reader(csvfile , delimiter=' ', quotechar='|')
 98          for row in reader:
 99              if row:
100                  sin_x_data.append([0, float(row[0])])
101                  sin_y_data.append(float(row[1]))
102
103      #Set validation data
104      sin_x_validate = []
105      sin_y_validate = []
106      with open(args.datafile_validate , 'r') as csvfile:
107          reader = csv.reader(csvfile , delimiter=' ', quotechar='|')
108          for row in reader:
109              if row:
110                  sin_x_validate.append([0, float(row[0])])
111                  sin_y_validate.append(float(row[1]))
112
113      #Set test data
114      sin_x_test = []
115      sin_y_test = []
116      values = 400
117      starting_point = 0
118      end_point = 4.0
119
120      step_size = abs(starting_point -end_point)/values
121      for i in range(values):
122              point = starting_point + i*step_size
123              sin_x_test.append([0, point])
124              sin_y_test.append(sine(point))
125
126      # Add some NOISE
127      sin_y_test += np.random.normal(0 ,0.05 ,values)
128
129      X_test , Y_test = np.array(sin_x_test), np.array(sin_y_test)
130      X_train , Y_train = np.array(sin_x_data), np.array(sin_y_data)
131      X_validate , Y_validate = np.array(sin_x_validate), ...
            np.array(sin_y_validate)
132
133      total_len = X_train.shape[0]
134      input_features = X_train.shape[1]
135
136      # Use GPU?
137      GPU = 0
138
139      # Hyperparameters
140
141      network_structure = [input_features] + ...
            [args.units]*args.num_layers + [1]
142      nonlinearity = {'sigmoid':tf.nn.sigmoid, 'relu':tf.nn.relu, ...
            'elu':tf.nn.elu}[args.nonlinearity]
143
144      # Set-up device for processing
```

BagDrop

```
145     with tf.device('/cpu:0' if GPU == 0 else '/gpu:0'):

146
147         # tf Graph input
148         x = tf.placeholder("float", [None, input_features])
149         y = tf.placeholder("float", [None])
150         mask = tf.placeholder("float", None)
151         learning_rate = tf.placeholder("float", None)

152
153         # Create model
154         def multilayer_perceptron(layer, network_structure, weights, ...
                biases, mask):
155             layer -= 2. # since data is centered at 2.
156             for i in range(len(network_structure))[:-1]:
157                 layer = tf.add(tf.matmul(layer, weights[i]), biases[i])
158                 if i != range(len(network_structure))[-2]:
159                     layer = nonlinearity(layer)
160                     layer = layer * mask[i]
161             return layer

162
163         # Store layers weight & bias
164         weights = []
165         biases = []
166         for i in range(len(network_structure))[:-1]:
167             weights.append(tf.Variable(tf.random_normal([network_structure[i], ...
                    network_structure[i+1]], 0, 0.1)))
168             biases.append(tf.Variable(tf.random_normal([network_structure[i+1]], ...
                    0, 0.1)))

169
170         # Store standard deviation
171         std = args.min_std + tf.exp(tf.Variable(0.0))

172
173         # Store bootstrap samples
174         samples_x = []
175         samples_y = []

176
177         if(args.bagging_models > 1):
178             if(args.bagging_models == args.models):
179                 print("Number of bagging models can not be the same as ...
                        number of BagDrop models")
180                 sys.exit()
181             else:
182                 bootstrap_samples = args.bagging_models
183         else:
184             bootstrap_samples = args.models

185
186         # Store bootstrapped training sets
187         for m in range(bootstrap_samples):
188             #initialize bootstrap sample vector
189             X_train_bootstrap = []
190             Y_train_bootstrap = []

191
192             #create bootstrap sample
193             for i in range(total_len):
194                 rand_int = np.random.randint(total_len)
195                 X_train_bootstrap.append(X_train[rand_int].tolist())
196                 Y_train_bootstrap.append(Y_train[rand_int].tolist())
197             samples_x.append(X_train_bootstrap)
198             samples_y.append(Y_train_bootstrap)
```

BagDrop

```
199         samples_x = np.array(samples_x)
200         samples_y = np.array(samples_y)
201
202         # Store masks
203         masks = []
204         default_mask = []
205         # Create default mask for test time
206         for layer in range(len(network_structure))[:-2]:
207             ones_tensor = np.full([network_structure[layer+1]], ...
                    1.0).tolist()
208             default_mask.append(ones_tensor)
209
210         # Create masks of the models
211         for m in range(args.num_masks):
212             masks_layer = []
213             for layer in range(len(network_structure))[:-2]:
214                 random_tensor = args.keep_prob
215                 random_tensor += ...
                        np.random.uniform(0,1,[network_structure[layer+1]])
216                 binary_tensor = np.floor(random_tensor)
217                 new_tensor = (np.divide(1.0, args.keep_prob) * ...
                        binary_tensor).tolist()
218                 masks_layer.append(new_tensor)
219             masks.append(masks_layer)
220         masks = np.array(masks)
221
222         # Construct model
223         pred = multilayer_perceptron(x, network_structure, weights, ...
                biases, mask)
224         pred = tf.transpose(pred)
225
226         # Define loss and optimizer
227         # Log-likelihood of normal distribution, not entirely right.
228         cost = tf.div(tf.reduce_mean(tf.square(pred-y)), ...
                tf.square(std))+tf.div(tf.constant(1.0),tf.constant(2.0))*tf.log(tf.square(
229         optimizer = ...
                tf.train.AdamOptimizer(learning_rate=learning_rate,beta1=args.beta1).minimi
230
231     bar_experiments = progressbar.ProgressBar()
232     log_likelihood_array = []
233     for _ in bar_experiments(range(args.experiments)):
234
235         # Add ops to save and restore all the variables.
236         saver = tf.train.Saver()
237
238
239         # Launch the graph
240         with tf.Session() as sess:
241
242             bar_bagging = progressbar.ProgressBar()
243
244             all_pred_means = []
245             all_pred_stds = []
246             #Setting up progress bar
247             for _ in range(args.bagging_models):
248                 sess.run(tf.global_variables_initializer())
249
250                 if(args.bagging_models > 1):
```

BagDrop

```
251                         random_sample = np.random.randint(bootstrap_samples)
252                         X_train, Y_train = samples_x[random_sample], ...
                                samples_y[random_sample]
253
254                    # Initialize log-likelihood comparison
255                    log_likelihood_comparison = -1000000.0
256
257                    # Training cycle with progress bar logging
258                    bar_epochs = progressbar.ProgressBar()
259                    for epoch in range(args.epochs):
260
261                        avg_cost = 0.
262
263                        # compute learning rate for this epoch
264                        progress = (1.*epoch)/args.epochs
265                        learning_rate_epoch = (1-progress)*args.alpha0 + ...
                                progress*args.alpha1
266
267                        #total_batch = int((1.0*total_len)/args.batch_size)
268                        assert total_len%args.batch_size == 0
269                        num_iterations_per_epoch = ...
                                int(total_len/args.batch_size)
270
271                        #print(total_batch)
272                        for i in range(num_iterations_per_epoch):
273
274                            # Set up arguments
275                            batch_x = []
276                            batch_y = []
277                            mask_batch = []
278
279                            # Fill arguments
280                            for i in range(args.batch_size):
281                                #get random model and random data point
282                                random_model = ...
                                    np.random.randint(args.num_masks)
283                                random_point = np.random.randint(total_len)
284
285                                #append datapoint to batch
286                                # if more than 1 model, apply 'bagging'
287                                if  args.models > 1:
288                                    batch_x.append(samples_x[random_model][random_point].t
289                                    batch_y.append(samples_y[random_model][random_point].t
290                                # if 1 model, use just the original ...
                                    training set
291                                else:
292                                    batch_x.append(X_train[random_point].tolist())
293                                    batch_y.append(Y_train[random_point].tolist())
294
295                                #for every layer, add an random mask
296                                for layer in ...
                                    range(len(network_structure))[:-2]:
297                                    if(i == 0):
298                                        #if its the first one, append
299                                        mask_batch.append(masks[random_model][layer].tolis
300                                    else:
301                                        #if its the second one, extend. ...
                                            Later it will be converted to ...
```

BagDrop

```
                                        the form that can be processed ...
                                        by the algorithm.
302                         mask_batch[layer].extend(masks[random_model][layer
303
304                 #convert data batches to numpy batches
305                 batch_x = np.array(batch_x)
306                 batch_y = np.array(batch_y)
307
308                 #convert mask_batch to appropriate form
309                 mask_batch = np.array(mask_batch)
310                 mask_batch = np.reshape(mask_batch, ...
                        (len(network_structure)-2, args.batch_size, ...
                        args.units))
311
312                 # Run optimization op (backprop) and cost op ...
                        (to get loss value)
313                 _, c, p = sess.run([optimizer, cost, pred], ...
                        feed_dict={x: batch_x, y: batch_y, mask: ...
                        mask_batch, learning_rate: ...
                        learning_rate_epoch})
314
315                 # Compute average loss
316                 avg_cost += c / num_iterations_per_epoch
317
318             # Compute validation set log-likelihoods. If these ...
                    are better, store the parameters.
319             if(args.early_stopping == 1):
320                 if(epoch%1000 == 0):
321                     if(args.early_stopping_default_mask == 1):
322                         pred_mean_current = sess.run(pred, ...
                                feed_dict={x: X_validate, mask: ...
                                default_mask})
323                         pred_std_current = sess.run(std)
324
325                         # Calculate log-likelihood of default ...
                                mask given test data
326                         pred_variance_current = ...
                                np.square(pred_std_current)
327                         log_likelihood_current = ...
                                -(1.0/2.0)*np.mean(np.square(pred_mean_current-Y_va
                                + np.log(pred_variance_current) + ...
                                np.log(2*np.pi))
328                     else:
329                         all_pred_means_step = []
330                         all_pred_stds_step = []
331                         for m in range(args.num_masks):
332                             all_pred_means_step.append(sess.run(pred, ...
                                    feed_dict={x: X_validate, mask: ...
                                    masks[m]}))
333                             all_pred_stds_step.append(sess.run(std))
334
335                         #calculate mean over predictions
336                         all_pred_means_step = ...
                                np.array(all_pred_means_step)
337
338                         #mean and standard deviation of ensemble
339                         bagged_pred_mean_step = ...
                                np.mean(all_pred_means_step, axis=0)
```

BagDrop

```
340                                bagged_pred_variance_step = ...
                                       np.square(np.std(all_pred_means_step, ...
                                       axis = 0))+ ...
                                       np.mean(np.square(all_pred_stds_step), ...
                                       axis = 0)
341                                bagged_pred_std_step = ...
                                       np.sqrt(bagged_pred_variance_step)
342
343                                # Calculate log-likelihood of model ...
                                       given test data
344                                log_likelihood_current = ...
                                       -(1.0/2.0)*np.mean(np.square(bagged_pred_mean_step[
                                       + ...
                                       np.log(bagged_pred_variance_step[0]) ...
                                       + np.log(2*np.pi))
345
346                            # Compare current log_likelihood on ...
                                   validation-set to past log-likelihood
347                            if(log_likelihood_current ≥ ...
                                   log_likelihood_comparison):
348                                # Store parameters
349                                print(epoch, ": ", log_likelihood_current)
350                                save_path = saver.save(sess, ...
                                       "tmp\model.ckpt")
351                                log_likelihood_comparison = ...
                                       log_likelihood_current
352
353                # Restore checkpoint with highest
354                if(args.early_stopping == 1):
355                    saver.restore(sess, "tmp\model.ckpt")
356
357                #calculate error and predicted values
358                pred_mean = sess.run(pred, feed_dict={x: X_test, mask: ...
                       default_mask})
359                pred_std = sess.run(std)
360
361                # Save current model prediction and standard deviation ...
                       if we have more than 1 bagging model
362                if(args.bagging_models > 1):
363                    all_pred_means.append(pred_mean)
364                    all_pred_stds.append(pred_std)
365
366            # Save all the predictions and standard deviations if we ...
                   have 1 model
367            if (args.bagging_models == 1):
368                for m in range(args.num_masks):
369                    all_pred_means.append(sess.run(pred, feed_dict={x: ...
                           X_test, mask: masks[m]}))
370                    all_pred_stds.append(sess.run(std))
371
372        # Cast mean to numpy array
373        all_pred_means = np.array(all_pred_means)
374
375        #mean and standard deviation of ensemble
376        bagged_pred_mean = np.mean(all_pred_means, axis=0)
377        bagged_pred_variance = np.square(np.std(all_pred_means, axis = ...
               0))+ np.mean(np.square(all_pred_stds), axis = 0)
378        bagged_pred_std = np.sqrt(bagged_pred_variance)
```

BagDrop

```
379
380        # Calculate log-likelihood of model given test data
381        log_likelihood = ...
               -(1.0/2.0)*np.mean(np.square(bagged_pred_mean[0]-Y_test)/bagged_pred_varian
               + np.log(bagged_pred_variance[0]) + np.log(2*np.pi))
382
383        # Calculate log-likelihood of default mask given test data
384        pred_variance = np.square(pred_std)
385        log_likelihood_default = ...
               -(1.0/2.0)*np.mean(np.square(pred_mean-Y_test)/pred_variance ...
               + np.log(pred_variance) + np.log(2*np.pi))
386
387        #log
388        print("Bagged model has log likelihood: ", log_likelihood)
389        print("Default mask has log likelihood: ", log_likelihood_default)
390
391        #reshape test data for plot
392        sin_x_test_plot = []
393        for i in range(len(bagged_pred_mean[0])):
394            sin_x_test_plot.append(sin_x_test[i][1])
395        sin_x_test_plot = np.array(sin_x_test_plot)
396
397        #reshape training data for plot
398        sin_x_data_plot = []
399        for i in range(np.shape(sin_x_data)[0]):
400            sin_x_data_plot.append(sin_x_data[i][1])
401        sin_x_data_plot = np.array(sin_x_data_plot)
402
403        #plot distribution of ensemble
404        pl.figure()
405        pl.plot(sin_x_test_plot, sin_y_test, c="blue", ...
               label=r"Observed test data")
406        if args.plot_default_mask:
407            pl.plot(sin_x_test_plot, pred_mean[0], 'k-', ...
                   label=r"Dropout", c='green')
408            pl.fill_between(sin_x_test_plot, pred_mean[0] - pred_std, ...
                   pred_mean[0] + pred_std, facecolor="0.75", edgecolor="0.5")
409        else:
410            pl.plot(sin_x_test_plot, bagged_pred_mean[0], 'k-', ...
                   c="black", label=r"Bagging + Quasi-Dropout")
411            pl.fill_between(sin_x_test_plot, bagged_pred_mean[0] - ...
                   bagged_pred_std[0], bagged_pred_mean[0] + ...
                   bagged_pred_std[0], facecolor="0.75", edgecolor="0.5")
412        pl.scatter(sin_x_data_plot, sin_y_data, c="red", ...
               label=r"Observed training data")
413        pl.legend()
414        pl.show()
415
416        # Save log_likelihood value
417        log_likelihood_array.append(log_likelihood_default)
418        #log
419        print("Finished one experiment!")
420
421    log_likelihood_array = np.array(log_likelihood_array)
422
423    for e in range(args.experiments):
424        print("Experiment ", e+1, " has log_likelihood: ", ...
               log_likelihood_array[e])
```

BagDrop

```
425
426    print("Mean of log_likelihoods: ", np.mean(log_likelihood_array, ...
           axis=0))
427    print("Standard deviation of log_likelihoods: ", ...
           np.std(log_likelihood_array, axis=0))
```

Appendix B: Results of Experiments

B-1 Appendix B.1: Generalization Performance Proof-of-Concept

Table B-1: Results of experiments Proof-of-Concept

	Baseline	Dropout	Quasi-Dropout	Bagging	Bagging + QD	BagDrop
1	-114.17389	-0.18595	-0.19365	0.49831	0.33037	0.79048
2	-160.27664	0.25434	0.15758	0.56219	0.45119	0.70552
3	-223.35540	0.33489	0.38168	-1.63772	0.07428	0.76931
4	-140.61131	0.32580	0.53243	0.49657	0.27808	0.80479
5	-289.07741	-0.18241	0.62623	0.55718	0.51265	0.78029
6	-183.65925	-0.14817	0.64782	0.51523	0.02871	0.84985
7	-161.94323	-0.15181	0.11675	0.42051	0.28928	0.78379
8	-229.42208	-0.16389	-0.19859	0.49543	0.11423	0.83259
9	-321.88875	-0.16608	0.20976	0.58313	0.35073	0.79066
10	-117.62754	-0.16189	0.08816	0.46935	0.35301	0.82440
11	-324.08235	-0.16150	0.09147	0.51429	0.25871	0.84745
12	-197.98399	-0.17738	0.35142	0.52532	0.19323	0.85397
13	-156.77452	0.31411	0.66467	0.39723	0.27967	0.77195
14	-242.50213	-0.16331	-0.20440	0.46073	0.37298	0.82159
15	-124.98405	-0.17355	0.15113	0.42687	0.07478	0.80842
16	-246.15205	0.38029	0.05779	0.44378	0.48238	0.81325
17	-227.62060	0.41406	0.13025	0.21809	0.29728	0.78045
18	-131.70400	0.42435	0.38430	0.44698	0.24949	0.84454
19	-142.20302	-0.12820	0.03156	0.52819	0.48360	0.84914
20	-205.99849	-0.16002	0.07794	0.39224	0.47777	0.76077
Mean:	**-197.10203**	**0.01618**	**0.20522**	**0.36569**	**0.29762**	**0.80416**
STD:	**65.14880**	**0.25365**	**0.26943**	**0.47831**	**0.14593**	**0.03804**

B-2 Appendix B.2: Generalization Performance Proof-of-Concept with Early Stopping

Table B-2: Results of experiments Proof-of-Concept with Early Stopping

	Baseline	Dropout	Quasi-Dropout	Bagging	Bagging + QD	BagDrop
1	0.970	0.361	0.981	0.358	-0.130	0.904
2	1.012	0.397	1.051	-0.144	0.602	0.901
3	1.032	-0.054	1.061	0.413	0.051	0.872
4	1.074	-0.175	1.007	0.502	-0.168	0.822
5	0.749	-0.044	1.104	0.032	0.679	0.852
6	0.873	-0.150	1.022	0.399	0.576	0.872
7	1.102	-0.152	1.028	-0.012	0.132	0.821
8	1.045	-0.176	1.082	0.323	0.534	0.834
9	1.015	0.370	1.054	-0.001	0.094	0.823
10	0.872	0.270	1.006	0.381	-0.276	0.883
11	0.981	-0.185	1.002	0.472	0.089	0.928
12	0.944	0.325	1.075	0.058	0.456	0.801
13	0.999	-0.209	1.165	0.392	-0.017	0.835
14	1.094	-0.073	1.057	0.410	0.044	0.887
15	0.997	0.027	1.062	0.465	0.037	0.873
16	0.982	-0.159	1.141	0.383	0.127	0.871
17	0.978	-0.180	1.134	0.411	0.510	0.851
18	0.817	-0.005	1.092	-0.138	0.008	0.885
19	1.062	0.360	1.145	0.418	0.652	0.903
20	1.101	0.320	1.098	0.057	0.118	0.829
Mean	**0.985**	**0.043**	**1.068**	**0.259**	**0.206**	**0.862**
STD:	**0.095**	**0.235**	**0.052**	**0.220**	**0.297**	**0.035**

BagDrop

B-3 Appendix B.2: Computational Cost Proof-of-Concept

B-3-1 Appendix B.2.1: Epochs run-time

	Baseline	Bagging	Dropout	Bagdrop	Quasi-Dropout	Bagging + QD
1	1.014	1.123	1.188	1.048	1.411	1.033
2	1.006	1.203	1.192	1.092	1.410	1.071
3	1.010	1.219	1.205	1.216	1.490	1.079
4	1.011	1.253	1.247	1.241	1.612	1.198
5	1.016	1.228	1.238	1.432	1.657	1.237
6	1.023	1.170	1.262	1.423	1.735	1.285
7	1.019	1.176	1.241	1.585	1.584	1.217
8	1.035	1.177	1.252	1.689	1.591	1.219
9	1.036	1.149	1.246	1.659	1.622	1.140
10	1.044	1.133	1.245	1.707	1.643	1.187
11	1.049	1.140	1.244	1.579	1.557	1.169
12	1.058	1.141	1.261	1.640	1.629	1.145
13	1.064	1.142	1.284	1.689	1.681	1.135
14	1.055	1.161	1.243	1.646	1.554	1.158
15	1.067	1.188	1.267	1.653	1.520	1.184
16	1.080	1.133	1.259	1.686	1.437	1.164
17	1.073	1.129	1.265	1.575	1.429	1.182
18	1.069	1.193	1.270	1.541	1.157	1.182
19	1.072	1.144	1.258	1.467	1.108	1.183
20	1.081	1.158	1.260	1.451	1.118	1.189
Mean:	**1.044**	**1.168**	**1.246**	**1.501**	**1.497**	**1.168**

BagDrop

B-3-2   Appendix: B.2.2: Experiments run-time

Table B-3: Results of Experiments regarding Experiment Run-time

	Baseline	Bagging	Dropout	BagDrop	Quasi-Dropout	Bagging + QD
1	52.369 s	18.775 m	61.253 s	54.382 s	74.038 s	17.272 m
2	51.864 s	20.118 m	61.376 s	56.391 s	73.005 s	17.900 m
3	52.582 s	20.410 m	62.593 s	63.272 s	77.721 s	18.061 m
4	53.277 s	20.996 m	65.401 s	65.139 s	85.753 s	20.064 m
5	54.154 s	20.605 m	65.767 s	76.784 s	88.292 s	20.763 m
6	55.054 s	19.669 m	67.681 s	77.036 s	93.883 s	21.597 m
7	55.737 s	19.783 m	67.252 s	86.270 s	88.273 s	20.509 m
8	57.377 s	19.841 m	68.819 s	93.901 s	88.712 s	20.572 m
9	58.313 s	19.405 m	69.193 s	92.869 s	92.013 s	19.272 m
10	59.780 s	19.175 m	70.211 s	99.964 s	94.565 s	20.087 m
11	60.976 s	19.342 m	71.780 s	93.506 s	94.517 s	19.842 m
12	62.533 s	19.402 m	73.464 s	99.109 s	95.829 s	19.496 m
13	63.647 s	19.474 m	75.826 s	101.259 s	98.472 s	19.362 m
14	64.596 s	19.851 m	74.944 s	102.517 s	94.741 s	19.805 m
15	66.405 s	20.385 m	77.293 s	102.721 s	94.634 s	20.295 m
16	68.398 s	19.497 m	78.110 s	104.869 s	91.376 s	20.026 m
17	69.743 s	19.490 m	79.931 s	101.506 s	94.636 s	20.412 m
18	70.218 s	20.654 m	81.719 s	101.292 s	81.987 s	20.483 m
19	72.564 s	19.897 m	82.806 s	99.267 s	74.756 s	20.577 m
20	74.953 s	20.222 m	84.528 s	102.457 s	78.126 s	20.776 m
Average:	**61.227 s**	**19.850 m**	**71.997 s**	**88.725 s**	**87.766 s**	**19.859 m**

BagDrop

Appendix C: Statistical Analysis of Experiments (in R)

C-1 Appendix C.1: Script

```
1  require(stats)
2  library(PMCMR)
3
4  log_likelihood = ...
       as.matrix(read.csv("experiments_stopping.csv",sep=";",header=TRUE))
5  print(log_likelihood)
6  boxplot(log_likelihood, ylab="Log-likelihood")
7  friedman.test(log_likelihood)
8  posthoc.friedman.nemenyi.test(log_likelihood)
```

C-2 Appendix C.2: Results

C-2-1 Appendix C.2.A: Without Early Stopping

Listing C.1: R output

```
 1  > require(stats)
 2  > library(PMCMR)
 3  >
 4  > log_likelihood = ...
        as.matrix(read.csv("experiments_no_stopping_no_baseline.csv",sep=";",header=TRUE))
 5  > boxplot(log_likelihood, ylab="Log-likelihood")
 6  > friedman.test(log_likelihood)
 7
 8      Friedman rank sum test
 9
10  data:  log_likelihood
11  Friedman chi-squared = 80.486, df = 5, p-value = 6.641e-16
12
13  > posthoc.friedman.nemenyi.test(log_likelihood)
14
15      Pairwise comparisons using Nemenyi multiple comparison test
16              with q approximation for unreplicated blocked data
17
18  data:  log_likelihood
19
20              Baseline Dropout Quasi.Dropout Bagging B.QD
21  Dropout      0.04674  —       —             —       —
22  Quasi.Dropout 0.00142 0.91341 —             —       —
23  Bagging      8.2e-08  0.03657 0.37525       —       —
24  B.QD         0.00024  0.70448 0.99829       0.65049 —
25  BagDrop      5.9e-14  3.6e-07 7.4e-05       0.09242 0.00049
26
27  P value adjustment method: none
```

C-2-2   Appendix C.2.B: Without Early Stopping without Baseline

Listing C.2: R output

```
1  > require(stats)
2  > library(PMCMR)
3  >
4  > log_likelihood = ...
       as.matrix(read.csv("experiments_no_stopping_no_baseline.csv",sep=";",header=TRUE))
5  > boxplot(log_likelihood, ylab="Log-likelihood")
6  > friedman.test(log_likelihood)
7
8       Friedman rank sum test
9
10 data:  log_likelihood
11 Friedman chi-squared = 52.68, df = 4, p-value = 9.942e-11
12
13 > posthoc.friedman.nemenyi.test(log_likelihood)
14
15      Pairwise comparisons using Nemenyi multiple comparison test
16               with q approximation for unreplicated blocked data
17
18 data:  log_likelihood
19
20               Dropout Quasi.Dropout Bagging B.QD
21 Quasi.Dropout 0.7514  -             -       -
22 Bagging       0.0042  0.1448        -       -
23 B.QD          0.4339  0.9874        0.3735  -
24 BagDrop       4.1e-10 6.6e-07       0.0166  9.5e-06
25
26 P value adjustment method: none
```

C-2-3   Appendix C.2.C: With Early Stopping

The results of the experiments are visualized in figure (C-1).

From visual inspection we can see that the BagDrop procedure has no better results than the baseline and the Quasi-Dropout procedure. However, BagDrop still differs significantly from the other methods. As the Kruskal-Wallis test indicates significance ($\chi^2_5 = 100.86, p = 2.2e-16 < 0.05$) for the experiments with Early Stopping, it is meaningful to conduct multiple comparisons in order to identify differences between the procedures. The Nemenyi test p-values can be found in table (C-1).

Table C-1: Nemenyi test pairwise p-value table for experiments with Early Stopping

	Baseline	Dropout	Quasi-Dropout	Bagging	Bagging + QD
Dropout	2.9e-08	-	-	-	-
Quasi-Dropout	0.80269	5.5e-12	-	-	-
Bagging	1.4e-05	0.88210	1.0e-08	-	-
Bagging + QD	7.4e-05	0.70448	8.2e-08	0.99942	-
BagDrop	0.65049	0.00011	0.05919	0.00944	0.02837

According to the Nemenyi post-hoc test for multiple joint samples without the baseline experiment, BagDrop log-likelihood differs highly significant ($p < 0.01$) from Dropout, Bagging and Bagging+QD. But as we have seen in the boxplot, the baseline and Quasi-Dropout have way better mean log-likelihoods than BagDrop.

In short, early stopping is one of the standards when using Neural Networks. We tuned the

Listing C.3: R output

```
1  > require(stats)
2  > library(PMCMR)
3  >
4  > log_likelihood = ...
       as.matrix(read.csv("experiments_stopping.csv",sep=";",header=TRUE))
5  > boxplot(log_likelihood, ylab="Log-likelihood")
6  > friedman.test(log_likelihood)
7
8       Friedman rank sum test
9
10 data:  log_likelihood
11 Friedman chi-squared = 87.2, df = 5, p-value < 2.2e-16
12
13 > posthoc.friedman.nemenyi.test(log_likelihood)
14
15      Pairwise comparisons using Nemenyi multiple comparison test
16               with q approximation for unreplicated blocked data
17
18 data:  log_likelihood
19
20                      Baseline    Dropout Quasi.Dropout Bagging ...
                           Bagging.Quasi.Dropout
21 Dropout               2.9e-08     -       -             -       -
22 Quasi.Dropout         0.80269     5.5e-12 -             -       -
23 Bagging               1.4e-05     0.88210 1.0e-08       -       -
24 Bagging.Quasi.Dropout 7.4e-05     0.70448 8.2e-08       0.99942 -
25 BagDrop               0.65049     0.00011 0.05919       0.00944 0.02837
26
27 P value adjustment method: none
```

hyperparameters in such a way that the baseline model overfits on the training data. But when we add an early stopping rule to the algorithm, the baseline experiment will not overfit anymore. It looks like the additional procedures we proposed are only makes the algorithm worse. But still, the BagDrop procedure has much better generalization performance compared to the procedures Dropout, Bagging and Bagging + Quasi-Dropout.
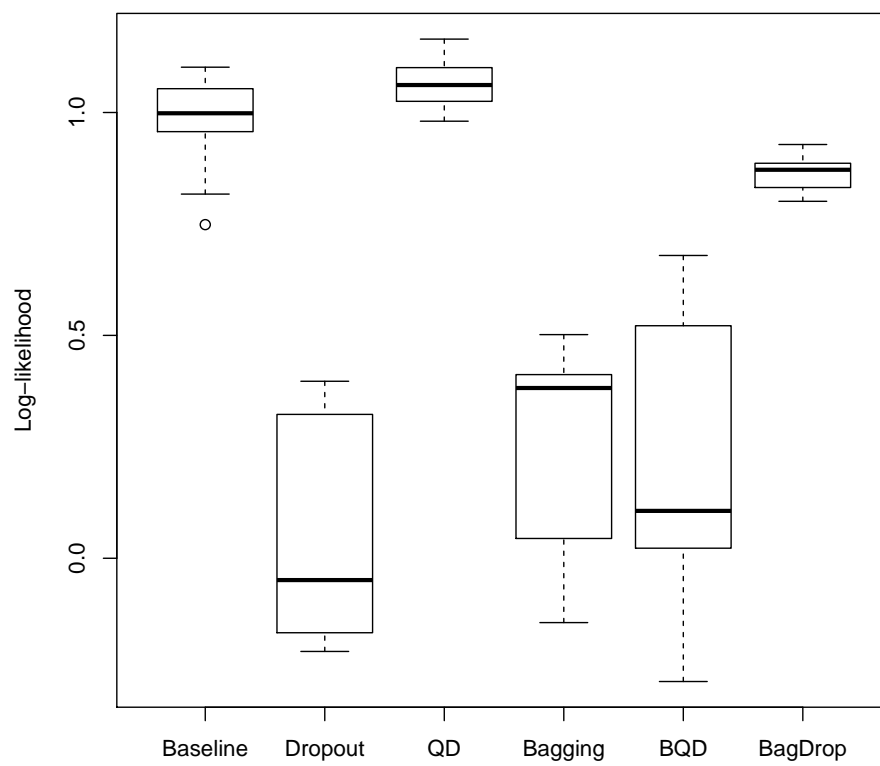
Figure C-1: Boxplot of Experiments with Early Stopping

Appendix D: Background Theory

D-1 Appendix D.1: Stochastic Gradient Descent

The updates of on-line gradient descent, also known as *stochastic gradient descent*, makes use of the fact that loss functions are based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point

$$L(\theta) = \sum_{n=1}^{N} L_n(\theta) \tag{D-1}$$

We often minimize the loss function on the training data mentioned above. However, we should really minimize the expected loss on all data.

$$\mathbb{E}\left(L(\theta)\right) = \int Pr^*(X = \mathbf{x}, Y = y) L(\theta) \; d(\mathbf{x}, y) \tag{D-2}$$

Now if we take the gradient

$$\nabla_\theta \, \mathbb{E}\left(L(\theta)\right) = \mathbb{E}\left(\nabla_\theta \, L(\theta)\right) \tag{D-3}$$

To approximate this gradient, we can use Monte Carlo methods. If we take S independent samples $\mathbf{x}, y \sim X, Y$ and we get a set of samples $\{\mathbf{x}^{(s)}, y^{(s)}\}$ with $s \in \{1, ..., S\}$, then we can approximate the gradient of the loss with

$$\nabla_\theta \, \hat{L}(\theta) = \frac{1}{S} \sum_{s=1}^{S} \nabla_\theta \, L(\theta, \mathbf{x}^{(s)}, y^{(s)}) \tag{D-4}$$

The samples taken from the distribution are in this case the training samples in the training set. Now in case we want an exact update rule for the weights, we should take a step in the direction of the negative gradient of the expected loss. However, we just showed that we can approximate this by Monte Carlo methods. Instead of using a whole set of training samples, we can just use one sample (or a mini-batch of samples) at the time. The size of such a mini-batch is called the *batch size* and is usually fixed. Now we can make an update to the weight vector based on sample (or a mini-batch of samples) at the time, so that

$$\theta^{(\tau+1)} = \theta^{(\tau)} - \lambda \nabla_\theta \, \hat{L}(\theta) \tag{D-5}$$

D-2   Appendix D.2: Generalization Error

The test error, also referred to as the *generalization error*, is the prediction error over an independent test sample (X, Y), where X and Y are drawn randomly from their joint distribution $Pr^*(X, Y)$, and *not* from the training sample S. The generalization error of some input point X is

$$Err_S(X) = \mathbb{E}(L(Y, M_{\theta|S}(X))) \tag{D-6}$$

Note that the test error refers to the error for this specific training set S. We also want to incorporate the fact that the training set could be random. So a more interesting measure is the expected test error of some input point X

$$Err(X) = \mathbb{E}(L(Y, M_{\theta|S}(X))) = \mathbb{E}(Err_S(X)) \tag{D-7}$$

This expectation averages also over the training set S that produced $M_\theta(\cdot)$. Besides the generalization error, we have the training error, which is defined as

$$Err_S(X) = \mathbb{E}(L(Y_{\text{train}}, M_{\theta|S}(X_{\text{train}}))) \tag{D-8}$$

where $(Y_{\text{train}}, X_{\text{train}})$ are random samples from the training sample S.

We also have the notion of a validation sample V, which is different from the training sample S and is used to differentiate between models and select the model that has the best performance. We also have the validation error, which is defined as

$$Err_S(X) = \mathbb{E}(L(Y_{\text{val}}, M_{\theta|S}(X_{\text{val}}))|S) \tag{D-9}$$

where $(X_{\text{val}}, Y_{\text{val}})$ are random samples from the validation sample V.

D-3   Appendix D.3: Bias-Variance Trade-Off

The goal of getting a model that is sufficiently flexible to capture the particular characteristics of the data is at odds with the goal of finding a function that does not overfit. If we keep functions simple enough to avoid overfitting we may introduce a bias in our predictions. However, letting the function be overly complex can lead to overfitting and thus higher variance in predictions. This is known as the bias-variance tradeoff.

Now, let's take a look at the bias-variance decomposition when we have continuous scalar Y as target. If we assume that $Y = \mu_\theta(X) + \epsilon$, with $\epsilon \sim N(0, \sigma^2)$, we can derive a expression of the fit $\mu_{\hat{\theta}}(X)$ at an input point $X = \mathbf{x}$ where $\hat{\theta}$ is the trained parameter. Now, using the squared error loss, the expected test error becomes

$$\begin{aligned} Err(\mathbf{x}) &= \mathbb{E}((Y - \mu_{\hat{\theta}}(\mathbf{x})^2)|X = \mathbf{x}) \\ &= \sigma^2 + (\mathbb{E}(\mu_{\hat{\theta}}(\mathbf{x})) - \mu_\theta(\mathbf{x}))^2 + \mathbb{E}(\mu_{\hat{\theta}}(\mathbf{x}) - \mathbb{E}(\mu_{\hat{\theta}}(\mathbf{x})))^2 \\ &= \sigma^2 + Bias^2(\mu_{\hat{\theta}}(\mathbf{x})) + Var(\mu_{\hat{\theta}}(\mathbf{x})) \end{aligned} \tag{D-10}$$

The first term is the variance of the target variable Y around its true mean $\mu_\theta(\mathbf{x})$, which often cannot be avoided no matter how well we estimate $\mu_\theta(\mathbf{x})$. The second term is the squared bias, the amount by which the average of our estimate differs from the true mean. The last term is the variance, the expected squared deviation of our estimate around its mean.

BagDrop

D-4   Appendix D.4: Early Stopping Rule

With the usage of neural networks an *early stopping* rule is commonly used to avoid overfitting (Goodfellow et al., 2016). When we have a neural network with sufficient representational capacity to overfit, we often observe that training error, decreases steadily over time, and the validation error decreases, but begins to rise again at a certain point. This behavior occurs very reliably (Goodfellow et al., 2016).

We could obtain a model with better validation error by returning to the parameter setting at the point in time with the lowest validation error at the end of training. Early stopping does not run the whole training till its is finished but stops training if the validation error has not improved for some amount of time. Every time the validation error improves, we store a copy of the model parameters. When training terminates, we return these parameters.

D-5   Appendix D.5: Quasi-Dropout

Secondly, we will describe a different way to apply Dropout then described in (5-4). Our goal is to create an ensemble of neural networks in which we can manage the predictions of all the ensemble components separately. One way to do this, is to randomly generate and fix B Dropout masks $M^{(b)}$ with $b \in \{1, .., B\}$ from a neural network architecture before training. Do this by dropping out hidden units by a chance of p. Now the training procedure is the same as in (6-1-1), except for the fact that the training samples (X, Y) are independent of the chosen mask M. Therefore, again, we can apply Monte Carlo by sampling $\mathbf{x}, y, m \sim X, Y, M$ and approximate the gradient of the expected loss by (6-1). This way, we can train multiple neural network architectures jointly. One expected difference when removing bagging from the BagDrop procedure, is the fact that the predictions of the ensemble components will have lower variance but higher bias.

D-6   Appendix D.6: Kruskal-Wallis test

The Kruskal-Wallis test is very simple. Let us have k learning algorithms and E experiments per algorithm. Let x_i^j be the log-likelihood of the j-th algorithm on the i-th experiment. For each experiment i, we rank the log-likelihood such that for each x_i^j we get a corresponding rank r_i^j. Now, find the values

$$
\begin{aligned}
\overline{r_j} &= \frac{\sum_{i=1}^{E} r_{ij}}{E} \\
\overline{r} &= \frac{1}{2}(kN + 1) \\
Q &= (kE - 1)\frac{\sum_{j=1}^{k} E(\overline{r_j} - \overline{r})^2}{\sum_{j=1}^{k} \sum_{i=1}^{k} E(r_{ij} - \overline{r})^2}
\end{aligned}
\tag{D-11}
$$

The Kruskal-Wallis statistic is given by Q. Note that the value of Q as computed above does not need to be adjusted for tied values. The $null-$hypothesis H_0 says that the algorithms are equivalent, so the ranks should be approximately equal. In fact, if the $null$-hypothesis is true, the Kruskal-Wallis statistic Q follows the chi-square distribution χ^2 with $k-1$ degrees of freedom, when E and k are large enough. The p-value is approximated by

$$
p \approx Pr(\chi_{k-1}^2 \geq Q)
\tag{D-12}
$$

BagDrop