76107 (errors:1) characters (not including spaces)

16381 (errors:1) words

University of Southern Denmark, Odense
The Faculty of Business and Social Sciences

Date 03.06.2024
Number of Characters: 82.164

Optimization Heuristics for Binary Neural Networks

Operations Research

Supervisor:
Marco Chiarandini
Department of Mathematics and Computer Science

The project is developed by:
Mads Skjøth Bruhn
23.03.1999, Matematik-Økonomi, 10. Semester

”Det erklæres herved på tro og love, at undertegnede egenhændigt og selvstændigt har udformet denne rapport. Alle citater i teksten er markeret som sådanne, og rapporten eller dele af den har ikke tidligere været fremlagt i anden bedømmelsessammenhæng.”

Mads Skjøth Bruhn
Mads Skjøth Bruhn

03-06-2024
Date

Abstract

Abstract in English...

Resume

Resume in Danish....

Table of Contents

1 Thesis Statement and Structure of Thesis

The aim of this thesis is to answer the Thesis Statement: *Is it possible to train binary and ternary neural networks using local search*? To answer this question, the following sub questions are considered:

- What is the best strategy for training discrete neural networks using local search?
- Is the local search training scalable with more data and larger networks?
- How does training binary and ternary neural networks using local search compare to the existing literature?

Before answering these questions, I first provide a literature review to summarize some of the most important studies related to training binary and ternary neural networks. Before introducing the local search model, I briefly present the most important machine learning theory. Having introduced the local search model I present how the local search works and how the discrete nature of the networks are utilized efficiently. I then outline the algorithms, that I have found to be the most promising. Finally, I present results for these algorithms, compare them and discuss limitations as well as future work.

2 Introduction

3 Literature Review

Deep learning (DL), a subset of machine learning (ML), has proven to be successfull in a large variety of tasks. Since Krizhevsky et al. (2012) won the ImageNet competition using deep convolutional neural networks (CNNs), the research and interest in DL has exploded. In the following years several important advances were made within DL research. The introduction of generative adversarial networks (GANs) by Goodfellow et al. (2014) revolutionized generative artificial intelligence (AI) and Google's DeepMind team developed AlphaGo using deep reinforcement learning (RL). Their agent defeated a world champion Go player, demonstrating the power of combining deep learning with other AI techniques. One of the most influential papers, 'Attention is All You need' by Vaswani et al. (2023), which introduced the transformer architecture. Until this, sequence-to-sequence models relied on recurrent neural networks (RNNs) and their variants like Long Short-Term Memory (LSTM), which processed sequences sequentially. As a consequence the training and inference were slow and difficult to parallelize. The transformer architecture is built upon a self-attention mechanism and it proved to be an important ingredience in natural language processing models. Beyond the already mentioned tasks where DL is used, it is also used in speech recognition, finance and healthcare.

Before the introduction of binary neural networks (BNNs) by Hubara et al. (2016) deep neural networks (DNNs) were primarily trained on graphic processing units (GPUs) using stochactic gradient descent (SGD) methods on enormous amounts of training data, which makes the training process computationally demanding. Hubara et al. (2016) introduced BNNs in which the weights and activations are restricted to the values $+1$ and -1 in order to come up with DNNs that could run on low-power devices. The restriction on the weights and activations makes it possible to represent them in 1-bits as 0 can be interpreted as -1. This means that the normal 32-bit floating points can be replaced by single bits, greatly reducing the memory usage. Another consequence of this is that the computationally demanding 32-bit floating point multiplication can be replaced by 1-bit XNOR-count operations, which could have a big impact on dedicated hardware.

Since Hubara et al. (2016) showed that BNNs could achieve state-of-the art results, much research has been conducted within the area. Large-weight models such as the AlexNet model, that won the ImageNet competition (Krizhevsky et al., 2012), contains 60 million floating-point parameters and 650,000 neurons, making it difficult to deploy on devices with limited ressources. Yuan and Agaian (2023) gives a great comprehensive review of BNN research and note that 1-bit values can theoretically have 32 times lower memory storage and 58 times faster inference speed than traditional 32-bit CNNs. Most of the research within BNNs is still concentrated within traditional NN training using SGD. In their review, Yuan and Agaian (2023) divide the optimization solutions of BNNs into 5 areas. Quantization error minimization are methods that try to minimize the information loss during the transformation from 32-bit values to 1-bit values. Another area is loss function improvement, where a special loss or regularization can be added. A third area of research is gradient approximation, where researchers try to solve the zero derivative issue. A different way to optimize BNNs is to optimize BNN by optimizing the network architecture. The remaining area is called training strategy and tricks in their paper. In this thesis I will not dive deeper into those threads of research, but will instead attack the problem of optimizing BNNs from a different angle.

The introduction of BNNs is interesting not only for the ML community, but also for the operations research (OR) community. Training BNNs can be seen as a discrete optimization problem. BNNs are mostly trained using traditional SGD methods, but where the weights are binarized in the forward pass. It is possible, however, to train BNNs using constraint programming (CP) and mixed-integer programming (MIP). These model-based approaches have stronger convergence guarantees than Gradient Descent (GD) (Toro Icarte et al., 2019), but face other problems. A particular constraint is that MIP-models do not scale to large datasets as the model size depends on the size of the training set. Further, solutions found by MIP and CP that are proved to have optimal training error are likely to overfit the data and does not generalize well.

To deal with the problem of scalability, the research in training BNNs using model-based approaches are mostly focused on few-shot learning, where the size of the training set

is very limited. This is itself interesting because there might be cases where the collection of large datasets are either impossible or very costly. Such cases could arise in areas as healthcare, where there in some cases is a limited amount of labelled data available (Min et al., 2018). It should be noted, that there is no consensus on the definition of a BNN. Some use the term for NNs in which the weights and activations are restricted to +1 and -1 and this is the notation that I will use as well. Others call it a BNN even though they allow weights to have a value of 0, meaning that the corresponding link can be removed from the network. This is also called a ternary neural network (TNN), which I will use to distinguish between these two types of networks.

Toro Icarte et al. (2019) show that their TNNs correctly classify up to 3 times more unseen examples compared to TNNs learned by GD when trained with few-shot learning. As a baseline GD model they extend the model from Hubara et al. (2016), such that it also allows weights to be zero. To adress the problem of overfitting on the training data, the models developed have objective functions that encourage simplicity and robustness, as these are two well-known ML principles for generalization. In the work of Toro Icarte et al. (2019), they force the training set to be correctly classified by introducing constraints ensuring this. As a result they can introduce objective functions that do not need to take into account how to maximize the training accuracy as this are automatically ensured by the constraints. As a measure for simplicity, they use the number of active connections, i.e. weights with values different from zero, which they try to minimize. Measuring robustness is somewhat more difficult, but they model this by the margins in each neuron. The margin of a neuron is defined to be the minimum absolute value of its preactivation. Larger margins require bigger changes on their inputs and weights to change the activation values and therefore help making the network robust.

They introduces three CP models - one without an robustness objective function, one that tries to minimize the number of connections in the network and one that seek to maximize the margins in the network. Two MIP models with these robustness objectives are also introduced as well as four hybrid models that combine the two approaches. For the hybrid models CP was used to initially find a feasible solution, which could then

be given to the chosen MIP model. They evaluated the performance on the well-known benchmark dataset MNIST, in which the task is to classify digits. The used a balanced training set of 1 to 10 examples for each of the 10 classes. As such the size of the training set is ranging from 10 to 100. Even though they used a time-limit of two hours, their MIP models struggle to find solutions. Their best performing model, one of their hybrid models, reaches an testing accuracy of 56 % with only 100 training examples and 2 hidden layers, each with 16 neurons.

Thorbjarnarson and Yorke-Smith (2023) see a potential in training NNs with MIP solvers. Unlike traditional state-of-the-art methods to train NNs, which require significant data, GPUs and extensive hyper-parameter tuning, MIP solvers do not need GPUs nor the same amount of hyper-parameter tuning. They recognize, however, that MIP models only can handle small amounts of data. For this reason, they do not expect NN training with MIP solvers to be competitive with gradient-based methods. According to them, the potential in MIP solvers is to train smaller networks with small batches of data.

The models trained by Thorbjarnarson and Yorke-Smith (2023) are integer neural networks (INNs), in which the weights can take any integer value in the interval $\{-P, \ldots, P\}$. Their base model are built upon the model from Toro Icarte et al. (2019) and are then modified to test three different models - max-correct, min-hinge and sat-margin. While max-correct only seek to maximize the number of correct predictions of training samples, it does not aim to make the predictions as confident as possible. Max-correct simply optimizes a sum of binary variables, where each variable indicate if the corresponding training sample is correctly predicted. The min-hinge model also aim to make correct predictions, but here the objective function is made in such a way that an optimal solution makes confident predictions. The min-hinge is inspired by the squared hinge loss, and in their model they approximate it using a piecewise linear function. Their last model, sat-margin, combines aspects from both models. Again, it maximizes a sum of binary variables, but contrary to max-correct, the binary variables can only be 1 if the prediction is above a certain margin.

Besides from solving the classification problem, they also aim to find the minimum number of neurons needed in the NN to fit the training data. To do this, they propose yet another objective function, which can be added to their max-correct and sat-margin model. The model compression objective function is to minimize a sum of binary variables, one for each neuron in all the hidden layers. This binary variable can only be 1 if all the weights going into the neuron and all the weights going out of the neuron are 0. This is a way of compressing the model and it is an advantage of using discrete optimization solvers, that makes it possible to simultaneously train the NN and optimize its architecture.

Another important contribution of their work is their batch training algorithm that increases the amount of data that can be used to train their NN significantly. The way it works are, in short terms, as follows: Start by distributing training data into small batches and train a MIP NN model on each batch independently of each other. When a model for each batch has been trained, all the NNs are combined into a single NN, where validation accuracies are used as weights to make it a weighted average. To ensure convergence, they then constrain the bounds of the weights before they repeat the process, which is then repeated until convergence. After convergence the model with the highest validation accuracy is chosen as the solution. It should be noted that the training of the MIP NNs in each iteration can be done in parallel.

Maria Bernardelli et al. (2023) introduce yet another approach to train both TNNs and INNs. Instead of training a single model for a classification problem, they use an ensemble approach, which is based on training a single NN for each possible pair of classes. After training, a majority voting scheme is used to predict the final output. Further, in the training of a NN, they suggest to train it by solving a lexicographic multi-objective MIP model. Their multi-objective model is actually quite intuitive. As a starting point, they start training their model to maximize the number of confidently correctly predicted data. The solution found in that model is then used as a warm start in a model that maximizes the margins to make the network more robust. Finally, the solution of that model is used as a warm start in a model that minimizes the number of connections. Notice, that the models in a way inherit constraints, such that the second

model must still be able to predict the training data confidently and the third model must satisfy the margins that the second model maximized.

Their ensemble approach is quite different from the other approaches. Instead of training a single network with a neuron in the last layer for each possible label, they need to train a network for each pair of classes. For each pair of classes they then solve a binary classification problem. A sample from the test dataset is then fed into all of the networks and a majority voting system is then used to determine the final output. For the MNIST dataset and with a balanced training dataset of 100 samples, their average testing accuracy is at 68 %. For this experiment they trained each of their NNs for 160 seconds, given a total training time of two hours, which could be greatly reduced in wall-clock runtime by training their networks in parallel. As architecture for their NNs they use the structure $[784, 4, 4, 1]$ or $[784, 10, 3, 1]$. They also investigate how their approach scales with the number of training images. As such they train their NNs on also 200, 300 and 400 training samples, but this time with 600 seconds allocated to each NN, which gives a total wall-clock runtime of 7.5 hours, when trained one by one. Their testing accuracy increases to a maximum of 81 % with 400 training samples.

4 Machine Learning Theory

ML is a branch of AI, where the AI systems have to acquire their own knowledge, which they do by extracting patterns from raw data. Some of the tasks that ML can help solve are classification, regression, transcription and machine translation. DL is a specific kind of ML. It is easy for AI in general to solve problems that can be described by some well-defined mathematical rules. However, initially AI struggled to do well on tasks that were easy for humans to perform. An example is object recognition in images. While a human, in most cases, easily can distinguish between dogs and cats, the problem is not so easy to translate into mathematical rules, that a computer can understand. DL use NNs, which often consist of many layers, which give rise to the name 'deep' learning. In this thesis, I will focus on solving the classification problem by training a NN in an untraditional way compared to standard NN training.

4.1 Feedforward Neural Networks

Feedforward NNs are also called multilayer perceptrons (MLPs) and they are one of the simplest NNs. In terms of the classification problem, the goal of a feedforward network is to map an input $\mathbf{x}$ to a class y. As such, a MLP defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$. The goal is then to learn the parameters $\boldsymbol{\theta}$ such that it gives the best function approximation. A MLP is represented by composing together different functions. Two functions, $f^{(1)}$ and $f^{(2)}$ can be used fo form $f(\mathbf{x}) = f^{(2)}(f^{(1)}(\mathbf{x}))$. The input is $\mathbf{x}$, $f^{(1)}$ is called the first layer of the network and $f^{(2)}$ the second layer, which in this case is also the last layer of the network called the output layer. During training, each example $\mathbf{x}$ comes with a label $y \approx f^*(\mathbf{x})$ and the goal of the output layer is to produce values, for each $\mathbf{x}$ that is close to the corresponding y. The behavior of the remaining layers are not specified. During training, the network must decide how to use those layers, called hidden layers, to produce the desired output. An example of a feedforward neural network is given in Figure 1. Given an input vector $\mathbf{x}$, the preactivation $s_v^l(\mathbf{x})$ of neuron v in layer l and its activation or output by $a_v^l(\mathbf{x})$ can be represented by:

$$s_v({}^l\mathbf{x}) = \sum_{u \in N_{l-1}} w_{uv}^l \cdot a_u^{l-1}(\mathbf{x}) \quad \text{and} \quad a_v^l(\mathbf{x}) = p(s_v^l(\mathbf{x}))$$

where N_{l-1} is the set of neurons at layer $l-1$ and p is an activation function, which are often an nonlinear function. The default recommendation to use is the rectificied linear unit (ReLU), which is defined as $g(z) = \max(0, z)$, but many other activation functions are possible. The activation function is only applied in the hidden layers. In the input layer, the output is simply equal to the input and in the output layer the preactivation values are often feed into a loss function, that then may send them through a different activation function.

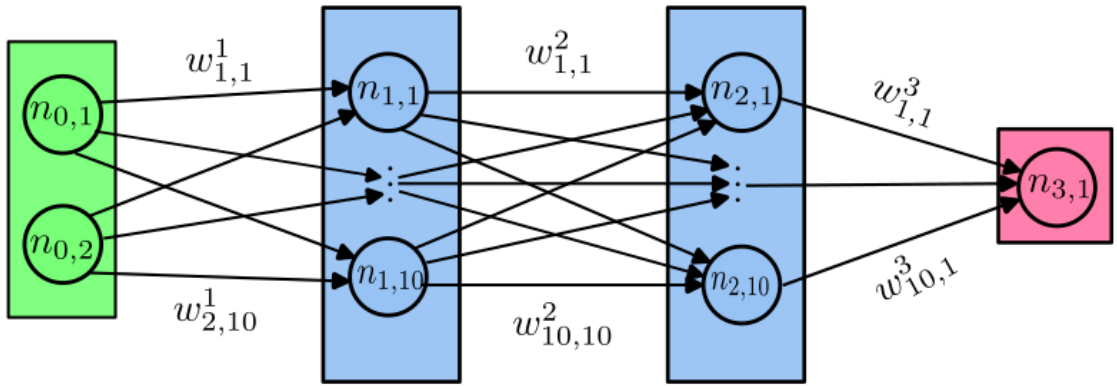

Figure 1: A feedforward neural network with 2 inputs, 2 hidden layers with 10 neurons each, and 1 output neuron. I use the notation n_{lv} to represent neuron v from layer l and w^l_{uv} to represent the weight from neuron u in layer $l-1$ to the neuron v in layer l.

4.2 The Classification Problem

Classification is a supervised machine learning method, in which training samples have class labels and the goal is to find a model such that the model performs well on new unseen test samples. In this thesis, I will work with binary classification, in which the model needs only to distinguish between two class labels, and multi-class classification, where the number of class labels is bigger than two. The most used loss function, or in LS terminology, objective function used to train NNs is the cross-entropy quantity, defined as:

$$H(P, Q) = -\mathbb{E}_{X \sim P} \log Q(x) \tag{1}$$

Intuitively, the cross-entropy measures the distance between two probability distributions, and in ML this is often used in gradient descent algorithms to minimize the loss. P is the true probability distribution and Q is the predicted.

Cross-entropy comes from information theory, but the cross-entropy quantity is actually also closely related to maximum likelihood estimation (MLE). If we denote the actual distribution of the training data by $p_{\text{data}}(\mathbf{x})$ and let $p_{\text{model}}(\mathbf{x};\boldsymbol{\theta})$ be the probability distribution that tries to describe the actual distribution of the training data, then the goal is find $\boldsymbol{\theta}$, so that $p_{\text{model}}(\mathbf{x};\boldsymbol{\theta})$ is as high as possible. Assuming that m examples, $X = \{x^{(1)}, \ldots, x^{(m)}\}$, are drawn, independently, from the true but unknown data generating distribution $p_{\text{data}}(\mathbf{x})$, the MLE estimator for $\boldsymbol{\theta}$ is defined as:

$$\begin{aligned}\boldsymbol{\theta}_{\text{ML}} &= \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(X;\boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\mathbf{x}^{(i)};\boldsymbol{\theta})\end{aligned}$$

where the rewriting is allowed because of the assumption of independence. This is known as the likelihood function. The goal is to maximize it, and by taking the logarithm of it, the arg max is not changed, but it transforms the product into a sum.

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)};\boldsymbol{\theta})$$

The arg max does not change when rescaling, so dividing by m, we obtain a version of the estimate that is expressed as an expectation with respect to the empirical distribution, $\hat{p}_{\text{data}}$ defined by the training data:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{X\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x};\boldsymbol{\theta})$$

Maximizing this is equivalent to minimizing equation (1), where $p_{\text{model}}(\mathbf{x};\boldsymbol{\theta})$ is the predicted probabilities and $\hat{p}_{\text{data}}$ is the true probability distribution. Normally in supervised learning, we are interested in the case of predicting $\mathbf{y}$ given $\mathbf{x}$, so the goal is to estimate the conditional probability $P(\mathbf{y} \mid \mathbf{x};\boldsymbol{\theta})$. Denoting by $\mathbf{X}$ all the inputs and $\mathbf{Y}$ all the targets, then the conditional MLE estimator is:

$$\begin{aligned}\boldsymbol{\theta}_{\mathrm{ML}} &= \arg\max_{\boldsymbol{\theta}} P(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta})\end{aligned}$$

4.2.1 Binary Classification Example

For binary classification, only a single neuron is needed in the output layer. Here the training samples can be labeled by '0' and '1' respectively and the preactivation value at the neuron in the output layer can then be used to calculate the probability that the training sample is labeled as '1', denoted by $\hat{y}$, and obviously the probability that it is labeled as '0' is then given by $1 - \hat{y}$

Suppose we are in a setting where we train a binary classifier and we have a training sample labeled '1', $y = 1$, and its preactivation value at the output neuron is -2. We want to know what the binary cross entropy loss is. As a first step, we need to figure out what the probability of $y = 1$ is. To do this, we use the logistic function to get the predicted probability, $\hat{y}$:

$$\hat{y} = \frac{1}{1 + \exp(-2)} \approx 0.1192$$

To get the loss, we use the formula from (1):

$$H(P, Q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) = -\log 0.1192 \approx 2.1269$$

The loss is quite large since the correct label is '1', and the preactivation value is negative. In binary classification, whenever the label is '1', the preactivation value should be as large as possible to get the highest predicted probability.

4.2.2 Multi-class Classification

The example and intuition above generalizes to multi-class classification, however now the number of neurons in the output layer need to be equal to the number of classes. In this case, we cannot use the logistic function anymore to get the predicted probability. Instead we take the vector of preactivation values from the output layer and apply the softmax function to it, which gives a vector of probabilities for each class such that the

probabilities sum up to 1. Formally, given the vector z with preactivation values, this is defined as:

$$\text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{m} \exp(z_k)} \tag{2}$$

where m is the number of classes and the classes are labeled $1, \ldots, m$. The cross-entropy loss, is, however, easier to calculate in this case using (1), as this reduces to the negative of the logarithm of the probability for the right label:

$$H(P, Q) = -\log \frac{\exp(z_y)}{\sum_{k=1}^{m} \exp(z_k)}$$

4.3 Traditional Neural Network Training

Optimization algorithms within ML and DL differ from traditional optimization problems. In ML it is the goal to perform well on some unknown test set, which is in contrast to traditional optimization problems, where solving the problem at hand is the task. To measure the performance of a ML model, the model is tested on a held-out test set. This is used to estimate how well the model generalizes. It can be beneficial to divide the remaining training set into two disjoint sets - a final training set and a validation set. The validation set can be used during training for model selection or hyperparameter selection, but it is not used to train the main parameters of the model. The test set is not seen before after the entire training process and as such it is only seen once, while the validation set can be seen many times.

For classification, we are concerned about predicting as many samples in the test set correctly as possible. To evaluate a classification model a well-known performance measure is the 0-1 loss, which basically returns the error rate, however, this loss-function is not well-suited for optimization tasks neither in traditional NN training nor in a LS context. In traditional NN training, a surrogate loss function is used instead, which could be the cross-entropy loss function introduced earlier. Some reasons for this are that the 0-1 loss is non-differentiable, non-continuous and it does not give as robust a model as when using the cross-entropy loss function. To prevent overfitting, which occur when the training error is small, but the test error is large, it is possible to use

an early stopping technique, where the model trained is tested at certain times on the validation set. Here it is often the 0-1 loss used and the model with the smallest loss is then chosen, even though a later model might have smaller loss when using the surrogate function.

The most common optimization algorithm within ML and especially DL is stochastic gradient descent (SGD). SGD works by sampling a small subset of samples from the training set. Sometimes this is referred to as a minibatch, but herein I will use the term 'batch'. The batch is then used to compute the gradients of the parameters and very small updates are made to the parameters. In algorithm 1 pseudocode of a simple version of SGD can be seen. In practice, the convergence criterion is often replaced by iterating over the entire training set for a certain number of epochs and the learning rate, which is a very sensitive hyperparameter should decrease during the training. Many extensions of this, conceptually simple, algorithm exists. Some of these extensions try to overcome some of the challenges that arise when training NNs. To accelerate learning momentum can be introduced, or to overcome the problem of choosing the correct learning rate, algorithms with adaptive learning rates can be used, which however, introduce more hyperparameters. Early stopping can be applied by testing the model on the validation set after each time the entire training set has been seen, called an epoch.

Algorithm 1 Pseudocode for Stochastic Gradient Descent

Require: Learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

while Convergence criterion is not met **do**

  Sample a batch from the training set

  Compute gradient estimate $\hat{\boldsymbol{g}}$

  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

end while

4.4 Binary Neural Networks

The development in the use of NNs have pushed the limits of AI significantly, however, before the introduction of BNNs the NNs were trained using one or many GPUs. This means that it is a challenge to run NNs on low-power devices. BNNs are networks, where weights and activations are limited to be either -1 or 1. Hubara et al. (2016) showed that it was possible to train BNNs without suffering any loss in classification accuracy. They used the traditional gradient descent approach and then made several tricks to come up with a solution that handles the challenges of training BNNs in this way. Normally, the gradient update in Algorithm 1 is based on the gradients found by backpropagation, but for BNNs the gradient of the sign function used to binarize the weights is zero almost everywhere. Instead they use an estimator for this gradient called the straigt through estimator. With their training approach, they compute the real-valued gradients during the backward pass, and only binarize during the forward pass. How to train BNNs using traditional GD methods have been an active research area and it is without the scope of this thesis to give a full overview. For a more comprehensive review, I refer to Yuan and Agaian (2023).

4.5 Ensemble Methods

In ML, ensemble methods are sometimes used, where several different models are trained before each of them vote on the output for test examples. Among the most known ensemble methods are bagging and boosting. Bagging is a method, where k different datasets are constructed by sampling from the original dataset with replacement. This give k datasets, but each dataset is missing some of the examples from the original dataset but instead contains duplicates. k different models are then trained giving k different models that can then be used to classify by voting. On the other hand boosting is a method where several models are also trained, but on the same dataset. The trick here is to assign new weights to the training examples after each model training, such that when model i has been trained, the weights for model $i+1$ are adjusted, ensuring that misclassified inputs in model i get higher weights in model $i+1$.

In this thesis I also implement the BeMi ensemble introduced by Maria Bernardelli

et al. (2023). Here I only focus on the one-versus-one scheme, compared to their more general notation. If C is the set of classes of the classification problem, this ensemble works by training $\binom{|C|}{2}$ different networks, such that each network learns to distinguish between two different classes from C. The idea is then to afterwards feed the inputs through all networks and if all the networks that have actually been trained on the label of the input classify the input correctly, then at most $\binom{|C|-1}{2-1} - \binom{|C|-2}{2-2}$ other networks can classify the input with a different label, so the ensemble classify it correctly.

4.5.1 The BeMi Ensemble on MNIST

MNIST is a well known dataset, where handwritten digits from 0 to 9 need to be classified. For this classification task, $\binom{10}{2} = 45$ binary classifiers needs to be trained such that a network is trained to distinguish between each pair of digits. When a test example needs to be classified it is given to all 45 networks. If all of the 9 networks, that have actually seen labels as the test example, classify it correctly, then a maximum of $\binom{9}{1} - \binom{8}{0} = 8$ other networks can classify the input with a different label, and so the prediction is correct. In the paper by Maria Bernardelli et al. (2023), they work with 7 different label statuses, but only 2 of these statuses are defined to give the correct prediction. The first one is when, as before, there is a label that gets the most votes and it is the correct one - this gives a correct prediction. The second one, is when two labels gets the same number of votes. In this case, the network used to distinguish between these two labels are used to determine the prediction. If the network used to distinguish between the two labels predicts the correct label, this also counts as a correct prediction.

5 Local Search Model for Binary and Ternary Neural Networks

A local search (LS) algorithm starts from a solution, which may be a random assignment of values to the decision variables. From here it moves from the current solution to a neighboring solution in the hope of improving a function f. A solution is denoted by s, and the set of neighboring solutions of s, $N(s)$, is called the neighborhood of s. In the following I will define the LS model I have used to train BNNs and TNNs and explain how local search is used.

5.1 Notation

A BNN is a network where the weights and activations are restricted to ± 1, whereas allowing the weights to be equal to 0 gives a Ternary Neural Network (TNN). This model applies for fully connected NNs, where the input comes from the data. I denote by $N = \{N_0, N_1, \ldots, N_L\}$ the set of layers in the network where N_0 is the input layer and N_L is the last layer. For each layer, the set of neurons are denoted by $N_l = \{1, 2, \ldots, n_l\}$ such that the width of layer l is n_l. The decision variables are the weights between each layer. The weight between neuron u in layer N_{l-1} and neuron v in layer N_l is denoted by w_{uv}^l. For the hidden layers, the activation function used to binarize the output is given by:

$$p(x) = 2 \cdot \mathbb{I}(x \geq 0) - 1 \tag{3}$$

The training set can be written as $TR = \{(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^T, y^T)\}$ such that $\mathbf{x}^i \in \mathbb{R}^{n_0}$ and y^i is the label of the instance i for every $i \in \{1, 2, \ldots, T\}$. It is beneficial to keep track of the preactivation values for all training instances for all the neurons in the layers $\{N_1, \ldots, N_L\}$. I use s_v^{il} to denote the preactivation for instance i at neuron v in layer l. Similarly, for all the hidden layers $\{N_1, \ldots, N_{l-1}\}$, the output of the activation is denoted by $u_v^{il} = p(s_v^{il})$.

All of the above can also be written in terms of matrices and vectors. The input matrix is then denoted by $X \in \mathbb{R}^{T \times n_0}$ such that every row corresponds to the input of a specific

instance and Y is a vector with the labels. Then the mathematical model for a BNN can be written as:

$$\begin{aligned}
\max \quad & f(S^L, Y) && && (4)\\
\text{s.t.} \quad & U^0 = X && && (5)\\
& S^l = U^{l-1} W_l && \forall l \in \{1, \ldots, L\} && (6)\\
& U^l = p(S^l) && \forall l \in \{1, \ldots, L-1\} && (7)\\
& W^l \in \{-1, 1\}^{n_{l-1} \times n_l} && \forall l \in \{1, \ldots, L\} && (8)\\
& X \in \mathbb{R}^{T \times n_0} && && (9)
\end{aligned}$$

where (3) is applied elementwise. Notice that I maximize a function, which is in contrast to standard ML, where the crossentropy loss is often minimized. I overcome this by multiplying by -1, whenever I use a loss function that is tradtionally minimized. If the input is real-valued, then S_1 is also real-valued but because of the binary activations and weights, S^l is integer-valued for $l \in \{2, \ldots, L\}$. For a TNN, 0 is included to be an option for the W-matrices. Occassionally I will also use the notation S^l_v, which is a vector of the preactivation values in layer l, but only for neuron v.

5.2 Solution Generation, Neighborhood and Move

As earlier mentioned, the decision variables are the weights between the layers. The preactivations and activations are determined by the input data and the weights. The number of variables in the model are thus given by $k = n_0 \cdot n_1 + n_1 \cdot n_2 + \ldots + n_{L-1} \cdot n_L$. This means that the number of possible solutions for a TNN is 3^k and for a BNN it is 2^k. To generate a solution, I use a uniform, random assignment of the allowed values to the variables. I use a 1-exchange neighborhood such that the neighbors to a solution s are the solutions s' in which it is only a single weight that has another value. This gives a neighborhood of size k for a BNN and $2k$ for a TNN. A move is thus defined as the operation that take a solution s to a new solution s' by changing the value of exactly one weight. For a BNN this is simply a 'flip' from -1 to 1 or from 1 to -1, whereas for a TNN there is always two new values that a weight can take.

Often the number of weights in a NN are very large, and even though I only work with relative small NNs the number of weights quickly grows large. As a result, I found it to be inefficient to implement a function that tests the effect of a single move. For this reason I developed another approach, in which I evaluate several moves at once. I will elaborate on this later with an example.

5.3 Objective Functions

As discussed earlier, when training NNs surrogate loss functions are often used, because the performance measure one is often interested in is non-differentiable, non-continuous and it might not be flexible enough. From now on I will denote the function to be maximized as an objective function, and whenever this corresponds to a loss function, which are normally minimized, I simply multiply the objective function value by -1 to get a maximization problem. In ML the objective function is used to compute gradients, that can be used to update the parameters in a gradient descent algorithm. In a LS context, an objective function does not necessarily have to be differentiable. In my implementation I support three different type of objective functions.

5.3.1 Cross Entropy Objective Function

I have already introduced the cross entropy function earlier. As I have implemented everything as a maximization problem, the objective function becomes:

$$\max \sum_{i=1}^{m} \log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) \tag{10}$$

where m is the number of samples in the batch, $\mathbf{y}^{(i)}$ is the label of sample i with input data $\mathbf{x}^{(i)}$. Since the probabilities are always between 0 and 1, and the logarithm takes each probability and map them to a value between $(-\infty, 0)$, the range of this objective function is $(-\infty, 0)$.

5.3.2 Integer Max Margin Objectice Function

The cross entropy objective function work with real-valued numbers. It might be beneficial to work with an integer-valued objective function for memory effiency reasons.

As such I propose what I call the Integer Max Margin objective function. It builds on exactly the same intuition as the others, which for the multiclassification problem is to maximize the preactivation value for the neuron corresponding to the correct label. As earlier let i denote a training instance and let y^i be the true label, and let $\hat{y}^i$ be the predicted value defined as:

$$\hat{y}^i = \max_{v \in \{1,\dots,n_L\}} s_v^{iL} \tag{11}$$

The goal is to stay within integers and at the same time predict confidently. One way to do this is to maximize the distance between the true label and the label with highest preactivation value, corresponding to highest probability, that are not the true label. So if $\hat{y}^i \neq y^i$ then $s_{\hat{y}^i}^{iL} > s_{y^i}^{iL}$ and the contribution to the objective function in this case will be $s_{y^i}^{iL} - s_{\hat{y}^i}^{iL}$ which is negative. The first goal would then be to minimize the gap between the wrongly prediction and the true label. On the other hand, if $\hat{y}^i = y^i$, then the model predicts correctly and in this case the goal is to maximize the distance to the label that come closest. This term can be written as: $s_{y^i}^{iL} - \max_{v \in \{1,\dots,n_L\} \setminus y^i} s_v^{iL}$. This can be summarized into a single line, so that the objective function, for a single instance, is given by:

$$f(\mathbf{x}^i, y^i) = s_{y^i}^{iL} - \max_{v \in \{1,\dots,n_L\} \setminus y^i} s_v^{iL} \tag{12}$$

As a result, the objective function for all m samples in a batch becomes:

$$\max \sum_{i=1}^{m} \left[s_{y^i}^{iL} - \max_{v \in \{1,\dots,n_L\} \setminus y^i} s_v^{iL} \right] \tag{13}$$

The range of this objective function is theoretically given by $[-m \cdot n_{L-1}, m \cdot n_{L-1}]$. For the binary classification problem, this does not work as there is only one neuron at the final layer. The bounds for the preactivation values are determined by the number of neurons in the second last layer, n_{L-1}, such that the bounds are given by $[-n_{L-1}, n_{L-1}]$. For binary classification problems, the labels can also be encoded as -1 and 1. For the samples labeled 1, the goal is to get the preactivation values as close to n_{L-1} as possible, or equivalently, maximize the distance to $-n_{L-1}$. The goal is opposite for the samples

labeled -1, here the goal is to get the preactivation values as close to $-n_{L-1}$ as possible. For a single instance, this can be summarized into:

$$f(\mathbf{x}, y) = y \cdot s^L + n_{L-1} \tag{14}$$

To see this, take the case of $y = 1$, then we want to maximize $s^L - (-n_{L-1})$ which is equivalent to (14). On the other hand, if $y = -1$, then we want to maximize the distance from s^L to n_{L-1}, which is the same as $n_{L-1} - s^L$. Again, when multiplying s^L with y this is equivalent to (14). For all m samples, the objective function is thus given by:

$$\max \sum_{i=1}^{m} \left[y^i \cdot S^{iL} + n_{L-1} \right] \tag{15}$$

5.3.3 Comparison Between Cross Entropy and Integer Objective Function

To illustrate the difference between this objective function and the crossentropy, suppose we are in a setting with 5 different classes, the correct label is the first label and the preactivation values for the 5 neurons in the output layer is given by $[4, 2, -2, -2, -2]$, such that the contribution to the crossentropy objective function is -0.133456 (after multiplying by -1). The contribution to the integer objective function is 2. Clearly, we are most interested in decreasing the preactivation value for the neuron with value 2, as this would mean a more confident (and correct) prediction of our instance. However, suppose we find a move that decreases the preactivation value of the third neuron from -2 to -4. This gives a new contribution from the crossentropy function equal to -0.131558, which is only a very small improvement. On the other hand, this is not an improvement for the integer objective function, where an improvement only is found if either the preactivation value of the first neuron increases or the preactivation value for the second neuron decreases.

5.3.4 Brier Score

So far I have introduced two objective functions, whose range is quite large. As I, later on, want to test whether it is possible to add a regularization parameter to the objective function to minimize the number of connections in a TNN, it is desirable to also have an objective function, which have a more constrained range. The Brier score, which

measures the accuracy of probabilistic predictions, gives this possibility. For binary classification using 0-1 encoding of the labels, the Brier score is defined as:

$$BS = \frac{1}{m}\sum_{i=1}^{m}(p(y^i) - y^i)^2 \tag{16}$$

where $p(y^i)$ is the probability that y^i is 1 and y^i is the actual outcome. This corresponds to the mean squared error, and it gives a value between 0 and 1, where 0 is the best score achievable. I formulate it without taking the mean and convert it to a maximization problem by multiplying with -1, so the objective function for binary classification becomes:

$$\max\sum_{i=1}^{m} -(p(y^i) - y^i)^2 \tag{17}$$

The range of this objective function value is $(-m, 0)$.

For the multiclassification task, the original BS score is defined by:

$$BS = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=1}^{C}(p(y^{ti}) - y^{ti}) \tag{18}$$

where C is the number of classes, $p(y^{ti})$ is the predicted probability for class t for instance i. y^{ti} is 1 of instance i belongs to the t-th class and zero otherwise. Here the range is double, from zero to two. In my implementation I use:

$$\max\sum_{i=1}^{m}\sum_{t=1}^{C} -(p(y^{ti}) - y^{ti}) \tag{19}$$

The range of this objective function is thus $(-2m, 0)$.

5.3.5 Minimizing the number of connections

In a TNN it is possible to add a second term to the objective function measuring how many active connections are in the model, i.e. how many weights are not zero. The idea is, that the model should be as simple as possible to avoid overfitting. In this context simple means having fewer active connections. Thus, the goal of the second objective term is to minimize the number of connections. As I am working with maximization, the second objective term would look like this:

$$-\alpha \sum_{l \in 1,\ldots,L} \sum_{u \in N_{l-1}} \sum_{v \in N_l} \text{abs}(w_{uv}^l) \tag{20}$$

where α is a weight used to penalize the number of connections. The term is negative as it is put into an maximization problem. The choice of α depends on the objective function it is used in, as the weighting of this second term should depend on what values the first term is taking. As such the Brier score objective function is an ideal choice to use in combination with this term, as the value of that is constrained. As an example suppose the number of training samples is given by $T = 1000$ and the number of weights in the neural network is 10000. Assume it is a multiclassification task. Then the objective function value for the Brier score is in the interval $(-2000, 0)$ and the second term would be in the interval $[-10000, 0]$, with $\alpha = 1$ that is. The total objective function value is as a result in the interval $(-12000, 0)$. A too high value of α would simply put all weight values to zero, but at the cost of predicting correctly. But a small value of α, say 0.001 would make the second term to be in the interval $[-10, 0]$, and it would no longer be beneficial to set all weights equal to zero at the cost of predicting correctly.

5.4 Delta Evaluation

A very important part of a LS algorithm is how to select moves to commit to a solution. Often this is done by selecting a possible move, evaluating that move and get its delta value, which is a value telling how the objective function value is affected by that move. Afterwards a threshold value is used to determine if that move should be accepted or not. For maximization problems, the threshold value could simply be that the delta value should be above 0 in order for the move to be accepted. A critical function in a LS algorithm is thus the function that evaluates a move, as this is something that is done many times throughout the solution search. I found it to be highly inefficient to work with a function that evaluates only a single move. As the solution improves, more and more moves need to be tried before finding an improvement and a function that only evaluate a single move is too slow for this. Also, I found that by evaluating several moves, in a structured way, I found several tricks that speed up the evaluation.

The way I evaluate moves is by selecting a neuron in the neural network with weights going into it, so I do not select neurons from the input layer. For this neuron I evaluate the possible moves for the weights going into it. This is done simultaneously, but still independently in the way that I am testing what would happen if a single weight changes value, not what would happen if all of them change values at once. After this evaluation, a sequence of moves will be returned, each with their own delta value, that I can then use to determine what move to take. I will now introduce the most important aspects of this and will with an example show how it works.

5.4.1 Critical Samples

Recall the binary activation function defined in (3). The input to that function is the preactivation values, s_v^{il} and in a LS context where only one-exchange moves are considered, there are values of these preactivation values such that the output of (3) is unchanged regardless which of the weights going into neuron v at layer l change values. The weights can only take on values -1, 1 (0 as well for a TNN), so they can only change their value by either -2 or 2 in a BNN where -1 and 1 are also possibilities in a TNN. Thus, if we denote the maximum output value, or activation value, (in absolute value) from the previous layer by $u_{\max}^{l-1}$, then we can define the set of critical samples for a neuron, which are those samples that can change activation value by making a single move for one the weights going into that neuron, by:

$$\text{critical }{}_v^l = \{i \in \{1, \ldots, T\} : s_v^{il} \geq -2 \cdot u_{\max}^{l-1} \wedge s_v^{il} < 2 \cdot u_{\max}^{l-1}\} \tag{21}$$

I use a general notation to both describe the case for the first hidden layer, where the output from the previous layer are dependent on the input, but from the second hidden layer, it is possible to skip the $u_{\max}^{l-1}$ term, as this is equal to 1 as a result of the binary activation function. The purpose of these 'critical' samples, is that it greatly reduces the number of instances for which the delta evaluation need to be evaluated for. This trick is only used for the hidden layers, as the same does not apply for the output layer.

5.4.2 Forward Propagation

The next trick that is useful when evaluating moves for a neuron is that it can be used to reduce the number of forward propagations. This logic applies both for a hidden layer

and the output layer. It is a bit more complicated for the output layer in the TNN case, but nevertheless it is very useful. After having found the critical samples for a neuron (in a hidden layer), we need to evaluate what would happen if a move was applied to the weights going into the neuron. A naive way to do this would be to apply a forward propagation for all of the weights on the critical samples to see what would happen with the objective function value. But the binary activation function gives a way to do this more efficiently. The only way that something changes for a specific instance is if the activation of that instance changes, in which case we know that it changes it value by -2 or 2, dependent on the value of it before. Thus, we can simulate what would happen if the activations change for all the critical samples and find their 'delta changes' by doing a single forward propagation. Afterwards we can, for each weight we want to calculate the effect of a move for, find out if making the move would change the activations and then the 'delta changes' can be looked up instead of calculating it again.

For the output layer, a similar logic applies. Again a neuron is selected, but this time the critical set is not relevant, as the binary activation function is not used here. But again, for a BNN exactly one of two things happen (assuming the presence of at least one hidden layer): either the preactivation value increases by 2 or it decreases by 2. This comes from the fact that the output from the previous layer is either -1 or 1 and the value of the weight changes by either -2 or 2. Thus, it is again possible to find the 'delta changes' for the samples by simulating what would happen if their values increase by 2 and what would happen if they decrease by 2. Afterwards, for each single weight it can quickly be determined which of the cases a sample is in and the effect can be looked up in the 'delta changes'-tables. This greatly reduces the number of times the function calculating the objective function value needs to be called. For a TNN, it is almost the same, except 4 'delta changes'-tables are needed as the values can also increase and decrease by 1 in this case.

5.4.3 Example of Delta Evaluation

To illustrate how the mechanisms described above works, I have constructed a BNN with the following structure $[784, 4, 4, 4, 10]$. I am training it on a balanced training set of 10 instances from the MNIST dataset. Initially I initialize a solution by randomly

assigning values to all the weights and I am now looking for what move to make in order to improve the current solution. Suppose now that I look at the fourth neuron in the second hidden layer. The preactivations are given in the vector:

$$S_4^2 = \begin{bmatrix} 0 & 4 & 4 & 2 & 2 & 2 & 4 & 2 & -2 & 2 \end{bmatrix}$$

Each element corresponds to a different training sample. Since we are in the second hidden layer, $u_{\max}^1 = 1$ and thus critical $\binom{2}{4} = \{1, 9\}$, as it is only the instances with 0 and -2 as preactivation values that can change activation. Thus, for the remainder of the delta evaluation process for this neuron, it is only necessary to look at these two instances. The first thing needed to do is to simulate what would happen if their activations change. Thus, I start by calculating the effect of this by propagathing through the network. During the process, I always try to do as little work as possible meaning that whenever I can benefit from using the values already stored I do so. The preactivation values for the next layer for these two instances is currently given by:

$$S^3 = \begin{bmatrix} -2 & 0 & -2 & 0 \\ -2 & 0 & -2 & 0 \end{bmatrix}$$

where each row corresponds to an instance and each column to a neuron at the next layer. The weight vector going into the next layer from the fourth neuron in the second layer is given by:

$$W = \begin{bmatrix} -1 & 1 & 1 & 1 \end{bmatrix}$$

Looking at the critical instances it is easy to recognize, that if they changed sign and thus activation, then the first would decrease from 1 to -1 and the second would increase from -1 to 1. Thus, we have:

$$\hat{S}^3 = S^3 + \begin{bmatrix} -2 \\ 2 \end{bmatrix} \circ \begin{bmatrix} -1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -2 & -4 & -2 \\ -4 & 2 & 0 & 2 \end{bmatrix}$$

Here $\circ$ is elementwise mulplication.

Until now it has been possible to calculate what is happening by looking at the preactivation values stored in memory and updating them. This is efficiently to do for the

neuron that is being evaluated and for the next layer, but afterwards it makes more sense to forget what is in memory and instead finish the forward propagation with the temporary matrices. To finish the forward propagation one need to apply the activation function, (3), to $\hat{S}^3$, yielding the result:

$$\hat{U}^3 = \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

The last step is to multiply with the last weight matrix to obtain the preactivation values for the last layer. Here I will only give the result:

$$\hat{S}^4 = \hat{U}^3 W^4 = \begin{bmatrix} 2 & 0 & 0 & -4 & 4 & 2 & -2 & 2 & 0 & 2 \\ -2 & 0 & 0 & 4 & -4 & -2 & 2 & -2 & 0 & -2 \end{bmatrix}$$

Compare this to the current preactivation values, which is determined after the random initialization:

$$S^4 = \begin{bmatrix} -4 & -2 & 2 & 2 & -2 & 0 & 0 & 0 & -2 & -4 \\ -4 & -2 & 2 & 2 & -2 & 0 & 0 & 0 & -2 & -4 \end{bmatrix}$$

Since the correct labels for these two instances are 4 and 9 respectively (using 0-indexing), the objective vector, using the integer objective function, is initially given by:

$$O = \begin{bmatrix} -2 - 2 \\ -2 - 4 \end{bmatrix} = \begin{bmatrix} -4 \\ -6 \end{bmatrix}$$

whereas using $\hat{S}^4$, the result is:

$$\hat{O} = \begin{bmatrix} 4 - 2 \\ -4 - 2 \end{bmatrix} = \begin{bmatrix} 2 \\ -6 \end{bmatrix}$$

This gives a vector of delta changes:

$$D = \hat{O} - O = \begin{bmatrix} 2 - (-4) \\ -6 - (-6) \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \end{bmatrix}$$

This means that we have found out that if the activations change for the two critical instances, then this has a positive effect for one of them and zero effect for the other. The last thing we need to do is to find out which, if any, of the four weights going

into the neuron can make the activations change. The current weights going into this neuron has the values:

$$W_4^2 = \begin{bmatrix} -1 & 1 & 1 & 1 \end{bmatrix}$$

and the output of the previous layer for the two critical instances are:

$$U^1 = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

What I then do is that I create four copies of the preactivation values for the critical instances, one for each weight going into the neuron. Afterwards I simulate what would happen if the weights are 'flipped', take the activations of these simulated preactivation values and compare them to the current activations. This gives me a matrix where each row represents an instance and the column represents a weight. The elements indicate whether the activation has changed or not, meaning that to get the effect of a weight we can columnwise take the inner product between the column and the delta changes vector, D, found earlier. I start by finding the simulated preactivation values:

$$\hat{S}_4^2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -2 & -2 & -2 & -2 \end{bmatrix} + \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \circ \begin{bmatrix} 2 & -2 & -2 & -2 \end{bmatrix} = \begin{bmatrix} 2 & 2 & -2 & -2 \\ 0 & 0 & 0 & -4 \end{bmatrix}$$

I then need to find out where there is a change in activation values compared to the current solution. Clearly, since the current preactivation value for the first instance is 0, this activation is currently 1, and for the second it is -1 as the preactivation value is -2. Thus, it gives the following 'changes' matrix, where 1 indicate that the activation has changed and 0 indicate that it has not.

$$\text{changes} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

As an example this shows that for the weight indexed by the third column, flipping that value from 1 to -1, will change the activation of both the critical instances. The last remaining thing to do is to, columnwise found the effect of flipping each weight. This gives the following delta values:

$$DW = \begin{bmatrix} 0 \cdot 0 + 1 \cdot 0 & 0 \cdot 0 + 1 \cdot 0 & 1 \cdot 6 + 1 \cdot 0 & 1 \cdot 6 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 6 & 6 \end{bmatrix}$$

Thus, an improvement of 6 in terms of the integer objective function can be found by flipping the third or fourth weight going into this neuron.

5.5 Solution Improvement

Recall, that the goal of the classification problem is to be able to classify test instances that are not seen during training. During training the model is trained on batches of training instances and an objective function is used to evaluate the quality of the model. I will test the effect of using a single batch compared to using several and a key ingrediense in most of my algorithms are the iterated improvement algorithm. In words, this algorithm takes a solution and search for improvements. I do this by iterating through the neurons in the network as described earlier. For each neuron, it takes the best move found and checks whether this is an improvement, and if it is, the move is commited and the current solution is updated. The algorithm only stops whenever the time limit has been reached or the solution has reached a local optima, which is when all the neurons have been checked without finding an improvement. In practice it is quite fast to end up in a local optima for a single batch The pseudocode for this algorithm can be seen in Algorithm 2.

Algorithm 2 Pseudocode for Iterated Improvement

```
1: Input:
2:     Initial solution currentSolution
3:     Time limit timeLimit
4: startTime ← current time
5: while current time − startTime < timeLimit do
6:     Shuffle the nodes of currentSolution
7:     improvementFlag ← False
8:     for each node in currentSolution do
9:         bestMove ← findBestMove(currentSolution, node)
10:        if delta value of bestMove > 0 then
11:            applyMove(currentSolution, bestMove)
12:            improvementFlag ← True
13:        end if
14:        if current time − startTime ≥ timeLimit then
15:            return currentSolution
16:        end if
17:    end for
18:    if improvementFlag = False then
19:        return currentSolution
20:    end if
21: end while
22: return currentSolution
```

Only using Algorithm 2 will typically not yield good results. While training a neural network many local optimas are usually visited and traditional neural network training has several techniques to escape from a local optima. I will use a simple optimization metaheuristic, iterated local search (ILS), to escape from the local optima. This metaheuristic works by taking the current solution, which is a locally optimal solution, and use it to get another solution which is no longer locally optimal. Afterwards, iterated improvement will be applied to the new solution until it arrives at a local optima and the process repeats itself until the solution has converged or the time limit has been

reached. The strategy used to get another solution from the current solution is called perturbing. In the context of training BNNs and TNNs it works by taking a solution and randomly change the values of a number of the weights. The important parameter here is how many weights to change values of. If too few weights are changed, there is a high chance that the solution falls back to the same local optima it came from. On the other hand, if too many weights are changed, it could be the same as using random restart, which could mean that the solution quality does not improve much. The pseudocode is given in Algorithm 3.

Another problematic aspect of this algorithm is that it is more difficult to determine when the model has converged. Instead of a convergence criterion I use a time limit, such that the runtime is controlled. A convergence criterion could be to set a maximum number of perturbations allowed without ending up in a local optima that is better than the best seen so far.

Algorithm 3 Pseudocode for Iterated Local Search

```
1: Input:
2:     Initial solution currentSolution
3:     Time limit timeLimit
4:     Perturbation size ps
5: startTime ← current time
6: bestSolution ← currentSolution
7: while current time − startTime < timeLimit do
8:     currentSolution ← IteratedImprovement(currentSolution,
           timeLimit - (current time - startTime))
9:     if currentSolution > bestSolution then
10:        bestSolution ← currentSolution
11:    else if currentSolution < bestSolution then
12:        currentSolution ← bestSolution
13:    end if
14:    currentSolution ← Perturb(currentSolution, ps)
15: end while
16: if currentSolution > bestSolution then
17:    return currentSolution
18: else
19:    return bestSolution
20: end if
```

5.6 Multiple Batch Training

One of the ways I try to make sure that the models trained generalize well is to make use of more data and train the network on several batches. It is the same solution that is being trained on all of the batches, in the sense that the solution after training one batch is the starting solution to the next batch. This gives rise to a couple of problems. The first one is, how do we know what the 'best' solution is? If the solution returned is the one after the last batch, then it might be that it is heavily influenced by the last batch and as such it would have been better to use one of the solutions earlier on.

Another consideration is how to make sure that the solution does not forget what it has learned from the previous batch. My attempt to solve these problems is to use an early stopping technique, where I after every k batches get the validation accuracy on the validation dataset and after the complete training process I return the solution with the best validation accuracy. Of course, this introduces another parameter, k. Ideally one could set $k = 1$ and get the validation accuracy for all the solutions, but this might be too costly as getting the validation accuracy involves evaluating the solution on a large dataset.

One of the concerns by training a model on several batches of data is that the model overfits on the batch it is currently trained on and forgets what it learned from the previous batch. For this reason, I investigate whether a sporadic local search approach helps the model to generalize better. This works by setting a parameter bp, which is a parameter in the interval $[0, 1]$, which is given to the Bernoulli distribution, which then returns a '1' with probability bp and a 0 otherwise. Before training on a batch, each weight samples a value from this distribution, and if the value is 1, then the weight is a part of the search for that batch and otherwise the value is kept fixed for that particular batch. For each new batch, new weights are selected to be part of the search. The hope is that this helps the model to avoid getting too focused on the current batch and reduce the overfitting gap.

The pseudocode for this algorithm is given in Algorithm 4. In line 15 I give an additional parameter compared to the iterated improvement pseudocode in Algorithm 2. In practice it works a bit different, but this is just to underline that not all weights are part of the search. Notice, that this could easily be adjusted to using Algorithm 3 instead in line 15. The only difference would be that the algorithm needs to take a perturbation size as input and that each ILS phase are given a fixed time limit instead as it would continue indefinetely otherwise.

Algorithm 4 Pseudocode for Multiple Batch Iterated Improvement

Input:
    Time limit *timeLimit*
    Set of batches *batches*
    Interval to early stopping k
    Number of epochs *epochs*
    Bernoulli parameter *bp*
$startTime \leftarrow$ current time
$currentSolution \leftarrow$ random solution
$bestSolution \leftarrow currentSolution$
$bestValidationAccuracy \leftarrow$ ValidationAccuracy($currentSolution$)
$counter \leftarrow 0$
while current time $- startTime < timeLimit$ **do**
    for *epoch* in range(*epochs*) **do**
        for *batch* in *batches* **do**
            $searchWeights \leftarrow$ SelectWeights(bp)
            $currentSolution \leftarrow$ IteratedImprovement($currentSolution$, $timeLimit$ - (current time - $startTime$), $searchWeights$)
            if $counter$ modulo $k = 0$ **then**
                $ValidationAccuracy \leftarrow$ ValidationAccuracy($currentSolution$)
                if $ValidationAccuracy > bestValidationAccuracy$ **then**
                    $bestSolution \leftarrow currentSolution$
                    $bestValidationAccuracy \leftarrow ValidationAccuracy$
                end if
            end if
            $counter \leftarrow counter + 1$
            if current time $- startTime \geq timeLimit$ **then**
                return *bestSolution*
            end if
        end for
        batches $\leftarrow$ ResampleBatches
    end for
end while
return *bestSolution*

So far I have looked at algorithms making moves based on one batch and tried to make sure that the model does not forget what it has learnt from other batches by only looking at a subset of the weights. An alternative method is to make less moves, but making

sure that the moves generalize better. One possibility for this is to sum up delta values for the moves across several batches and only after a certain number of batches, some moves are committed. This should make sure that the moves committed benefit not only a single batch of samples, but multiple batches. An important decision to make is when to make updates. The more batches seen before making updates, the more confident will the algorithm be that the moves generalize well, but it will also be slower. For this reason I add another aspect to the algorithm, such that it initially make updates based on very few batches and later on it makes updates on more batches. The parameters for this are given in line 5-7 in Algorithm 5, which shows the pseudocode for this algorithm.

A problematic aspect of this algorithm is that the delta values are calculated under the assumption that a single move is taken, but here, to speed up the process, I take many moves at once. However, to avoid the moves interfering too much with each other I only take one move per neuron. A different problem is again the convergence problem, where I, to overcome this problem, use a time limit. For this algorithm I do not use early stopping as the use of several batches to make updates should make sure that it generalizes better compared to the version introduced earlier.

Algorithm 5 Pseudocode for Multiple Batch Aggregation Algorithm

Input:
    Time limit *timeLimit*
    Set of batches *batches*
    Bernoulli parameter *bp*
    How many batches before making updates in the beginning *updateStart*
    The maximum number of batches before making updates *updateEnd*
    How often to increase the update interval *updateIncrease*
$startTime \leftarrow$ current time
$currentSolution \leftarrow$ random solution
$updateInterval \leftarrow updateStart$
$counter,\ updateCounter \leftarrow 0,\ 0$
$searchWeights \leftarrow$ SelectWeights(bp)
$deltaValues \leftarrow 0$
while current time $- startTime < timeLimit$ **do**
    for *batch* in *batches* **do**
        $updateCounter \leftarrow updateCounter + 1$
        $deltaValues \leftarrow deltaValues +$
          CalculateDeltaValues($currentSolution$, *batch*, $searchWeights$)
        if $updateCounter = updateInterval$ **then**
            for each *node* in *currentSolution* **do**
                $bestMove \leftarrow$ findBestMove($deltaValues$, *node*)
                if delta value of $bestMove > 0$ **then**
                    applyMove($currentSolution$, $bestMove$)
                end if
            end for
            $updateCounter \leftarrow 0$
            $deltaValues \leftarrow 0$
            if *counter* modulo $updateIncrease = 0$ **then**
                $updateInterval \leftarrow updateInterval + 1$
            end if
        end if
        $counter \leftarrow counter + 1$
    end for
    batches $\leftarrow$ ResampleBatches
end while
return *bestSolution*

5.7 Code Organization

One of the major challenges of this thesis has been to develop the framework for training BNNs and TNNs. The framework needed to be quite flexible so that it allowed both BNNs and TNNs but also such that all of the algorithms and experiments could be tested within the same framework. All of the source code for this thesis can be found at: (GitHub link). The implementation is done in Python. To train a BNN or TNN, 'main.py' needs to be called with the right parameters. From this file, everything else runs automatically. As a starting point, it initializes the 'Reader' class and loads the training, validation and testing set by calling the load data function. The current framework support loading the MNIST, Fashion-MNIST (FMNIST) and the Adult dataset. To load from other datasets, the necessary function need to be implemented in the Reader class. Next, the 'Instance' class is called, which takes all the settings of the experiment as input as well as the three datasets. The Instance class has the very important 'loader' function, needed to iterate through batches from one of the datasets.

Having this, one of the algorithms presented is called. When an algorithm is called, it initially starts by loading a batch and create a 'Solution' object. This object is the core of the implementation. The most important attributes of this object is the W, S and U matrices introduced earlier as well as a vector, O denoting the contribution of each sample to the objective function. It also has an attribute of the nodes in the network, that can be iterated through. The S, U and O attributes are dependent on the current batch and as a result the Solution object has a function to change the batch, which includes re-evaluating these attributes. The object also has functions to initialize a random solution and copy the weights. More importantly, it is also here the function to commit a move is placed, where the necessary updates are done. The remaining functions are functions to perturb the solution, used in ILS, and a function to select search weights, used in sporadic local search.

The algorithm then moves on and calls one the 'solvers' implemented. There are two 'solvers' implemented, iterated improvement which follows Algorithm 2 or iterated local search, described in Algorithm 3, which uses Algorithm 2. In the iterated improvement

algorithm, the 'Delta Manager' object is called. This object has a single function, delta calculation, which either use a delta function for BNN or TNN. This function takes the current solution and a node as input. For this node, all the weights going into this node, which are part of the search, is tested to see what the delta value is if the weight changed value. This follows the procedure described in section 5.4. The delta calculation function returns a sequence of moves back to the iterated improvement algorithm, which takes the best of the moves and decide whether to commit the move or not. If the algorithm uses multiple batch training, it loads a new batch, change the batch on the solution and the process repeats itself until the time limit has been reached.

6 Experimental Analysis

I have implemented several algorithms and each algorithm are dependent on different parameters, which makes it difficult to make a one-factor-at-a-time analysis. Instead I try to be as structured as possible to test what algorithm, objective function, network structure and network type (BNN or TNN), that works best. I divide the experimental analysis into several sections. At first I only test on the well-known MNIST dataset, which is a large database of handwritten digits. The dataset contains 60,000 training images and 10,000 testing images. Each image is a 28×28 grayscale image of a handwritten digit. At the end of the experimental analysis I take the knowledge gained about what parameters work best for MNIST and try to see how it works on different datasets. The first section will be concentrated about single batch training, which could be relevant in few-shot learning. Here the goal is to test what objective function works best, how the amount of data influences the accuracy and what the effect of the network size is. This section will only use the ILS algorithm outlined in Algorithm 3. Some of the network architectures tested will be identical to those of Toro Icarte et al. (2019) and Thorbjarnarson and Yorke-Smith (2023), such that a comparison can be made. At the end a small hyper-parameter study will be conducted to see how sensitive the algorithm is to the choice of ps.

The second section is about multiple batch training. Again the goal is to investigate what objective functions work best, but the goal is also to test which of the two algorithms presented for multiple batch training works best. A focus point will also be how the algorithms scale with increasing network size. The first two sections will only deal with BNNs. In the third section I will test the TNN, where I speficially aim to investigate whether a regularization parameter help. Another focus point will be how it compare against a BNN, especially if it requires more training time to obtain similar accuracies or not. In the fourth section I present the results on the BeMi ensemble introduced by Maria Bernardelli et al. (2023). Here the focus will be how the ensemble works with increasing number of training data, as this is a limitation for their MIP implementation. Lastly, in the fifth section I intend to use the knowledge obtained from the previous four sections to test how the best found algorithms and parameters

generalize to other datasets.

The source code for the implementation in Python and experiment scripts can be found at: (GitHub link). For all the experiments, the results reported are an average of 5 runs for each configuration tested. The experiment is run on a Windows 10 computer with a 64-bit operating system with 8 GB RAM. The processor is a Intel(R) Core(TM) i5-6400 CPU with 2.70GHz.

6.1 Single Batch Training

This section will only use the ILS procedure given in Algorithm 3. In each perturbation I initially change the value of 25 randomly chosen weights. The first goal is to explore what objective function works best under what circumstances. The first experiment will be to test the influence of more training data, but with a short time limit of only 60 seconds. The second experiment tests how the results develop if the time limit increases. In experiment 3, I test different network architectures and finally in experiment 4, I test how sensitive the results are to the choice of ps. For all the experiments in this section, to compare against existing literature, I use a balanced training and test set meaning that the number of instances are equal for each class. The test accuracy is reported on a set of 8,000 instances from the test set, 800 instances for each digit.

6.1.1 Comparing Objective Functions

As a starting point I start by testing how the different objective functions compare against each other. The network has a single hidden layer, such that the network structure is $[784, 16, 10]$. I use a BNN with a time limit of 60 seconds. Besides from comparing the objective functions against each other, I also want to see how the results develop as the amount of training data increase and to see this I let the total number of training samples vary from 100 to 2000, or from 10 to 200 examples from each class. In Figure 2 I plot the mean of the test accuracy, with the standard deviation as a shaded area around it. Initially as more training samples are added, the accuracy increases a lot, but later on the effect takes of and the accuracy actually decreases at the end. It is possible that the results could be even better with more time, which will be ex-

plored in the next experiment. At first the cross entropy objective function works best, but eventually the integer objective function takes over. The objective function based on the Brier score does not seem to be able to compete against the other objective functions. The maximum mean accuracy achieved for this result is with the integer objective function and a batch size of 1800, which gives a mean accuracy of 71.73 %.

This network architecture is also tested by Thorbjarnarson and Yorke-Smith (2023), who only uses a training set of 100 samples, 10 from each digit. Their best performing model gets an average testing accuracy of 51.1 % with an average runtime of 1852 seconds. With the cross entropy objective function I get a mean testing accuracy of 55.41 % for the same amount of data, but with a time limit of only 60 seconds. For the integer objective function, the result is a testing accuracy of 49.66 %. Both of my results may very well be subject to improvement, if the time limit increases. Also, my results are on a pure BNN, while their results are based on a TNN. My results also show that the network architecture, even though it is quite small, can find a model that generalizes better, if the amount of training data increase.

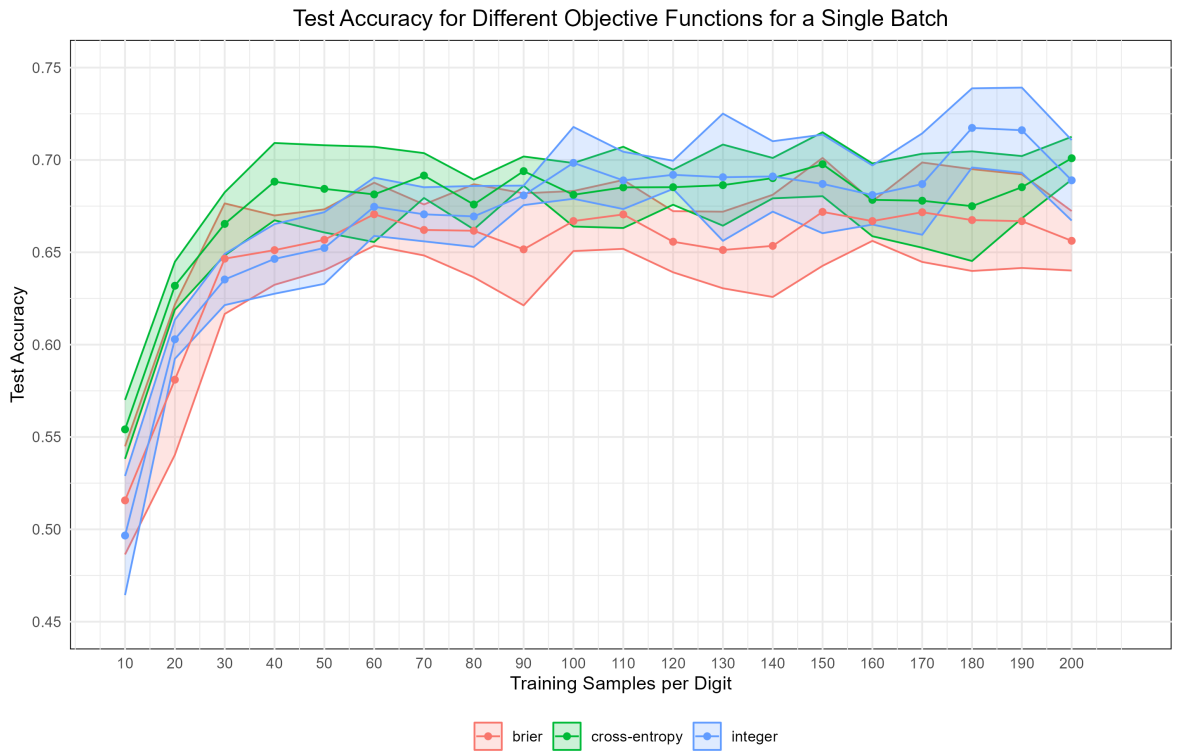

Figure 2: Testing accuracies for training a single batch on an increasing number of training samples. The x-axis shows the number of samples from each class in the batch and the y-axis shows the test accuracy. The results are for a BNN with a single hidden layer with 16 neurons using ILS with a time limit of 60 seconds. The batches are balanced, meaning that there is the same number of samples present from all classes. The testing accuracy is reported for 8000 samples, 800 from each class. The figure shows the mean accuracy of 5 runs as a line and the standard deviation of the accuracy as a shaded area around it.

6.1.2 Testing the Effect of the Time Limit

As a next step I investigate what the effect of the time limit is. Since the previous experiment showed that more training data give better results, there were signs that the effect took of as the amount of training data increased. The goal of this experiment is to investigate whether the results can get even better if the training is given more time. I test with a batch size of 2000 but now with different time limits ranging from 1 minute to 5 minutes. The mean accuracies can be seen in Table 1, where it is evident that a time limit of 60 seconds was not enough. For all the objective functions, the

accuracy increases as the time limit increases. The Brier objective function still cannot compete against the two other objective functions, which get very similar results. The maximum mean accuracy for this experiment is now 74.23 %, an significant improven to the 71.73% obtained before.

ObjectiveFunction	TL: 60	TL: 120	TL: 180	TL: 240	TL: 300
brier	0.6562	0.6896	0.7063	0.7113	0.7149
cross-entropy	0.7009	0.7202	0.7298	0.7360	0.7423
integer	0.6890	0.7150	0.7340	0.7407	0.7413

Table 1: The mean test accuracies on MNIST for different time limits. The results are obtained by training a BNN on a single batch with 200 samples for each digit. The algorithm used is the ILS algorithm with perturbation size set to 25. The time limit is in seconds.

6.1.3 Comparing Different Network Structures

I also want to test what effect the network architecture has. From the previous experiments, I found that increasing time limits and increasing amounts of data give better results. For this reason I use a time limit of 300 seconds and again a total batchsize of 2000 balanced sampled samples. Apart from the already tested network architecture with one layer with 16 neurons, I also try one with two layers, each with 16 neurons. I also try two larger networks with 128 neurons in each layer, where I test with one and two layers again.

From Table 2, it can be seen that the general pattern is that more neurons in the layers boost the accuracy significantly. For some reason, this does not seem to apply to the Brier objective function, where the accuracy decreases significantly. In deep learning, it often helps adding more layers, which is not the case here. The most likely reason for this is the fact that the amount of training data does not justify the second layer. This will be tested further in the next section. The cross-entropy objective function

performs slightly better that the integer objective function in networks with only one hidden layer, but for networks with two hidden layers, the integer objective function outperforms the others.

Objective function	Hidden layers	16	128
brier	One	0.7149	0.5096
cross-entropy	One	0.7423	0.8185
integer	One	0.7413	0.8003
brier	Two	0.5125	0.3795
cross-entropy	Two	0.4844	0.6614
integer	Two	0.5340	0.7284

Table 2: The mean test accuracies on MNIST for different network structures. The results are obtained by training a BNN on a single batch with 2000 samples using the ILS algorithm with a time limit of 300 seconds. The columns indicates the width of the hidden layers

6.1.4 Fine Tuning the Perturbation Size

So far I have used a fixed parameter for the number of weights to change values for in the perturbation phase. The goal of this experiment is to see if there are room for improvement, i.e. if changing the perturbation size changes the results to the better. Ideally, all the results should be re-run with a different perturbation size to see if any of the conclusions drawn so far changes, but since each experiment takes a long time to run, this is not computationally feasible. Instead I take the configurations from the best results obtained so far, and use them to test if changing the perturbation size improves the results further. The best result obtained so far was with the cross entropy objective function and with a single hidden layer with 128 neurons. Since this is a relatively big network compared to some of the others, I also test changing the perturbation size for the network with 16 neurons in the hidden layer. The time limit is again 300 seconds. Besides from 25, I now try with perturbation sizes of 5, 10, 15, 20, 30, 35 and 40 as well. The results are given in Table 3. For the small network architecture, the highest

mean accuracy is achieved with a perturbation size of 40, whereas for the large network architecture it is with a perturbation size of 30. In general the results show that the perturbation size should not be too small. This is likely due to the reason that small perturbation sizes might mean that the solution return to the same local optima. This hyoerparameter experiment also shows that for the small network architecture tested in the first experiments, it is possible to achieve a even better accuracy of 75.95%, again with only 2000 data samples. Further, with a larger network it is possible to achieve as high accuracy as 82.20% using only this limited amount of data.

NeuronsHiddenLayer	PerturbationSize	Mean	SD
16	5	0.7050	0.0085
16	10	0.7127	0.0153
16	15	0.7272	0.0240
16	20	0.7384	0.0207
16	25	0.7423	0.0074
16	30	0.7575	0.0129
16	35	0.7504	0.0149
16	40	0.7595	0.0153
128	5	0.8135	0.0041
128	10	0.8159	0.0018
128	15	0.8179	0.0042
128	20	0.8154	0.0037
128	25	0.8185	0.0058
128	30	0.8220	0.0054
128	35	0.8204	0.0040
128	40	0.8209	0.0076

Table 3: The mean test accuracies on MNIST for two different networks with a single hidden layer. The results are obtained by training a BNN on a single batch with 2000 samples using the ILS algorithm with a time limit of 300 seconds.

6.2 Multiple Batch Training

The goal of this section is to go a step further and train on more data using multiple batch training. The main motivation is that seeing more training data should help the model generalize better, but the important question is how to make use of more training data. From the first section it was already clear that using more data in a single batch setting requires more time. Instead of trying to continue to increase the batch size and train on a single batch, I instead want to train a model that sees many batches throughout the training. In this section I work with a overall time limit of 600 seconds. There are three algorithms that I test and compare against each other. I test two versions of Algorithm 4, the first version is as the algorithm is presented with iterated improvement for each batch. The other version is with iterated local search for each batch. The goal of this is to investigate whether it helps seeing more data, but for less time, which is the case for the iterated improvement version, or if it is better to see less data, but find a better solution each time. These algorithms use early stopping to make sure that the solution returned is not too dependent on the last batch seen.

The third algorithm is Algorithm 5, where less moves are committed, but every move should generalize better, as a move is only committed if it improves the solution across several batches. The main question is, whether this algorithm can compete against the others and whether it is too slow compared to the others. In this section I do not use balanced training and test sets anymore. MNIST is not uniformly distributed and the main reason for using balanced training and test set was to make fair comparisons against existing literature. I still train on a BNN and I use a batch size of 1,000. Whenever I use early stopping I take 20 % of the training set, 12,000 samples, and use as validation set. For iterated improvement I calculate the validation accuracy after every fourth batch, where for ILS I do it after every single batch. I test the same four network architectures as earlier. The accuracies are calculated on the entire test set of 10,000 samples.

The first experiment in this section is again to test objective functions against each other and to see if they perform differently under different circumstances. For this I

test the three different objective functions on four different network structures and with all three algorithms. Afterwards I investigate whether the sporadic local search has any positive impact by testing the *bp* parameter.

6.2.1 Comparing Objective Functions

The goal of the first experiment is to see what objective function and algorithm work best. I use the settings specified above. For the ILS version, I let each ILS phase take 5 seconds and set $ps = 25$. For all the algorithms I set $bp = 0.2$. I set $updateStart = 1$, $updateEnd = 15$ and $updateIncrease = 10$. Table 4 presents the mean accuracies. For the networks with 16 neurons in each hidden layer, it is the case for all configurations, that the single layer version performs best. Further, for the network with 16 neurons and a single hidden layer, the Brier objective function now perform almost as good as the cross entropy in two of the algorithms and for the aggregation algorithm even sligtly better. With two hidden layers, each with 16 neurons, the Brier score objective function clearly outperforms the other objective functions.
When the network size increases, the results are different. For both networks with a single hidden layer and two hidden layers each with 128 neurons, the aggregation algorithm is the best. What is more surprising is that the integer objective function now is best in all cases except for the one layer case with the aggregation algorithm. With two hidden layers, the integer objective function clearly outperforms the other objective functions. One of the reasons why this objective function is better might be due to the fact that it is much faster. To see this, look at Table 5, where I take a deeper look into the iterated improvement version of Algorithm 4. Specifically, I take a look at how many batches the different objective functions iterate through within the time limit of 600 seconds. Here it is clear, that for all the configurations, the integer objective function iterates through more batches and as a results see more data.

I also take a look at the number of moves made per batch. As this number is expected to decrease as the number of batches increase, I also present the number for the first 30 batches. In both cases, the pattern is the same. For the integer objective function, less moves are made. This suggests that the number of batches seen is not higher only because the integer objective function is faster in computation time, but also because

less moves are made in each iteration. This is not necessarily a bad thing. The more moves that are made at each iteration, the more the current solution depends on the current batch. As a result, the gap between training accuracy and validation or test accuracy might be larger for the objective functions that make more moves based on the current batch. This is especially evident from Figure 3, 4 and 5 that plots the training accuracies and the validation accuracies for each of the three objective functions. The plots are based on iterated improvement algorithm and the network architecture with two hidden layers with 128 neurons each. The gap is quite small for the integer objective function compared to the Brier and cross entropy objective functions.

These are possible reasons why the integer objective function is better, but they do not necessarily explain why it is only better for larger networks. However, this is probably due to the definition of the objective function, that does not work with probability or probability distributions, but with maximizing the margins. For small networks it seem that this is not possible in the same way as with larger networks, where the integer objective function is better than the other objective functions in most cases.

Objective function	Hidden layers	Algorithm	16	128
brier	One	Iterated Improvement	0.8205	0.8479
brier	One	Aggregation Algorithm	0.8491	0.8858
brier	One	Iterated Local Search	0.7938	0.8499
cross-entropy	One	Iterated Improvement	0.8223	0.8432
cross-entropy	One	Aggregation Algorithm	0.8478	0.8655
cross-entropy	One	Iterated Local Search	0.7992	0.8416
integer	One	Iterated Improvement	0.8038	0.8668
integer	One	Aggregation Algorithm	0.7919	0.8781
integer	One	Iterated Local Search	0.7887	0.8632
brier	Two	Iterated Improvement	0.7852	0.7005
brier	Two	Aggregation Algorithm	0.8265	0.8454
brier	Two	Iterated Local Search	0.7458	0.7191
cross-entropy	Two	Iterated Improvement	0.7390	0.6736
cross-entropy	Two	Aggregation Algorithm	0.6936	0.7997
cross-entropy	Two	Iterated Local Search	0.7048	0.6986
integer	Two	Iterated Improvement	0.5677	0.8878
integer	Two	Aggregation Algorithm	0.4416	0.9153
integer	Two	Iterated Local Search	0.7095	0.8766

Table 4: The mean test accuracies on MNIST for different network structures and algorithms. The network is a BNN trained with a time limit of 600 seconds. Iterated Improvement and Iterated Local Search refers to Algorithm 4, and here early stopping is used. The validation dataset is 12,000 samples and for II, the validation accuracy is calculated every fourth batch, whereas for ILS, it is after every batch. The solution with the highest validation accuracy is returned. I set bp equal to 0.2 and ps to 25. Each ILS phase is allowed to run for 5 seconds. For the Aggregation Algorithm, I do not use early stopping, but return the solution at the end. The parameters updateStart, updateEnd and updateIncrease are set to 1, 15 and 10 respectively. For all the algorithms a batch size of 1000 is used.

Objective function	Hidden layers	Mean	MeanBatches	MeanMoves	MeanMoves30
brier	16	0.8205	427.20	213.0588	236.5333
cross-entropy	16	0.8223	475.20	174.5439	217.9800
integer	16	0.8038	594.20	107.2392	115.9400
brier	16-16	0.7852	392.80	179.3991	181.5333
cross-entropy	16-16	0.7390	394.60	183.8709	178.2067
integer	16-16	0.5677	591.60	66.0823	84.8200
brier	128	0.8479	45.80	4205.4210	4191.3800
cross-entropy	128	0.8432	62.60	2854.2981	2988.1267
integer	128	0.8668	147.80	993.4502	1158.5133
brier	128-128	0.7005	48.40	2542.8706	2049.0067
cross-entropy	128-128	0.6736	47.00	2579.0009	2537.3467
integer	128-128	0.8878	110.80	887.7500	935.8800

Table 5: Summary statistics for the Iterated Improvement version of Algorithm 4. 'MeanMoves is the average number of moves made per batch and MeanMoves30 is the average number of moves made per batch in the first 30 batches.

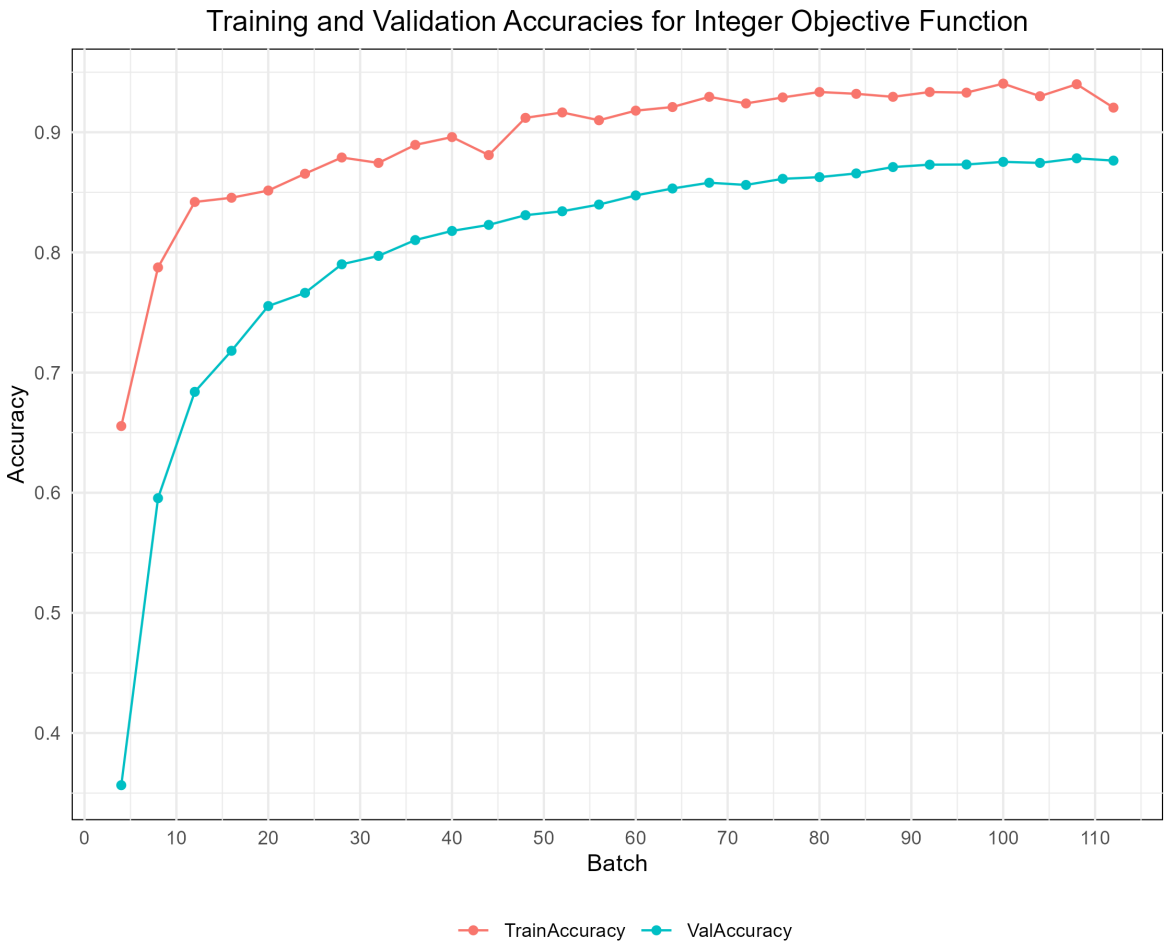

Figure 3: Training and validation accuracies for the iterated improvement algorithm with the integer objective function. The network structure is with two hidden layers, each with 128 neurons in each. As such, the plot is for the bottom row of Table 5.

Figure 4: Training and validation accuracies for the iterated improvement algorithm with the Brier objective function. The network structure is with two hidden layers, each with 128 neurons in each. As such, the plot is for the third last row of Table 5.

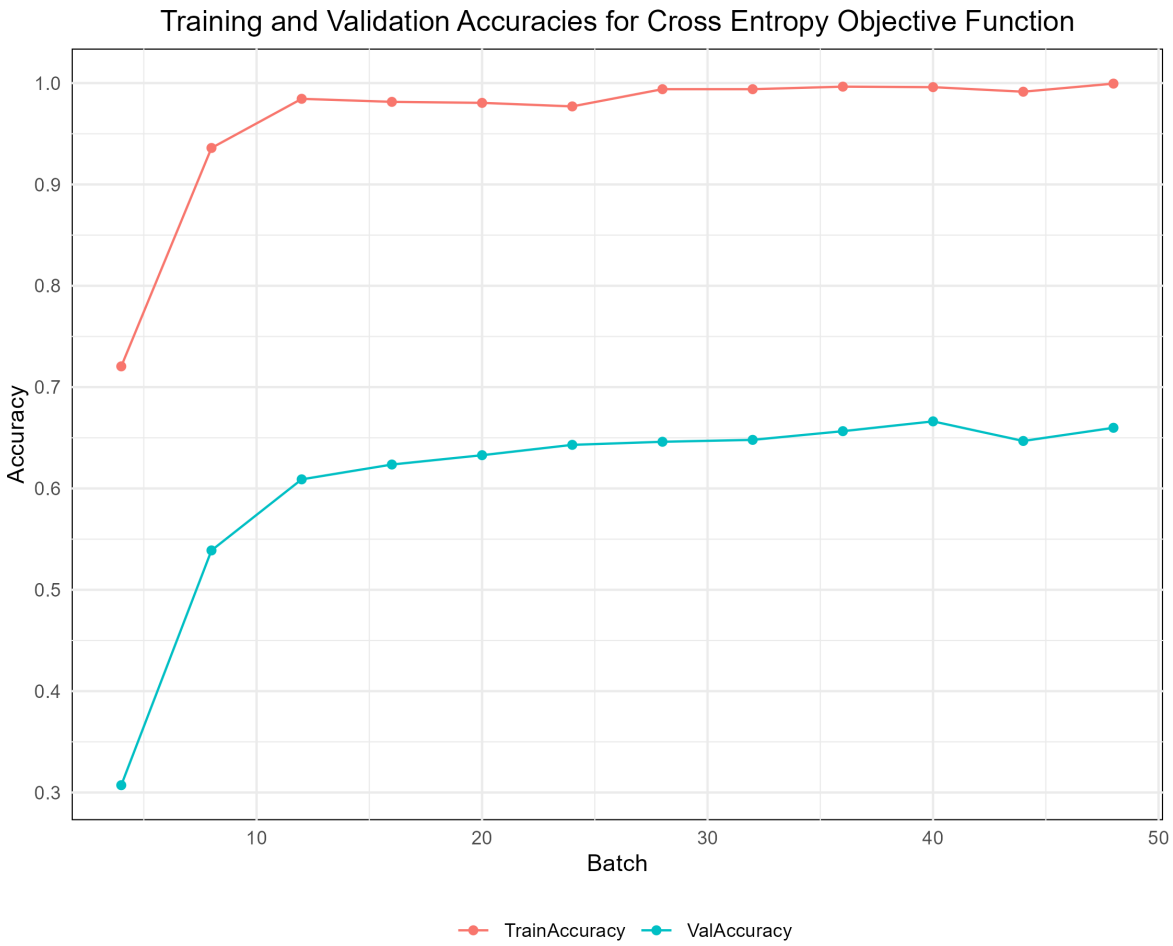

Figure 5: Training and validation accuracies for the iterated improvement algorithm with the cross entropy objective function. The network structure is with two hidden layers, each with 128 neurons in each. As such, the plot is for the second last row of Table 5.

6.2.2 Testing the Effect of Sporadic Local Search

For all the three algorithms I used sporadic local search, where the parameter bp determines the probability that a weight is part of the search in that iteration. Until now I used $bp = 0.2$, but now I change the value of this to see how important it is and if it has any affect at all. Notice that $bp = 1$ corresponds to not using sporadic local search, as here all weights are subject to change in each iteration. The hope is that using sporadic local search would improve the ability of the model to generalize by being less dependent on the current batch. For the aggregation algorithm, this is not necessarily true, as the algorithm here do not make moves based on a single batch. Instead of $bp = 0.2$ I

now try with 0.1, 0.3, 0.4, 0.5 and 1 as values. I choose the best configuration from the previous experiment, so I use a network architecture with 2 hidden layers, each with 128 neurons and I use the integer objective function. The mean test accuracies can be seen in Table 6, where the results are somewhat disappointing. For ILS and iterated improvement, the test accuracies are better with no use of sporadic local search and for the aggregation algorithm the results are very similar across different values of bp, so there seems to be no positive effect of using sporadic local search.

Algorithm	BP	Mean	SD
Iterated Improvement	0.1	0.8825	0.0030
Iterated Improvement	0.2	0.8878	0.0033
Iterated Improvement	0.3	0.8917	0.0039
Iterated Improvement	0.4	0.8896	0.0051
Iterated Improvement	0.5	0.8917	0.0033
Iterated Improvement	1.0	0.8976	0.0030
Aggregation Algorithm	0.1	0.9130	0.0030
Aggregation Algorithm	0.2	0.9153	0.0025
Aggregation Algorithm	0.3	0.9125	0.0009
Aggregation Algorithm	0.4	0.9132	0.0013
Aggregation Algorithm	0.5	0.9131	0.0032
Aggregation Algorithm	1.0	0.9147	0.0032
Iterated Local Search	0.1	0.8564	0.0054
Iterated Local Search	0.2	0.8766	0.0050
Iterated Local Search	0.3	0.8806	0.0039
Iterated Local Search	0.4	0.8826	0.0031
Iterated Local Search	0.5	0.8844	0.0038
Iterated Local Search	1.0	0.8939	0.0033

Table 6: The mean test accuracies on MNIST for three different algorithms with different values of bp. The results are obtained by training a BNN for a time limit of 600 seconds. For II k=4, for ILS k=1 and each ILS phase runs 5 seconds and perturbation size is 25. For AA updateStart, updateEnd and updateIncrease are 1, 15 and 10 respectively.

6.3 Ternary Neural Networks

In a TNN, the weights can also be zero, which increases the capacity of the model. As a result the solution space gets bigger and the delta evaluation takes more time. When considering a move in a BNN, there is only one other possible value. For a TNN, two

other values need to be considered. For this reason, I expect that training a TNN takes longer time, but the hope is that, as the model has a larger capacity, it will be able to give higher accuracies as well. The goal of this section is to explore TNNs. At first I will test the effect of adding a regularization parameter, as introduced in section 5.3.5. Secondly, I will investigate if it takes longer to train a TNN compared to a BNN. The results in these sections are comparable to existing literature on training NNs using MIP, so here I again use balanced training and test sets and train on a single batch.

6.3.1 Testing the Effect of a Regularization Parameter

I start by testing the effect of the regularization parameter by testing different values. I train on a single batch, with 2000 samples and the network has a single hidden layer with 16 neurons, so the most comparable results obtained so far are those from the Single Batch Training section. I try with all 3 objective functions and different values of the regularization parameter. Whenever this parameter is equal to zero it corresponds to training a TNN without any additional regularization term. From Table 1, the results for training a BNN with the same settings are given. Here, for a time limit of 300 seconds, the accuracies were 71.49 %, 74.23 % and 74.13 % for the Brier, cross entropy and integer objective function respectively. From Table 7, it can be seen that whenever no regularization parameter is added, the mean accuracies already increase for the Brier and cross entropy objective functions, where for the integer the accuracy decrease compared to BNN.

In Table 7, the 'Connections' column show the average number of active connections, i.e. the weights whose value are not 0. The same value of the regularization parameter does not seem to have the same effect across objective functions. It seems that the integer objective function needs higher values of the regularization parameter compared to the other objective functions. This is related to the definition of the objective function and in particular the range of the objective function value. For the Brier objective function, there seems to be a positive effect of adding a regularization parameter of around 1.0, so I run another experiment to finetune this further. From Table 8, it seems to be the case that a value between 1.5 and 2.5 gives the highest accuracies with a maximum mean accuracy of 76.22 % for a value of 2. For this value, the average

number of active connections is 572. The total number of connections in the network is $784 \cdot 16 + 16 \cdot 10 = 12,704$, so it is less than 5% of the connections that are actually active.

For the cross entropy function, at first sight looking at Table 7, there is no value of the regularization parameter that gives higher accuracy, but a value of 1.0 comes close, so again I try another experiment to finetune further. Table 9 shows indeed that it was possible to find values of the regularization parameter, that gave higher accuracies. The maximum mean accuracy here os 76.65 %, but this time with a value of 4.0 for the regularization parameter. It is interesting that the average number of active connections in this case is 574, which is very close to the same number for the Brier objective function. For the integer objective function, there does not seem to be a positive effect, neither in Table 7 or 10, where I tried with more values.

It should be mentioned, that this finetuning was with respect to the specific settings used here. By the way the objective function terms are defined it is highly likely that changing either the number of connections in the network, i.e. the network architecture or the number of samples in the batch, then a new finetuning is possible. Thus, it is not expected that these values of the regularization parameter generalize well to other settings, but nevertheless the results show that for the Brier and cross entropy objective functions, there is something to be gained when regularizing the network. For the integer objective function, regularization did not seem to have any positive influence.

Reg	ObjectiveFunc	Mean	SD	Connections	LocalOptimas
0.00000	brier	0.7264	0.0209	8813	98
0.00001	brier	0.7057	0.0124	5053	59
0.00010	brier	0.7100	0.0095	4928	60
0.00100	brier	0.7143	0.0137	4785	68
0.01000	brier	0.7025	0.0047	4447	82
0.10000	brier	0.7088	0.0177	3776	106
1.00000	brier	0.7492	0.0089	1348	116
10.00000	brier	0.6811	0.0202	393	62
0.00000	cross-entropy	0.7471	0.0158	8774	117
0.00001	cross-entropy	0.7270	0.0076	5011	69
0.00010	cross-entropy	0.7283	0.0175	4984	69
0.00100	cross-entropy	0.7251	0.0095	4923	74
0.01000	cross-entropy	0.7242	0.0093	4743	78
0.10000	cross-entropy	0.7346	0.0128	4093	105
1.00000	cross-entropy	0.7446	0.0112	2349	130
10.00000	cross-entropy	0.7359	0.0123	228	65
0.00000	integer	0.7315	0.0174	8690	235
0.00001	integer	0.7120	0.0143	4462	130
0.00010	integer	0.7134	0.0139	4461	131
0.00100	integer	0.7121	0.0142	4462	130
0.01000	integer	0.7127	0.0139	4462	130
0.10000	integer	0.7140	0.0138	4462	131
1.00000	integer	0.7214	0.0153	4166	127
10.00000	integer	0.7257	0.0196	1212	142

Table 7: Summary statistics for single batch training of a TNN with 2000 samples. The network trained has a single hidden layer with 16 neurons and is trained for 300 seconds.

Reg	ObjectiveFunc	Mean	SD	Connections	LocalOptimas
0.00	brier	0.7264	0.0209	8813	98
0.50	brier	0.7308	0.0099	2349	123
0.75	brier	0.7372	0.0116	1829	121
1.00	brier	0.7492	0.0089	1348	116
1.25	brier	0.7471	0.0036	1141	105
1.50	brier	0.7555	0.0078	844	95
1.75	brier	0.7545	0.0118	727	85
2.00	brier	0.7622	0.0136	572	85
2.25	brier	0.7511	0.0124	510	82
2.50	brier	0.7569	0.0098	441	79
2.75	brier	0.7467	0.0171	385	65
3.00	brier	0.7419	0.0111	365	62
3.25	brier	0.7435	0.0126	308	57
3.50	brier	0.7574	0.0087	265	57
3.75	brier	0.7489	0.0199	222	63
4.00	brier	0.7514	0.0121	221	61

Table 8: Summary statistics for single batch training of a TNN with 2000 samples. The network trained has a single hidden layer with 16 neurons and is trained for 300 seconds.

Reg	ObjectiveFunc	Mean	SD	Connections	LocalOptimas
0.0	cross-entropy	0.7471	0.0158	8774	117
0.5	cross-entropy	0.7334	0.0098	3069	131
1.0	cross-entropy	0.7446	0.0112	2349	130
1.5	cross-entropy	0.7474	0.0170	1705	137
2.0	cross-entropy	0.7609	0.0114	1320	126
2.5	cross-entropy	0.7603	0.0098	1065	116
3.0	cross-entropy	0.7529	0.0101	894	111
3.5	cross-entropy	0.7578	0.0114	708	101
4.0	cross-entropy	0.7665	0.0136	574	94
4.5	cross-entropy	0.7543	0.0052	511	92
5.0	cross-entropy	0.7529	0.0109	439	87

Table 9: Summary statistics for single batch training of a TNN with 2000 samples. The network trained has a single hidden layer with 16 neurons and is trained for 300 seconds.

Reg	ObjectiveFunc	Mean	SD	Connections	LocalOptimas
2.5	integer	0.7149	0.0246	3173	148
5.0	integer	0.7307	0.0084	2278	140
7.5	integer	0.7298	0.0118	1766	136
10.0	integer	0.7258	0.0193	1212	140
12.5	integer	0.7298	0.0117	964	117
15.0	integer	0.7183	0.0208	779	105
17.5	integer	0.7083	0.0183	686	107
20.0	integer	0.7055	0.0196	552	95
22.5	integer	0.6976	0.0280	508	93
25.0	integer	0.6863	0.0222	454	87
27.5	integer	0.6719	0.0207	392	84
30.0	integer	0.6502	0.0102	331	81

Table 10: Summary statistics for single batch training of a TNN with 2000 samples. The network trained has a single hidden layer with 16 neurons and is trained for 300 seconds.

6.3.2 Comparing Binary Versus Ternary Neural Networks

In this section I investigate whether training a TNN takes longer time compared to training a BNN. In the previous section, I found that training a TNN using the cross entropy and Brier objective functions could boost the accuracy, especially with the correct regularization parameter. Here, I use these two objective functions and for each objective function I train a BNN, a TNN with no regularization parameter and a TNN with the best regularization parameter from Table 8 and 9. I let the time limit vary from 30 to 300 seconds to be able to take a look at the effect of the time limit. Figure 6, shows the three configurations for the Brier objective function. Although, the TNN is not as fast to train as a BNN, the standard TNN with no regularization is better than the BNN, even for short time limits. The TNN with regularization parameter is initially worse, but after 90 seconds it is better than the other objective functions and

it remains above the other objective functions for all the other time limits.

For the cross entropy objective function Figure 7, the different configurations are closer to each other and for most of the time, the BNN is better than the TNN without regularization. The TNN with regularization is better after 90 seconds and remains better for all time limits higher than this.

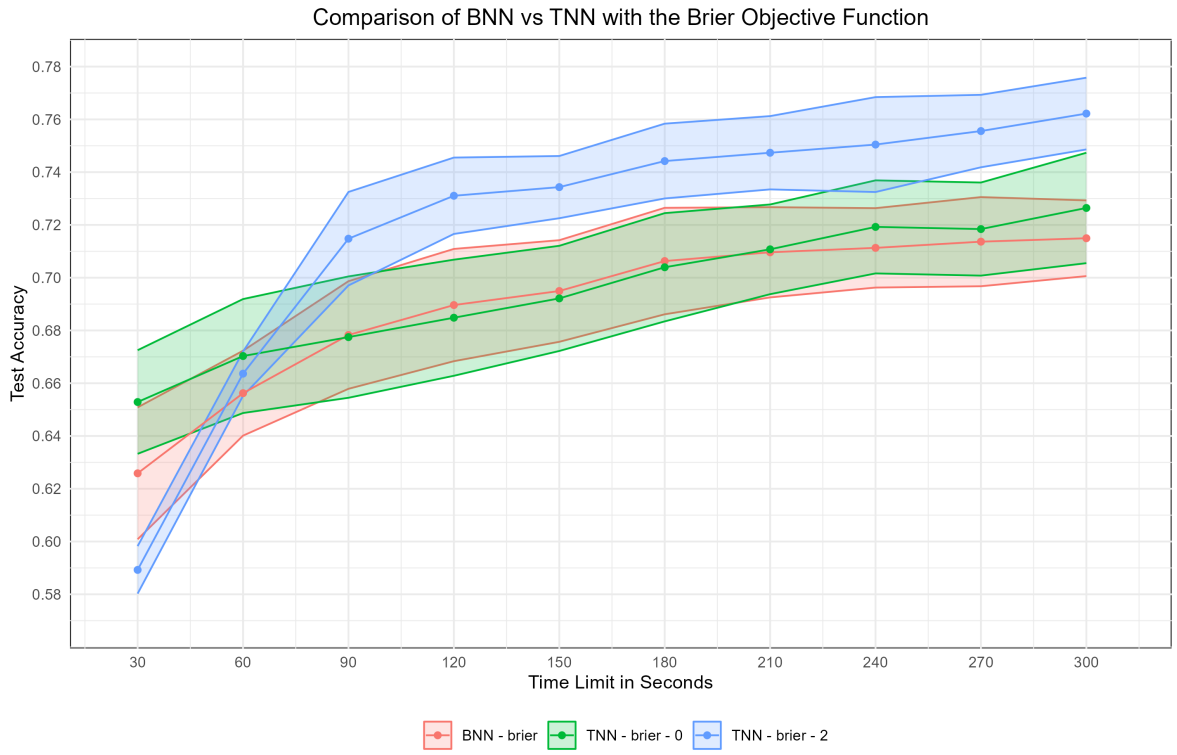

Figure 6: Test accuracies for the MNIST dataset. The networks are trained on a single batch with 2000 samples, 200 for each digit. The results are for a neural network with a single hidden layer with 16 neurons. The labels indicate what type of network is trained and what the regularization parameter is. The figure shows the mean accuracy of 5 runs as a line and the standard deviation as a shaded area around it.

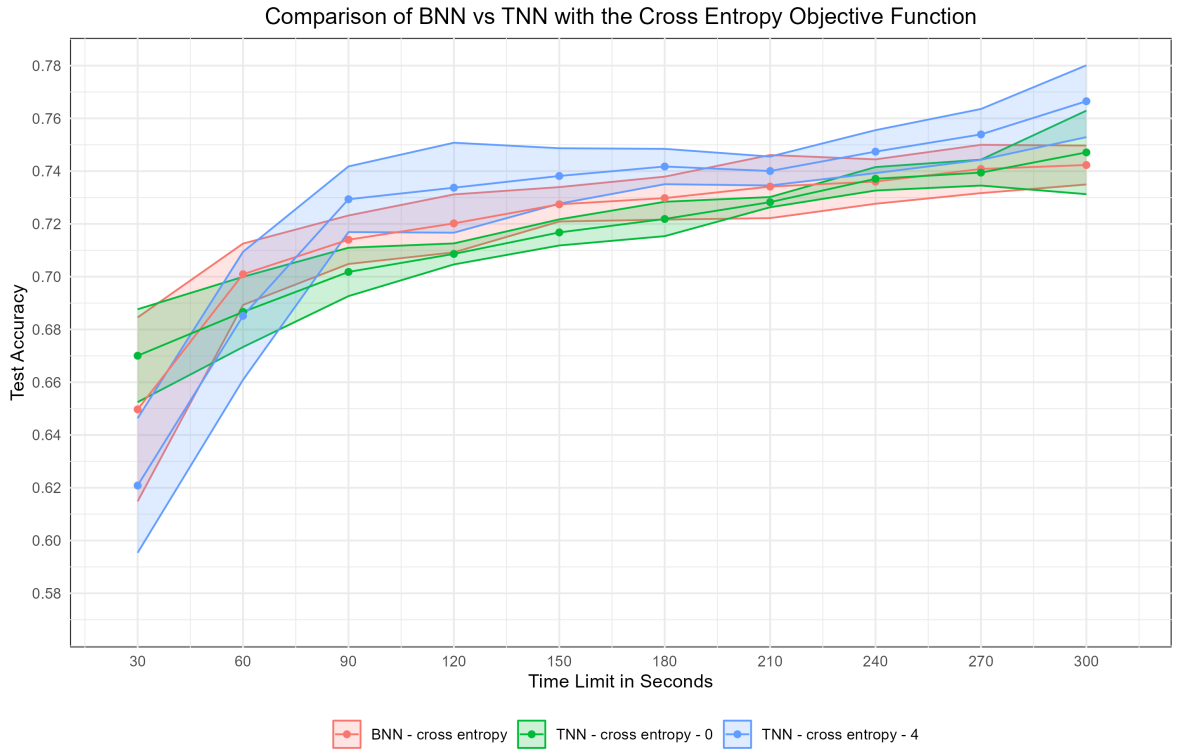

Figure 7: Test accuracies for the MNIST dataset. The networks are trained on a single batch with 2000 samples, 200 for each digit. The results are for a neural network with a single hidden layer with 16 neurons. The labels indicate what type of network is trained and what the regularization parameter is. The figure shows the mean accuracy of 5 runs as a line and the standard deviation as a shaded area around it.

6.4 The BeMi Ensemble

The implementation also support training binary classifiers. As a result it is possible to implement the BeMi ensemble introduced by Maria Bernardelli et al. (2023). In their paper, they test their ensemble for two network architectures, both with two hidden layers. One of them has 4 neurons in both hidden layers, while the other has 10 in the first and 3 in the second. I will try to train networks with the same structure as them, i.e. 10 neurons in the first hidden layer and 3 in the second hidden layer. Besides from this structure, I also train a network with a single hidden layer with 10 neurons. I hypothesize that this architecture is better, as the range of the preactivation values for the neuron in the last layer is larger. Recall, that the BeMi emsemble works by training a binary classifier for each pair of classes. For MNIST, this means that 45 networks

must be trained. Again, I use balanced batches to make a fair comparison against the existing literature. Maria Bernardelli et al. (2023) report their best average accuracy on MNIST to be 81.66 %, using 40 images per digit and a total training time of 7.5 hours, as each of the 45 binary classifiers is trained for 600 seconds.

The remaining of this section is structured as follows: At first I start by testing the different objective functions against each other. Afterwards I explore the importance of training data and the effect of the time limit. Lastly, I move on from BNNs to TNNs and see if it is possible to improve on the BNN results. The original BeMi ensemble is trained on TNNs.

6.4.1 Comparing Objective Functions

For binary classifiers, there is only one neuron at the output layer, so across objective functions the goal is the same, but the values of the objective functions can differ, which might lead to different results. I start by testing the different objective functions. I test with the two network architectures described above. I also test with both 10 and 100 images per digit and with a time limit for each binary classifier of 5 and 10 seconds. The results can be seen in Table 11, where as expected the objective functions give very similar results. For the network with a single hidden layer, the cross entropy gets the highest mean accuracies in 3 out of 4 cases and in the third it is only beaten by 0.02 % by the Brier objective function, which reaches a mean accuracy of 86.59 % with 100 images per digit and 10 seconds of training time per classifier, a total training time of 450 seconds. For all configurations the additional training time gives slightly better results, but in general the classifiers are much faster to train than in the work of Maria Bernardelli et al. (2023), despite using more data.

ObjFunc	Images	Time	Mean1	SD1	Mean2	SD2
brier	100	225	0.5643	0.020748	0.3915	0.017008
brier	100	450	0.5766	0.024475	0.3857	0.035266
brier	1000	225	0.8569	0.004764	0.7417	0.010035
brier	1000	450	0.8659	0.004340	0.7470	0.013553
cross-entropy	100	225	0.5708	0.028872	0.3720	0.021409
cross-entropy	100	450	0.5851	0.029226	0.3735	0.011863
cross-entropy	1000	225	0.8582	0.007717	0.7442	0.017460
cross-entropy	1000	450	0.8657	0.005599	0.7483	0.012503
integer	100	225	0.5709	0.029312	0.3733	0.025777
integer	100	450	0.5804	0.029027	0.3844	0.015866
integer	1000	225	0.8580	0.005094	0.7282	0.010162
integer	1000	450	0.8633	0.004024	0.7287	0.009162

Table 11: Mean accuracies for the BeMi ensemble. Mean1 and SD1 is for a BNN with a single hidden layer with 10 neurons. Mean2 and SD2 is for a BNN with a hidden layer with 10 neurons followed by a hidden layer with 3 neurons. The training time is the total training time, so each of the 45 networks are trained for 5 and 10 seconds respectively. The number of images, is the total number of images, so there is 10 and 100 images for each digit respectively.

6.4.2 Testing the Effect of More Training Data

One of the limitations of the MIP model that the BeMi ensemble was originally trained on is the amount of data it can handle. In a LS context, this limitation does not apply in the same way, so it is possible to train on more training data. Figure 8, shows the importance of more training data, as the test accuracy quickly increases. The figure is with a time limit of only 5 seconds, so it is highly likely that increasing the time limit will yield a even higher accuracy.

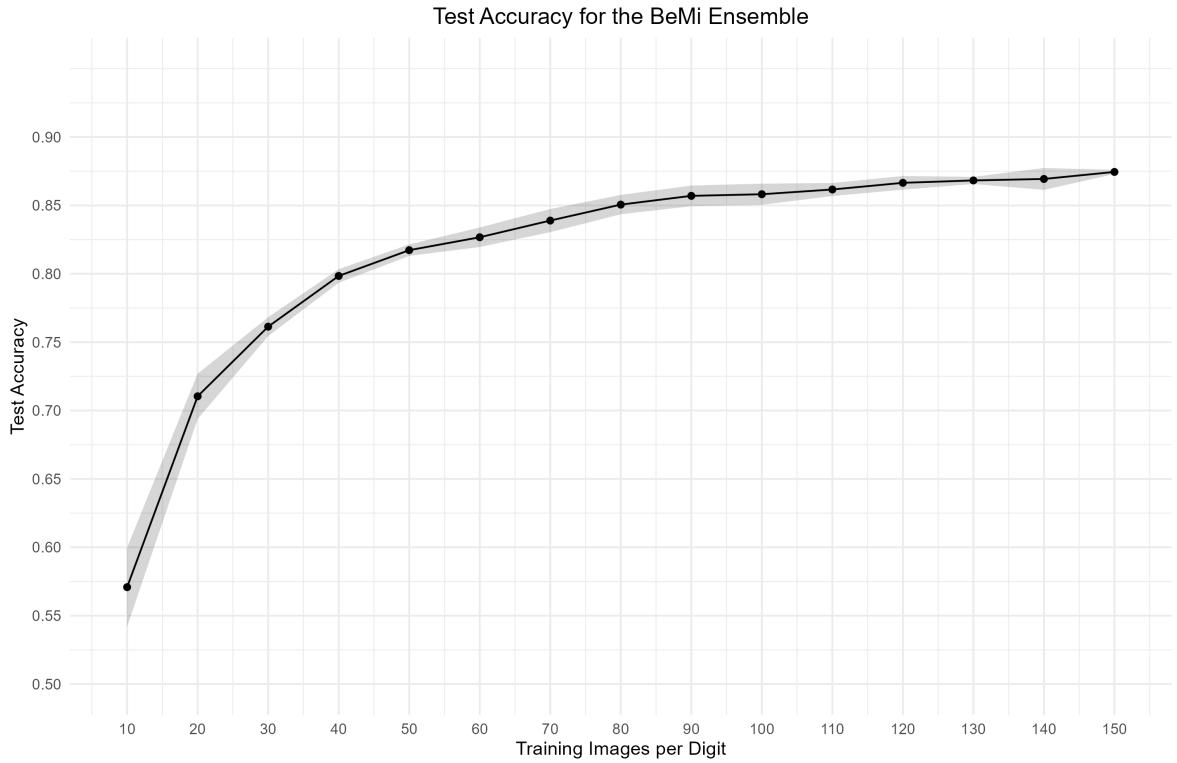

Figure 8: The test accuracy for the BeMi ensemble as the amount of training data increases. The network is a BNN with a single hidden layer with 10 neurons. Each binary classifier is trained for 5 seconds, giving a total training time of 225 seconds.

6.4.3 Testing the Effect of the Time Limit

To test how much the results improve as the time limit increases I fix the amount of training data to 100 images per digit and let the time limit vary. Table 12 shows, that increasing the time limit improve the accuracy slightly.

TrainingTime	Mean	SD
225.00	0.8568	0.006649
450.00	0.8657	0.005599
675.00	0.8669	0.005870
900.00	0.8684	0.005119

Table 12: Summary statistics for the BeMi ensemble for different time limits. The network is a BNN with a single hidden layer with 10 neurons and the training is based on 100 images per digit.

6.5 Testing on Other Datasets

So far I have only used the MNIST dataset. In this section I look further and test also on the more challenging Fashion-MNIST (FMNIST). This dataset also has 60,000 examples in the training set and 10,000 in the test set. Similarly to MNIST, the FMNIST consists of 28×28 grayscale images, but this time of Zalando articles. There is 10 different labels so it is again a multi-classification problem.

7 Conclusion and Future Work

References

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized Neural Networks. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., editors, Advances in Neural Information Processing Systems, volume 29. Curran Associates, Inc.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, Advances in Neural Information Processing Systems, volume 25. Curran Associates, Inc.

Maria Bernardelli, A., Milanesi, S., Gualandi, S., Chuin Lau, H., and Yorke-Smith, N. (2023). Multi-Objective Linear Ensembles for Robust and Sparse Training of Few-Bit Neural Networks. INFORMS Journal on Computing, ??(??).

Min, E., Ball, M., Huang, G., Jain, S., Karnati, H., Satopaa, V., and Zhang, G. (2018). Opportunities and obstacles for deep learning in biology and medicine. Journal of the Royal Society Interface, 15(20170387):1–47.

Thorbjarnarson, T. and Yorke-Smith, N. (2023). Optimal Training of Integer-valued Neural Networks with Mixed Integer Programming. PloS one, 18(2).

Toro Icarte, R., Illanes, L., P. Castro, M., A. Cire, A., A. McIlraith, S., and Beck, J. C. (2019). Training binarized neural networks using MIP and CP. International Conference on Principles and Practice of Constraint Programming, 11802.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention Is All You Need.

Yuan, C. and Agaian, S. S. (2023). A comprehensive review of Binary Neural Network. Artificial Intelligence Review, 56(11):12949–13013.