Opleiding DSAI

Hypernetworks

for binary neural networks

Elyanne Oey

Supervisors:
Evert van Nieuwenburg & Felix Frohnert

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/08/2023

Abstract

Binary neural networks are neural network with binary weights and activations. The weights and activations only have two possible values, and only one bit is needed to represent the values. This can be very useful for devices that have limited computational power and memory. This paper discusses the performance of a binary neural network on two classification problems, compared to the performance of a full precision neural network. Finding the optimal parameters through training a neural network with backpropagation is not the only way. Another way is through the use of hypernetworks. Those are neural networks that find the weights of another network, called the target network. This thesis discusses how the weights of a binary neural network, which functions as the target network, can be found using a hypernetwork.

Contents

1 Introduction

The interest in artificial intelligence has grown immensely the last century [MSKE22]. Artificial intelligence is a science that seeks to make machines that can perform tasks that usually require human intelligence [M+07]. An artificial neural network (ANN) is an artificial intelligence technique that is widely used to solve intelligent tasks. ANNs are inspired by the biological neural networks in a brain. ANNs have already been able to perform all kind of different tasks. For a long time they have been able to classify images [EPdH02]. For example, ANNs can classify skin cancer images [Elg13], which allows for early detection, since it is not an expensive technique to use. ANNs have also been used to solve natural language processing problems, such as speech recognition [HDY+12] and text-to-speech [ACC+17]. Even though, artificial neural networks are very useful and effective, training them requires a lot of computational power and memory [SL19]. This makes them hard to use one small devices, such as mobile phones. One solution to this issue is the use of binary neural networks. Binary neural networks have binary weights and activations [CHS+16]. Those binary weights and activations can only have two possible values, which usually are -1 and 1. They only need one bit to represent the values, unlike full precision neural networks, which usually have 32-bit floating point values. Thus binary neural networks take up less memory. Also, because the weights and activations are only one bit values, some mathematical operations, that happen inside artificial neural networks, can be brought down to bitwise operations [SL19]. Thus the computational power needed also becomes less. The use of binary neural networks can be very beneficial. However finding the optimal parameters, hyperparameters and architectural designs stays a complicated task. Recently, the use of so-called hypernetworks has become popular. Hypernetworks are neural networks that generate weights of other neural networks [HDL16]. Hypernetworks can be smaller than the network whose weights the hypernetwork is trying to find. This means that less weights need to be trained [CZL+23]. In a recent study conducted by Carrasquilla et al., they made a quantum hypernetwork, which finds the optimal parameters, hyperparameters and architectural design of binary neural network [CHAI+23]. The quantum hypernetwork is a parameterized quantum circuit, which outputs the weights of the binary neural network upon measurement. This inspired us to study how the weights of a binary neural network can be found using a hypernetwork.

The main contributions of this thesis are:

- We evaluate the performance of a binary neural network that needs to solve two different classification problems. And we compare this to the performance of a full precision neural network.
- We demonstrate how the weights of a binary neural network can be found with a hypernetwork. And we investigate how well the binary neural network performs when it uses the weights found by the hypernetwork.

1.1 Thesis overview

The github repository containing the code discussed in this paper can be found with the following link: https://github.com/ElyanneOey/bachelor_thesis

This paper has the following structure. This chapter contains the introduction; Section 2 includes the definitions; Section 3 discusses related work; Section 4 describes the experiments, including the data and the methods used; Section 5 discusses the found results; Section 6 concludes and discusses possible future research topics. This bachelor thesis at LIACS was supervised by Evert van Nieuwenburg and Felix Frohnert.

2 Definitions

In this section we discuss the inner workings of artificial neural network, binary neural networks and hypernetworks.

2.1 Artificial neural networks

An artificial neural network (ANN) is an mathematical model inspired by biological neural networks [Suz11, Abr05]. Biological neural networks, like the human brain, consists of individual units, biological neurons [Ber14]. The basic structure of a neuron consists of three parts: the cell body, the dendrites and the axon. The cell body contains a nucleus. The dendrites receive signals from other neurons. And the axon sends signals to other neurons. The place where one axon ends and a dendrite of another neuron begins is called a synapse. There are excitatory synapses that promote firing. And there are inhibitory synapses, which inhibit the firing. All the signals, that reach the dendrites of one neuron, together is the total signal. If this signal is bigger than a certain threshold the neuron will fire [Ber14].

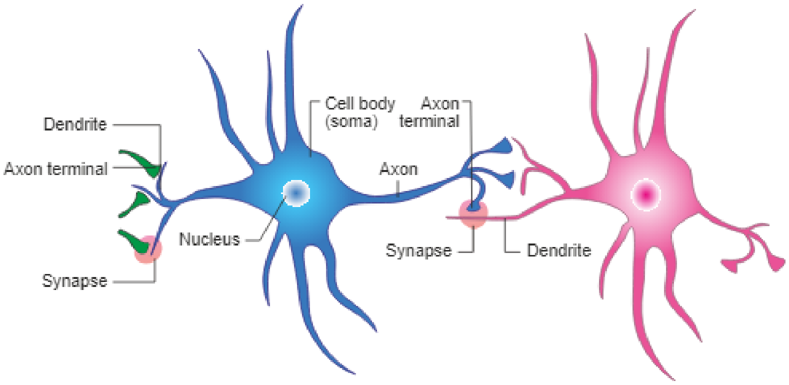

Figure 1: Structure of a neuron and the synaptic connection to another neuron. Source:https://www.chegg.com.

A network of neurons can work together to perform specific functions [Abr05]. Just like a biological neural network consists of individual units, an artificial neural network also consists of individual units, called artificial neurons, which mimic general features of a biological neuron [Ber14]. It is a simplified mathematical model of a neuron [Abr05].
Figure 2 shows the structure of an artificial neuron. An artificial neuron receives information through inputs. In an artificial neuron the input is multiplied by connection weights. Which can be seen as the synapses of the artificial neuron. The total input is the weighted sum of the inputs. A bias can also be added. The sum is then passed through an activation function (also called a transfer

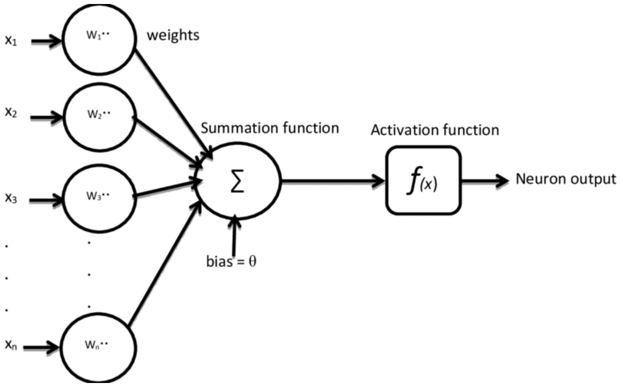

Figure 2: structure of an artificial neuron. Showing the mathematical equations that take place inside the neuron. Source: [YB18].

function). The result is the output of the neuron [Suz11]. Formula 1 shows the mathematical description of an artificial neuron.

$$y = f(\sum_{i=1}^{n} w_i \cdot x_i + b) \tag{1}$$

x_i is the input, that go from $x_1, ..., x_n$. w_i is the corresponding weight. b is the bias. And f is the activation function. There are different types of activation functions. Using no activation function would be equivalent to using a simple linear function [SSA20]. The output would be the same as the input to the activation function. However, to solve more complex tasks non-linear activation functions are needed, like the sigmoid, ReLU, tanh or softmax function. The sigmoid function is the most widely used activation function [SSA20]. Formula 2 shows the sigmoid function. The output of the function will be between 0 and 1. The bigger/more positive the input the closer to 1 the output will be. And the smaller/more negative the input, the closer to 0 the output will be. The output can be seen as the predicted probability of something belonging to a class [SSA20].

$$y = \frac{1}{1 + e^{-x}} \tag{2}$$

Where y is the output of the function. And x would be the weighted sum of the inputs.
By combing multiple artificial neurons we get an artificial neural network [Suz11]. These networks are able to perform more complex tasks, like image classification. In an artificial neural network, neurons are structured in layers. Most artificial neural networks consist of an input layer, zero, one or multiple hidden layers and an output layer, as depicted in figure 3. In a feed-forward network, the signals flow from one layer to the next. Thus the output of neurons in a layer is the input of the neurons in the next layer [Abr05]. Each neuron of one layer is connected to every neuron of the next layer. Each connection has a weight [Ber14]. The weight determines the strength of the connection between neurons [Goh95]. The input layer receives the information. For example, in the case of image classification each neuron in the input layer receives the color intensity of one pixel. And the output layer would have the number of neurons as there are classes. Each neuron would output the probability of the image belonging to that class. There are also other types of architectures artificial neural networks can have. Different types of architectures are able to perform better in

other kind of tasks. For example, recurrent neural networks are networks where at least one neuron forms a connection with itself. Recurrent neural networks are better at handling time-series data, such as speech recognition [SSB+18].

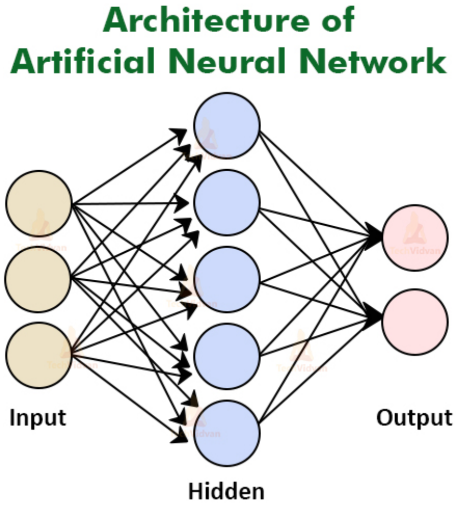

Figure 3: Architecture of an artificial neural network. Source:https://blog.knoldus.com.

In order to get an artificial neural network to give the desired output, the weights on the connections need the have the right value. So that in each neuron the right calculations can be made, to get the desired output. There are different ways to get the weights to have the right values. The most commonly used way is by training the artificial neural network through giving the network training examples, and changing the weights according to a learning rule [Abr05]. The most widely used training algorithm is backpropagation which uses gradient descent as learning rule [ENG20]. Before training, the weights of the network are initialized randomly. Training a neural network with backpropagation has two phases [Wyt93, ENG20]. There is the forward propagation phase. Which involves an example passing through the network to produce an output. And there is the back propagation phase. Which is where the 'learning' happens. Because we know the label of an example input, we know what the output should be. We can calculate an error, using an error-function, between the output and the actual label. An example of an error-function is the sum of squared errors [WTTI04]. The error-function can also be referred to as the loss function. We want to minimize the error. Gradient descent finds the right values of the weights by minimizing the error [ENG20]. It does this by calculating the gradients of error with respect to the weights, starting from the last layer and making its way through the network layer by layer. To calculate the gradient of a weight in on layer, you need the gradients of the weights in later layers [LSM+20]. The most straightforward way to update the weights using the gradients is by changing the weights proportional to the negative of the gradients, as shown in formula 3 [LSM+20].

$$W_{kj} = W_{kj} - \eta \frac{\partial E}{\partial W_{kj}} \tag{3}$$

Where W_{kj} is the weight on the connection between neuron k and j. η is the learning rate. And $\frac{\partial E}{\partial W_{kj}}$ is the gradient of the error with respect to W_{kj}.

This is an example of supervised learning. Learning the correct weights values can also happen through unsupervised learning and reinforcement learning [ZASS14]. We will only focus on supervised learning.

2.2 Binary neural networks

Typical artificial neural networks are high precision neural networks with 32-bit floating point values [SL19]. ANNs need a lot of storage and computational power. This makes them hard to use on small devices, such as mobile phones. People have proposed different techniques to resolve this issue. For example, by reducing the number of weights in a convolutional layer by using channel-wise separable convolutions, as proposed by Howard et al. [HZC+17]. Another way to resolve the issue, is by quantizing the neural network. Quantizing a neural network means going from a full-precision network to a low-bit network [YSX+19]. Binarization is an example of quantization. Binarized variables can only have two possible values, which usually are -1 and 1. ”Binary neural networks are neural networks with binary weights and activations at run time” [CHS+16]. The weights and activations are constrained to -1 and 1. When the weights and activations are binary, when multiplying them, which happens in the forward pass, the multiplication is brought down to bitwise operations [SL19]. This is a lot faster than multiplying multi-bit floating points.
To binarize the weights and activations different functions can be used. We used the *Sign*-function, as shown in formula 4.

$$x^b = Sign(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases} \tag{4}$$

x is the real value and x^b is the binarized variable. Binarizing the weights and activations with the *Sign*-function does create a problem when backpropagating, since the derivative of the *Sign*-function is zero almost everywhere. And if the gradient is zero, the weights will not be updated [CHS+16]. Since the weights will be updated by multiplying the gradient with the learning rate and subtracting that from the current weights. Thus learning will not take place. Also subtracting a value, in this case the gradient multiplied with the learning rate, is not possible with binary values. By keeping track of the real valued weights, this problem can be resolved. During the backwards pass the real valued weights can be updated. And during the forwards pass the real valued weights can be binarized with the *Sign*-function. A way around the gradient problem is by using a straight-through estimator [CHS+16]. The straight-through estimator approximates the gradient of the sign-function by passing over the gradient. For the *Sign*-function: $q = Sign(r)$ this would mean that if g_q is an estimator of the gradient $\frac{\delta C}{\delta q}$. $\frac{\delta C}{\delta q}$ is the gradient of the error with respect to q. Then according to the straight-through estimator the gradient $\frac{\delta C}{\delta r}$ (the gradient of the error with respect to r) would be:

$$g_r = g_q 1_{|r| \leq 1} \tag{5}$$

Thus the gradient of the error with respect to r is the gradient of the error with respect to q, thus passing over the gradient. $|r| \leq 1$ cancels the gradient if r is to large. If $|r| \leq 1$ is true, than $1_{|r| \leq 1}$ becomes 1, thus keeping the gradient. Else it becomes 0, meaning that the gradient becomes 0. This clips the real-valued weight to -1 or 1 if the updated real-valued weight is outside [-1,1]. Because the weights can actually only have two possible values. During the backwards pass, when the weights are updated, two possible things can happen. The weights stay the same or they are flipped. The real-valued weights can be seen as how hard it can be to flip the weight. Because the bigger the real-valued weight the bigger of a change they need to flip. Not clipping the real-valued weights can make them grow very large, this worsens the performance of the networks [CHS+16].

2.3 Hypernetworks

As mentioned before, there are different ways of finding the right values of the weights of neural networks. Hypernetworks are an example of this. Hypernetworks are neural networks that generate weights of other neural networks [HDL16], first introduced by Ha et al.. As discussed in section 2.1, usually a neural network is trained on a training dataset and the weights are updated using backpropagation. When using a hypernetwork, only the weights of the hypernetwork are trained. A hypernetwork receives an input vector C and it outputs the weights for a so called target/main network. The weights of the target network are then set to what the hypernetwork outputs. The target network receives input from the training dataset. The loss of the output from the target network is then used to update the weights of the hypernetwork. The hypernetwork, as introduced by Ha et al., is trained with end-to-end backpropagation together with the target network [HDL16]. Our approuch to using hypernetworks for finding the weights of a neural network is slightly different, and will be discussed in more detail in section 4.2.3.

There are several advantages of using a hypernetwork to train neural networks. For one, hypernetworks are usually smaller than the target networks. Thus hypernetworks have less weights that need to be trained [CZL+23]. This useful when the hardware is limited, because having to train less parameters, takes up less memory and processing power. Another advantages is that hypernetworks can also be used for transfer learning [CZM+23, vOHGS22]. Transfer learning is a technique that refers to training an already trained model on a different, but related, task [PY10]. The benefit being that the model can use the knowledge it has already learned on the new task.

Hypernetworks can be categorized depending on its input, output, and architecture [CZL+23]. The input vector can be task conditioned, data conditioned or noise conditioned. The inputs can be static, meaning that the inputs are predefined and fixed, or they can be dynamic, meaning that the inputs change usually depending on the input to the target network. Hypernetworks can output all the weights of a target network [SNFC21]. However, it can also output chunks of weights [CZM+23] or weights of a component (e.g. layer) [ZKSvO20]. The output of the hypernetwork can also be static or dynamic. This relates to whether the number of weights of the target network is fixed or not. There are four major groups hypernetworks can be categorized in based on their architecture [CZL+23]. The hypernetworks can be multi-layer perceptrons, convolutional neural networks, recurrent neural networks, and attention-based networks.

3 Related work

3.1 Quantum hypernetwork

In a paper published by Carrasquilla et al., they discuss making a quantum hypernetwork that finds the optimal parameters, hyperparameters and architecture of a binary neural network [CHAI+23]. There they used a parameterized quantum circuit, which outputted the weights of the neural network when measured. The output of the quantum circuit was also used to determine which hyperparameters to use, for example, which activation function, and to determine the architecture of the binary neural network. A quantum circuit consists of qubits, which are quantum bits. On those qubits are a sequence of elementary quantum operations performed [Mon16]. Upon measuring the quantum circuit the qubits collapse to either 0 or 1. The collapsed qubits can be used as the weights for the binary neural network, as shown in figure 4.

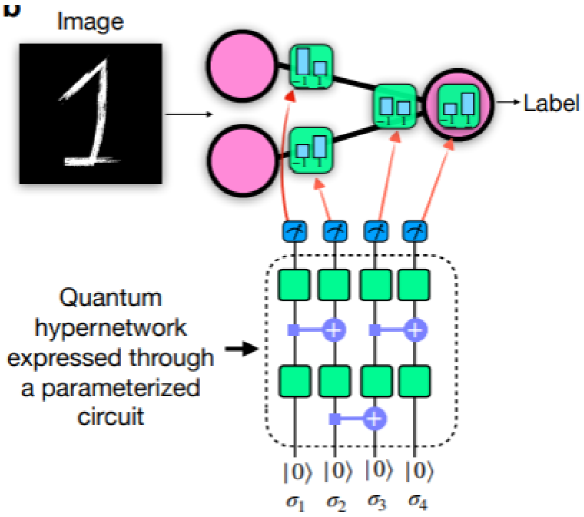
b
Image
Label
Quantum hypernetwork expressed through a parameterized circuit
$|0\rangle$ $|0\rangle$ $|0\rangle$ $|0\rangle$
σ_1 σ_2 σ_3 σ_4

Figure 4: Parameterized quantum circuit which functions as a quantum hypernetwork. Upon measuring the quantum circuit outputs the weights for a binary neural network. Source: [CHAI+23].

To optimize the parameters of the quantum circuit they needed to minimize the objective function, i.e. loss function. They did this using a gradient-based method [CHAI+23]. They used techniques to estimate the gradients of the objective function with respect to the parameters of the quantum circuit.

They found that the quantum hypernetwork was able to find the optimal parameters, hyperparameters and architectural design of a target network that could solve two toy classification problems. One was a two dimensional Gaussion dataset, and the other a resized version of the MNIST dataset. One big advantage of using a quantum circuit to find the optimal parameters, hyperparameters and architecture of a binary neural network is that the use of a quantum computer is much more energy efficient than a classical computer [VLB+20].

4 Experiments

The original goal of this paper was to create a quantum hypernetwork for binary neural networks ourselves. We decided to chose a standard architecture for the binary neural network, to make the task less complex. We chose to keep the binary neural network small, because a quantum circuit can only have a small amount of qubits (approximately confined to 20 qubits) [Mon16]. One qubit represents one weight, thus the number of weights that a quantum circuit can find is limited. Because we choose a standard architecture for the binary neural network, we first tested the performance of the binary neural network compared to a high precision neural network of the same size on two different datasets. We did not want to create a quantum hypernetwork immediately. We first wanted to make a hypernetwork that could find the weights of a binary neural network. We created a hypernetwork that can be swapped out with a quantum hypernetwork. And we tested the performance target networks that used the weights found by the hypernetwork. However do to time restrictions we were not able to also implement a quantum hypernetwork.

4.1 Data

For the experiments we used two different datasets. One is a toy-dataset, which we created ourselves. And the other dataset is the MNIST-dataset, which we resized to be 2x2.

4.1.1 Toy-dataset

The toy-dataset consists of 2x2 images in gray-scale. There are two different classes. The first class is the 'left-under-white' class. For all the images in this class, the pixel that is in the bottom left of the image has a value of 255, which shows as a white pixel. The other three pixels have a value between 0 and 254, which is assigned to them randomly. An example of an image from this class is shown in figure 5. The other class is the 'right-above-white' class. All the images in this class have a white pixel in the top right of the image. The other three pixels again have a value randomly assigned to them between 0 and 254. An example of an image from this class is shown in figure 6. Each class has 50000 images. The total dataset is split in a training set and test set, where the test set is 20% of the total dataset.

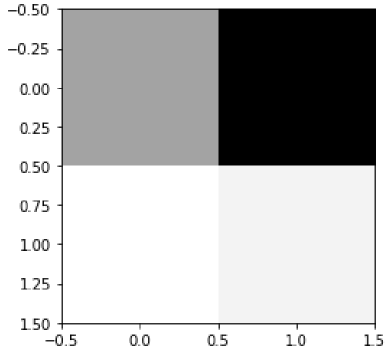

Figure 5: Example of an image of class 'left-under-white' class from the toy-dataset.

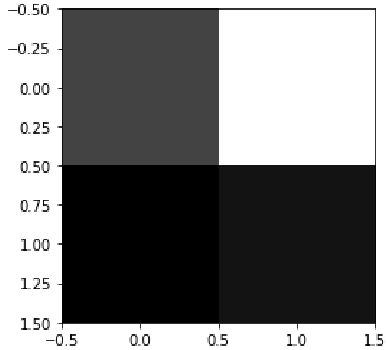

Figure 6: Example of an image of class 'right-above-white' class from the toy-dataset.

4.1.2 MNIST-dataset

The MNIST dataset is a dataset consisting of images of handwritten digits and their labels. Because we wanted to keep the neural networks small, we only used two classes. And we resized the images from 28x28 to 2x2. This does result in a loss of information. It is hard to make out which class the new image belongs too. However the neural networks will still be able to distinguish between classes. Figures 7 and 8 shows what the average of the images of two different classes looks like. We used images of the two classes that originally were of class with handwritten digits 0 and 1. We looked at the average pixel intensity of the images in each class (after standardizing them to be between -1 and 1). Figures 9 and 10 show a histogram of the average pixel intensity of each image. Classes 0 and 1 showed the most difference. However, there are a few images in each class that are identical to each other. The neural networks won't be able to distinguish to which class those images belong to. This restricts the maximum accuracy. In figure 9 there are about 600 images that have an average pixel intensity of -1 (after standardization). This means that those images are completely black. More than 5000 images of class 1 are also completely black. The circa 600 images of class 0 will most likely be classified as being from class 1. The class with label 0 has a total number of 5923 images. And the class with label 1 has a total number of 6742 images. So the dataset contains a total of 12665 images. Thus at least approximately 5% will be classified incorrectly.

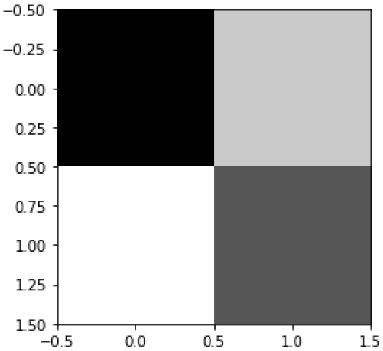

Figure 7: The average image of the images of the resized mnist dataset of class 0

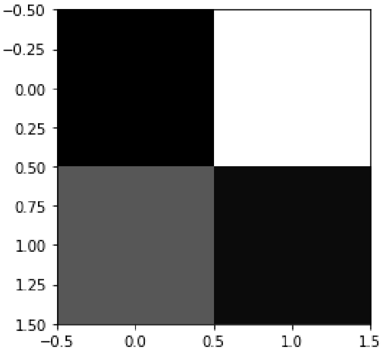

Figure 8: The average image of the images of the resized mnist dataset of class 1

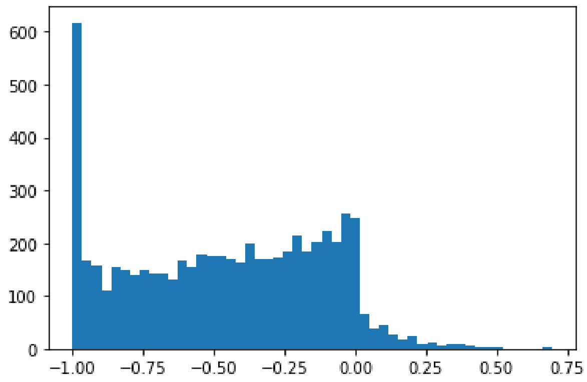

Figure 9: The histogram of the average pixel intensity of each image in class 0 after standardizing the pixel values to between -1 and 1.

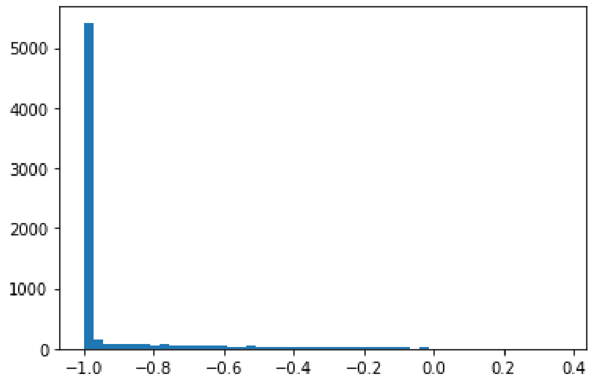

Figure 10: The histogram of the average pixel intensity of each image in class 1 after standardizing the pixel values to between -1 and 1.

4.2 Methods

4.2.1 High precision neural network

We created a high precision neural network with the Keras library in python. The network is a dense network. This means that each neuron of one layer is connected to all the neurons in the next layer. The network we created has three layers. Both datasets consists of images with 4 pixels. Thus the input layer has 4 neurons. The second layer has 6 neurons and the output layer has 2 neurons, because there are two classes. The structure of the network is shown in figure 11

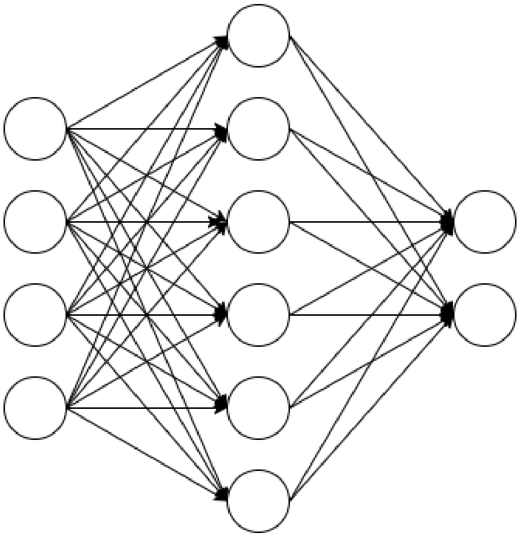

Figure 11: Structure of artificial neural network and binary neural network that we used for our experiments.

The activation function used in the second layer is the sigmoid function. And the activation function used in the last layer is the softmax activation function. The softmax function is a combination of sigmoid functions [SSA20]. The softmax function can be used for multi-class classification. The output of the sigmoid function will be between 0 and 1. The bigger/more positive the input the closer to 1 the output will be. And the smaller/more negative the input, the closer to 0 the output will be. The output can be seen as the predicted probability of something belonging to a class [SSA20]. The sigmoid function can be used for binary classification. The softmax function is shown in formula 6. It returns the probability for each class.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., k \tag{6}$$

Where z is the input vector and K is the number of classes. We did not use biases in the layers. To keep the amount of neurons in the hypernetwork small.

4.2.2 Binary neural network

To create the binary neural network, we used the open-source Python library Larq. Larq creates neural networks with extremely low-precision weights and activations [Plu]. Larq has quantized layers which are interchangeable with keras layers. These quantized layers, quantize the incoming activations and weights of the layer. The activations of the layer are computed according to the following formula:

$$y = \sigma(f(q_{kernel}(w), q_{input}(x)) + b) \tag{7}$$

y is the output, w is the real-valued weight, x is the input, f is the layer operation (i.e. summation of the multiplication of the weights and inputs), b is the bias, and σ is the activation function. In the Larq library the incoming activations are quantizited, instead of the output. They chose to do this to prevent unintended non-binary operations when using batch normalization layers [Plu]. One of the quantized layers Larq has is the QuantDense layer. This layer has an input_quantizer argument and a kernel_quantizer argument. These arguments determine how the weights and activations are quantized. If they are None, the QuantDense layer functions just like a Dense layer.
We created a binary neural network that has the same structure as high precision neural network. But instead of using a Dense layers, we used a QuantDense layers. We set the input_quantizer and kernel_quantizer to ste_sign. This binarizes the activations and weights according to formula 4. And it calculates the gradient with the straight-through estimator. This layer also has an argument called kernel_constraint which can be set to "weight_clip". This clips the weights to be between -1 and 1. We decided not to binarize the input. Because this hurts the accuracy.

4.2.3 Hypernetwork

We created a hypernetwork that generates the weights of a target network. This target network is a binary neural network and it has the same structure as binary neural network in section 4.2.2. As stated before, Ha et al. trained the hypernetwork with end-to-end backpropagation together with the target network [HDL16]. Even though only the weights of the hypernetwork are updated. We did not want the gradient flow to go through target network. Because our original goal was to make a quantum hypernetwork for binary neural networks. There the gradients do not flow through the binary neural network. Because making a quantum hypernetwork is very complicated we decided to first make a hypernetwork that is just a neural network, which can find the weights of a binary neural network. Where the gradients also do not flow through the binary neural network, but directly from the objective function through the hypernetwork. So our take on a hypernetwork is a little different than that from Ha et al..

The hypernetwork consist of multiple layers. The input layer has four neurons, because the input to the hypernetwork is the same as that to the target network. The succeeding layer with 16 neurons, this layer has a linear activation function. The next layer generates the weights of the target network. This layer has 36 neurons, since the target network has 36 weights. This layer has a custom activation function which binarizes the output to -1 or 1. This function also makes sure that the gradient is calculated according to the straight-through estimator. Then there is a lambda layer with a custom function inside it. Inside the custom function is the target network. The lambda layer returns the output of the custom function. That output is then used as the output of the hypernetwork. It is important to note that the gradients do not flow through the lambda layer, i.e. the gradients do not flow through the target network. An overview of the architecture of the hypernetwork and target network can be seen in figure 12

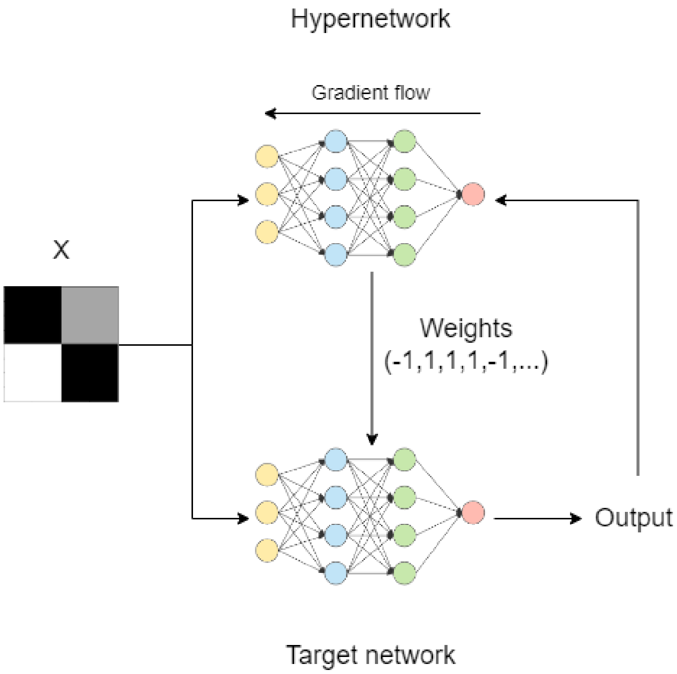

Figure 12: Overview of how the hypernetwork and the target network are connected. X is the input to the hypernetwork. Which generates the weights for the target network. The target network then classifies the input X. The output of the target network is used as the output of the hypernetwork.

The hypernetwork creates for each images a new set of weights. Which are then given to the target network. Which then classifies that image using the weights that are created from that image. After training the hypernetwork, one image can be given to the hypernetwork to get the final weights for the target network.

4.3 Experiment 1: Full precision NN vs. BNN

For our first experiment we tested how well the binary neural network performs compared to the full precision neural network. For this we used the full precision neural network and the binary neural network discussed in sections 4.2.1 and 4.2.2. We trained and tested the networks on both the toy-dataset and the resized MNIST dataset. The toy-dataset has in total 100000 images, the test set is 20% of the total dataset. The resized MNIST dataset with only class 0 and class 1 has in total 14780 images. Of which 2115 are from the test set, thus the test set is 14,3% of the total dataset.
Both models used 'adam' as optimizer with a learning rate of 0.0001 and 'categorical_crossentropy' as loss. We trained each model for 50 epochs and with a batch size of 75. We trained and tested each model ten times. As evaluation metric we used accuracy.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives and FN = False negatives.

4.4 Experiment 2: BNN vs. hypernetwork

In our second experiment we study how well, the found weights for the target network by the hypernetwork, can classify the test datasets. Again we tested the performance on both datasets. For the hypernetwork we used the 'adam' optimizer with a learning rate of 0.0001 and 'categorical_crossentropy' as loss. We trained the hypernetwork for 50 epochs and with a batch size of 32. Because the final weights of the target network depend on which input you give to the hypernetwork, as mentioned in section 4.2.3, we tested the performance of 500 different sets of weights, i.e. 500 different inputs to the hypernetwork, from both classes. The inputs used are from the training set. We again used accuracy as our evaluation metric.

5 Results

5.1 Experiment 1

Table 1 shows the accuracy of the full precision neural network and the binary neural network on both datasets. Overall, the full precision neural network has an higher accuracy for both datasets. This is not unexpected since there is a loss in information when the weights and activations are binary [SHXW19, II20]. The full precision neural network has an accuracy of 0.8016 ± 0.0080 on the resized MNIST dataset. The reason for the accuracy not being higher is because some of the resized MNIST images have some images that appear in both classes. Thus the networks won't be able to classify them correctly. Thus the accuracy can not be 1.000.
Another notable thing is that the accuracy of the binary neural network differs a lot on the resized MNIST dataset. We looked at different things that could be the reason for this. One of those reasons being that the random weight initialization effects the accuracy. To test this theory we set a seed when initializing the model. With the seed the model got the same accuracy each time we trained it (we trained it 10 times). This suggest that random weight initialization effects the

accuracy. With certain initializations the model could get stuck in a local minima resulting in different accuracies when running the model multiple times.

Models	Toy-dataset	resized MNIST dataset
Full precision neural network	0.9997 ± 0.0004	0.8016 ± 0.0080
Binary neural network	0.8440 ± 0.0242	0.6336 ± 0.1680

Table 1: Results of experiment 1: the table shows the accuracies of the full precision neural network and a binary neural network on the Toy-dataset and the resized MNIST dataset.

5.2 Experiment 2

5.2.1 Toy-dataset

Figures 13 and 14 show the accuracy of the target network on the toy-dataset test-set, given different inputs of the trainingset to the hypernetwork, resulting in different weights sets for the target network. Figures 15 and 16 show the accuracy ordered from high to low, to give a clearer picture. With most inputs the target network is able to perform as well as a traditionally trained network. The binary neural network trained the traditional way had an accuracy of 0.8440 ± 0.0242 on the toy-dataset. When finding the weights of the model with a hypernetwork it sometimes achieves an accuracy of around 0.9. However, about a fifth of the time the target network performs poorly, with accuracies sometimes even being around 0.2. We looked at what input resulted in high accuracies and what input in low accuracies. And it seems that input that clearly belongs to one class results in high accuracies. For example, an input that clearly belongs to class 'right-above-white' has a white pixel in the top right corner and the left under pixel is dark gray or black. And inputs that result in low accuracies are inputs where it would be harder to classify them. For example, an input that belongs to the class 'right-above-white' has a white pixel in the top right corner and the left under pixel is also very close to white. It does not make to seem much of a difference from which class the input belongs to.

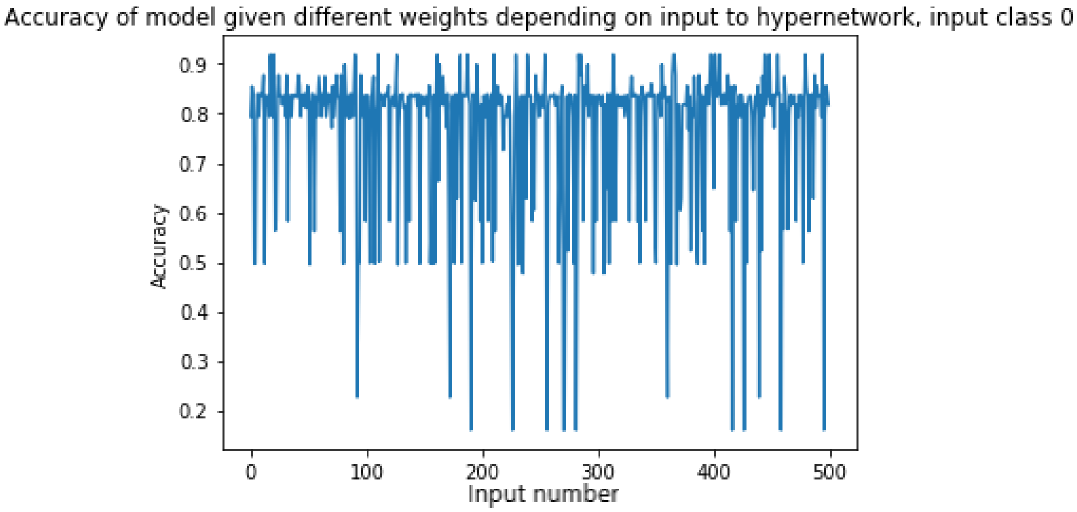

Figure 13: Accuracy of the target network on the toy-dataset test-set, given different inputs of the trainingset of class 'right-above-white' to the hypernetwork, resulting in different weights sets for the target network.

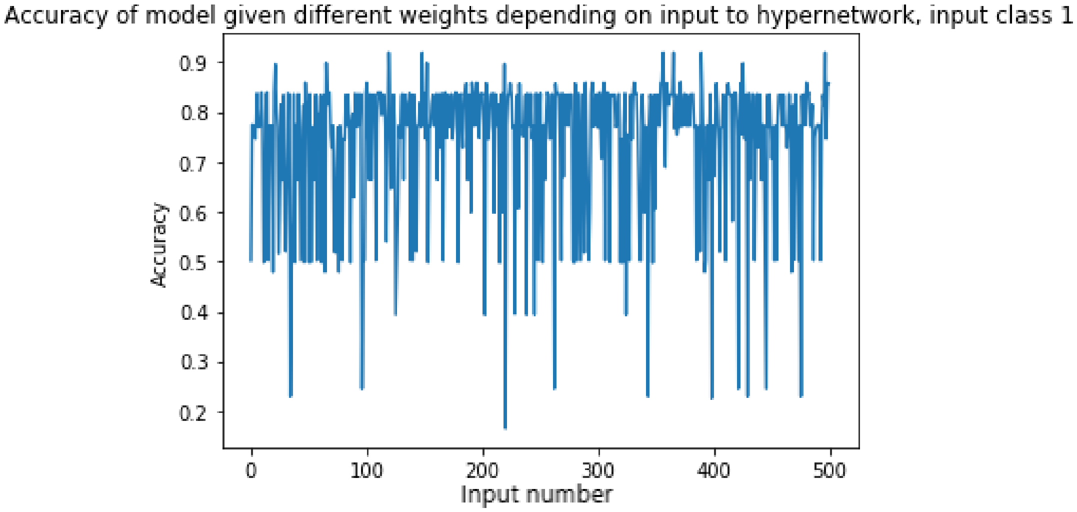

Figure 14: Accuracy of the target network on the toy-dataset test-set, given different inputs of the trainingset of class 'left-under-white' to the hypernetwork, resulting in different weights sets for the target network.

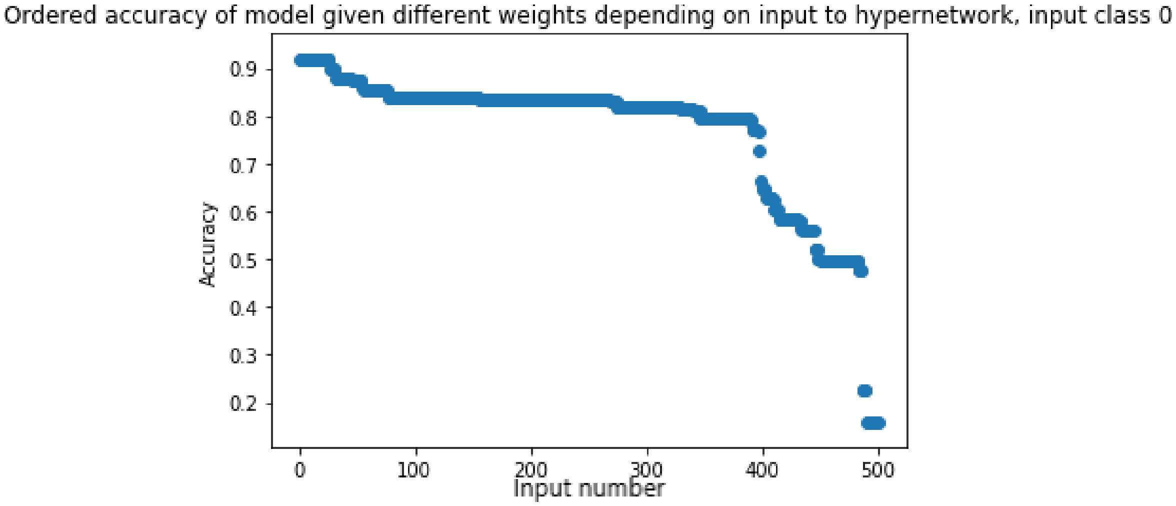

Figure 15: Ordered accuracy (from high to low) of the target network on the toy-dataset test-set, given different inputs of the trainingset of class 'right-above-white' to the hypernetwork, resulting in different weights sets for the target network. Each dot is one input.

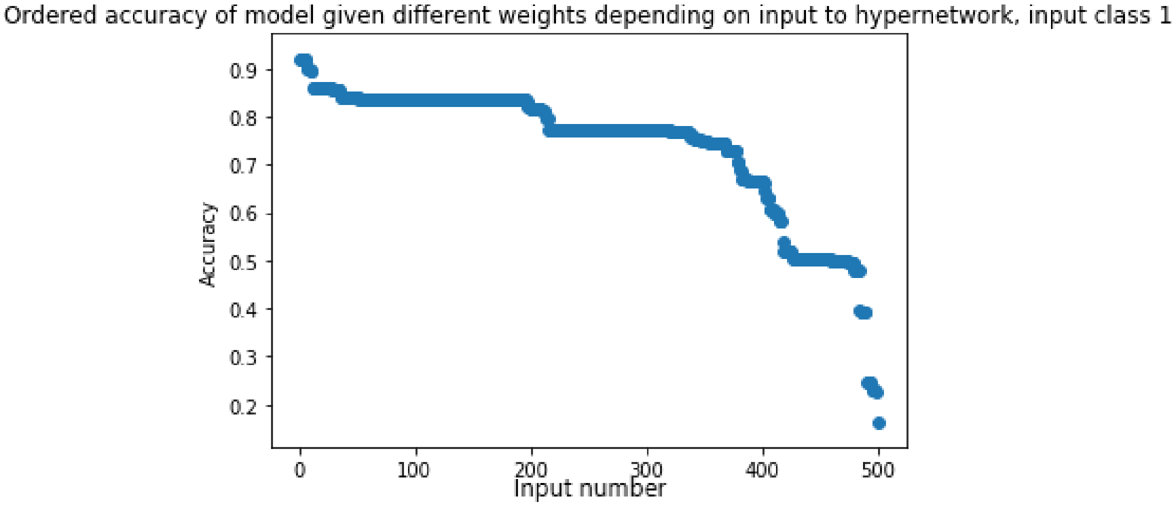

Figure 16: Ordered accuracy (from high to low) of the target network on the toy-dataset test-set, given different inputs of the trainingset of class 'left-under-white' to the hypernetwork, resulting in different weights sets for the target network. Each dot is one input.

5.2.2 Resized MNIST dataset

Figures 17 and 18 show the accuracy of the target network on the toy-dataset test-set, given different inputs of the trainingset to the hypernetwork, resulting in different weights sets for the target network. Figures 19 and 20 show the accuracy ordered from high to low, to give a clearer picture. With this dataset less inputs to the hypernetwork result in a high accuracy. Inputs from the class 0 result in low accuracies most of the time. Only about a fifth of the time it finds an accuracy that is higher than 0.6. For the inputs from class 1 a lot result in the same accuracy. We looked at those inputs and they are images that look the same. These images are all black, class 1 has a lot of these images. For this dataset it is harder to say which images would be easier to classify. So it is hard to say if inputs that result in high accuracies are easier images to classify.

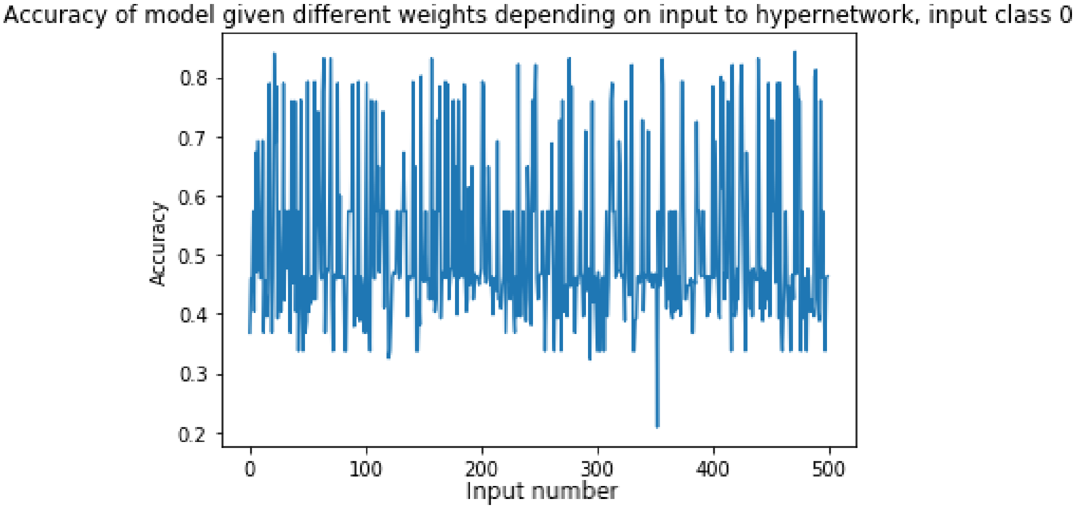

Figure 17: Accuracy of the target network on the resized MNIST test-set, given different inputs of the trainingset of class 0 to the hypernetwork, resulting in different weights sets for the target network.

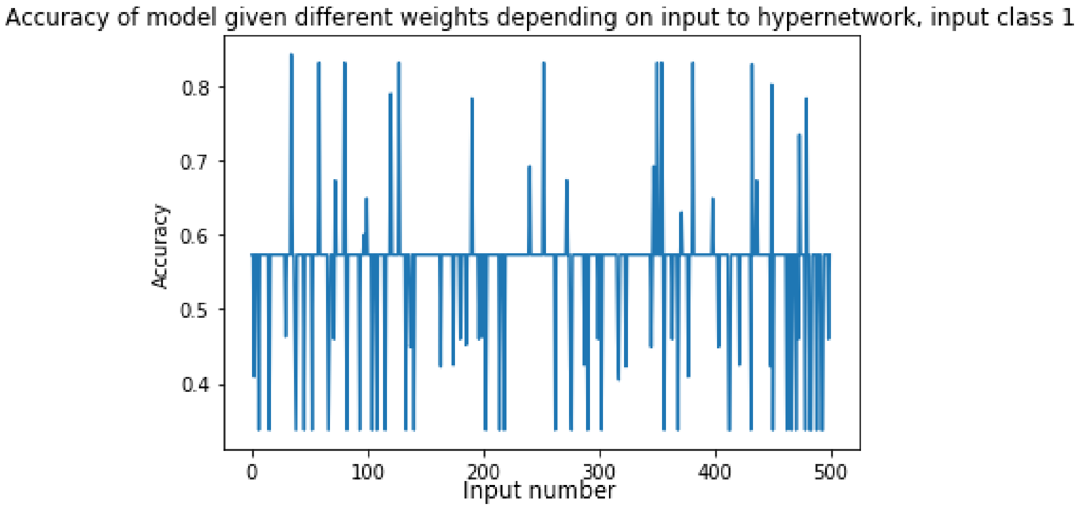

Figure 18: Accuracy of the target network on the resized MNIST test-set, given different inputs of the trainingset of class 1 to the hypernetwork, resulting in different weights sets for the target network.

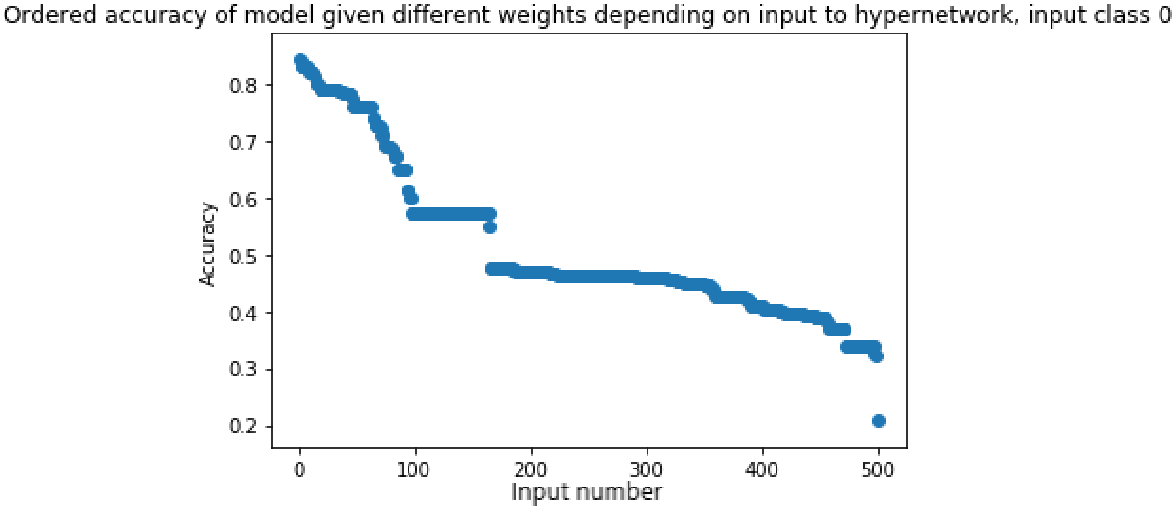

Figure 19: Ordered accuracy (from high to low) of the target network on the resized MNIST test-set, given different inputs of the trainingset of class 0 to the hypernetwork, resulting in different weights sets for the target network. Each dot is one input.

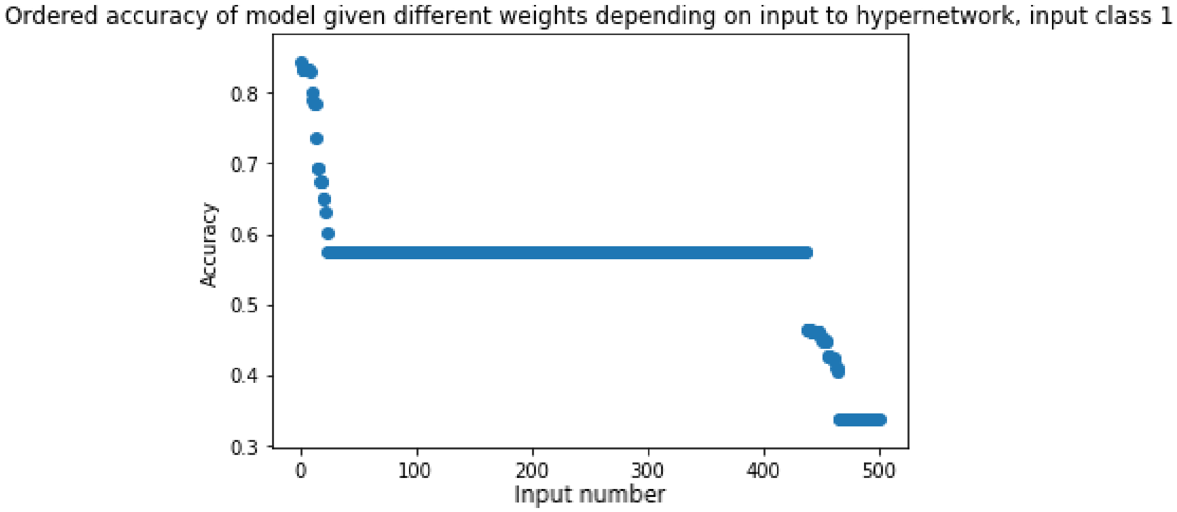

Figure 20: Ordered accuracy (from high to low) of the target network on the resized MNIST test-set, given different inputs of the trainingset of class 1 to the hypernetwork, resulting in different weights sets for the target network. Each dot is one input.

6 Conclusions and Further Research

In conclusion the binary neural network is able to classify the two datasets with reasonably high accuracies. Although the full precision neural networks does fastly outperform the binary neural network. With more hyperparameter tuning and/or a different architecture the binary neural network might be able to get higher accuracies. We found that the random weight initialization seems to have an effect on the accuracies of the binary neural network on the resized MNIST dataset. The hypernetwork that we created can find weights for a target network, that are able to achieve the same accuracies as when a network is trained the traditional way. However, not every input to the hypernetwork produces weights with which the target network is able to achieve such high accuracies. For the toy-dataset it seems that the inputs that would be easier to classify result in higher accuracies. This is harder to say for the resized MNIST dataset. It would be interesting to study the effect of the input to the hypernetwork on the accuracy of the targetnetwork further by trying other datasets.

There is a lot that can be studied further. For example, both networks can be expended. With a bigger binary neural network harder tasks can be solved. For example, classifying the original MNIST dataset. When expanding the binary neural network, more weights need to be generated, thus the hypernetwork should also be bigger. However, the hypernetwork can be smaller than the target network, which means that less weights need to be trained. It is also interesting to study whether a binary hypernetwork could be able to find the weights of a binary target network. Since our original goal was to create a quantum hypernetwork, which we were not able to do because of time constrictions, it would also be interesting to study how well our binary neural network performs with weights found by a quantum hypernetwork.

References

[Abr05] Ajith Abraham. *Artificial Neural Networks*, page 901–908. Wiley, 2005.

[ACC$^+$17] Sercan Ö. Arık, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, Shubho Sengupta, and Mohammad Shoeybi. Deep voice: Real-time neural text-to-speech. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 195–204. PMLR, 06–11 Aug 2017.

[Ber14] José Luis Bermúdez. *chapter eight: Neural networks and distributed information processing*, page 209–237. Cambridge University Press, second edition, 2014.

[CHAI$^+$23] Juan Carrasquilla, Mohamed Hibat-Allah, Estelle Inack, Alireza Makhzani, Kirill Neklyudov, Graham W. Taylor, and Giacomo Torlai. Quantum hypernetworks: Training binary neural networks in quantum superposition, 2023.

[CHS$^+$16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[CZL$^+$23] Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A. Clifton. A brief review of hypernetworks in deep learning, 2023.

[CZM$^+$23] Vinod Kumar Chauhan, Jiandong Zhou, Soheila Molaei, Ghadeer Ghosheh, and David A. Clifton. Dynamic inter-treatment information sharing for heterogeneous treatment effects estimation, 2023.

[Elg13] Mahmoud Elgamal. Automatic skin cancer images classification. *International Journal of Advanced Computer Science and Applications*, 4(3), 2013.

[ENG20] ANDRIES P. ENGELBRECHT. *Computational intelligence: An introduction*. JOHN WILEY, 2020.

[EPdH02] M. Egmont-Petersen, D. de Ridder, and H. Handels. Image processing with neural networks—a review. *Pattern Recognition*, 35(10):2279–2301, 2002.

[Goh95] A.T.C. Goh. Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering*, 9(3):143–151, 1995.

[HDL16] David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks, 2016.

[HDY$^+$12] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[HZC^{+}17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[II20] Dmitry Ignatov and Andrey Ignatov. Controlling information capacity of binary neural network. *Pattern Recognition Letters*, 138, 07 2020.

[LSM^{+}20] Timothy P. Lillicrap, Adam Santoro, Luke Marris, Colin J. Akerman, and Geoffrey Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, 2020.

[M^{+}07] John McCarthy et al. What is artificial intelligence. 2007.

[Mon16] Ashley Montanaro. Quantum algorithms: An overview. *npj Quantum Information*, 2(1), 2016.

[MSKE22] M. Mazzucato, M. Schaake, S. Krier, and J. Entsminger. Governing artificial intelligence in the public interest. *UCL Institute for Innovation and Public Purpose, Working Paper Series (IIPP WP 2022-12).*, Dec 2022.

[Plu] Plumerai. Getting started.

[PY10] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[SHXW19] Mingzhu Shen, Kai Han, Chunjing Xu, and Yunhe Wang. Searching for accurate binary neural architectures, 2019.

[SL19] Taylor Simons and Dah-Jye Lee. A review of binarized neural networks. *Electronics*, 8(6):661, 2019.

[SNFC21] Aviv Shamsian, Aviv Navon, Ethan Fetaya, and Gal Chechik. Personalized federated learning using hypernetworks, 2021.

[SSA20] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, 2020.

[SSB^{+}18] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks, 2018.

[Suz11] Kenji Suzuki. *Artificial Neural Networks - Methodological Advances and Biomedical Applications*. 04 2011.

[VLB^{+}20] Benjamin Villalonga, Dmitry Lyakh, Sergio Boixo, Hartmut Neven, Travis S Humble, Rupak Biswas, Eleanor G Rieffel, Alan Ho, and Salvatore Mandrà. Establishing the quantum supremacy frontier with a 281 pflop/s simulation. *Quantum Science and Technology*, 5(3):034003, apr 2020.

[vOHGS22] Johannes von Oswald, Christian Henning, Benjamin F. Grewe, and João Sacramento. Continual learning with hypernetworks, 2022.

[WTTI04] X.G. Wang, Z. Tang, H. Tamura, and M. Ishii. A modified error function for the backpropagation algorithm. *Neurocomputing*, 57:477–484, 2004. New Aspects in Neurocomputing: 10th European Symposium on Artificial Neural Networks 2002.

[Wyt93] Barry J. Wythoff. Backpropagation neural networks: A tutorial. *Chemometrics and Intelligent Laboratory Systems*, 18(2):115–155, 1993.

[YB18] Joseph Yacim and Douw Boshoff. Impact of artificial neural networks training algorithms on accurate prediction of property values. *Journal of Real Estate Research*, 40:375–418, 11 2018.

[YSX$^+$19] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[ZASS14] Magdi Zakaria, Mabrouka AL-Shebany, and Shahenda Sarhan. Artificial neural network: A brief overview. *Int. Journal of Engineering Research and Applications*, 4(2):7–12, Feb 2014.

[ZKSvO20] Dominic Zhao, Seijin Kobayashi, João Sacramento, and Johannes von Oswald. Meta-learning via hypernetworks. *NeurIPS*, 2020.