Use of Stochastic Switching in the Training of Binarized Neural Networks

Svante Åkesson
sv3084ak-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Mattias Borg

Examiner: Erik Lind

August 15, 2022

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Most prior research into the field of Binarized Neural Networks (BNNs) has been motivated by a desire to optimize Deep Learning computations on traditional hardware. These methods rely on bit-wise operations to drastically decrease computation time, however to address the resulting loss in accuracy it is common practice to reintroduce continuous parameters and train using batch normalization. Here a separate application for BNNs is investigated to find if it is a feasible application to build and train BNNs on hardware based on binary memristors with stochastic switching. To do this BNNs have been trained without batch normalization and multiple new algorithms have been proposed which attempt to utilize the stochastic switching ability in place of storing multiple bits of information per weight. The results show that BNNs without batch normalization are limited to lower accuracies than conventional CNNs, and it is essential to include an element of gradient accumulation for stochastic switching to work. To adress instability issues with stochastic switching based training an undo function is introduced which is shown to stabilize training well. Future research should look into accumulating gradients using stochastically switching memristors.

Popular Science Summary

Approach to Training Artificial Neural Networks using Randomness in Updates

When only allowing the connections between neurons to have two possible strengths it is no longer possible to improve the network by making small changes to the connections. Here, the proposed solution is to use random chance to instead make large changes with a low probability.

Artificial neural networks (ANNs) are a powerful tool for solving complicated programming problems such as pattern recognition in images. Inspired by the human mind, the data in ANNs is organized as neurons where different neurons are connected through synapses and the strength of these connections determines how the network will behave. Normally the strength of the connections is allowed to have many different values which allows for incremental updates, making small improvements to the network which accumulate over time, but the network can be simpler to implement if you only allow the connections to have two possible strengths.

The interest in training networks with two possible connection strengths (so called binarized networks) comes from a desire to build specialized hardware for training neural networks. It is not very energy efficient to simulate a network on a classical computer, but it is possible to build computer chips where data is structured similarly to neural networks with nodes sending data to their neighbours. If such a node can be represented by a single microscopic device this could simplify the hardware significantly and while it is possible to build such microscopic devices (called memristors) these memristors can oftentimes only have two different states, which means that they can only represent two connection strengths. This explains why it is interesting to train networks with two possible connection strengths but why use randomness when updating the connections? It just so happens that a couple of promising memristor technologies such as the Ferroelectric Tunnel Junction, and Spin-Transfer Torque Magnetic Memory, apart from allowing only two different states, also allow switching between these states with varying controllable probabilities.

This property inspired an investigation into training techniques relying on random updates, and the main technique used in this work relied on replacing the incremental change of the connections with small probabilities for large changes. The results showed that, while a network trained in this way significantly outperforms a random guess in classifying hand-written digits, with an 80% accuracy this falls quite short of the 99%

accuracy of a classic network.

As a curious consequence of the large random changes being made to the networks the risk was much higher that a good network could get broken by an unfortunate random update and the technique which was used to combat this was to give the training algorithm the equivalent of an undo button. With this, if an update turned out to be devastating for the network then it could just undo the update and try something else. This "undo button" was essential for training to be possible at all.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Deep Learning is a rapidly growing field. As the range of applications expand and error margins are squeezed ever tighter, the computational load continues to increase, requiring more and more computing power and memory-band-width both during training and when the network is in use[1]. This makes Deep Learning problematic to implement on systems with low power and compute resources.

To tackle this problem developments are being made in both software and hardware. Through the use of Binarized Neural Networks (BNNs), characterized by the use of binary weights and activation functions, it has been shown that computational load can be reduced by a factor of 23 without significant loss in accuracy when implemented on traditional hardware[2].

On the hardware side many developments have been made on dedicated architectures with the goal of optimizing neural network computations. The developments considered in this thesis target in-memory computing, where some hardware computations are performed locally inside the memory, reducing power consumption [3][4]. The reason for reduced energy demand in such systems is because memory states can be updated with small current spikes sent between neighbouring memory elements[3].

The hardware property in focus is memristors with stochastic switching, that is, memory which can hold binary states but where the state can be updated according to a stochastic rule.

Some examples of such memory elements are the Ferroelectric Tunnel Junction [5][6] and Spin-Transfer Torque Magnetic Memory [4].

In this thesis I investigate the feasibility of training BNNs using binary weights with stochastic switching in place of continuous floating point variables. To do this I propose a set of algorithms based on gradient descent, which attempt to address the problems which emerge from allowing such minimal precision during training. The feasibility of an algorithm is determined partially from its performance in training high accuracy networks but also from how well it utilizes the stochastic switching property in place of expensive floating point calculations.

Chapter 2
Deep Learning

2.1 Deep Neural Networks

Before introducing BNNs it is important to know the classic Deep Learning concepts of forward- and backpropagation. All networks considered in this thesis are feed-forward networks which means that information only flows in one direction through the network from input to output[7][8].

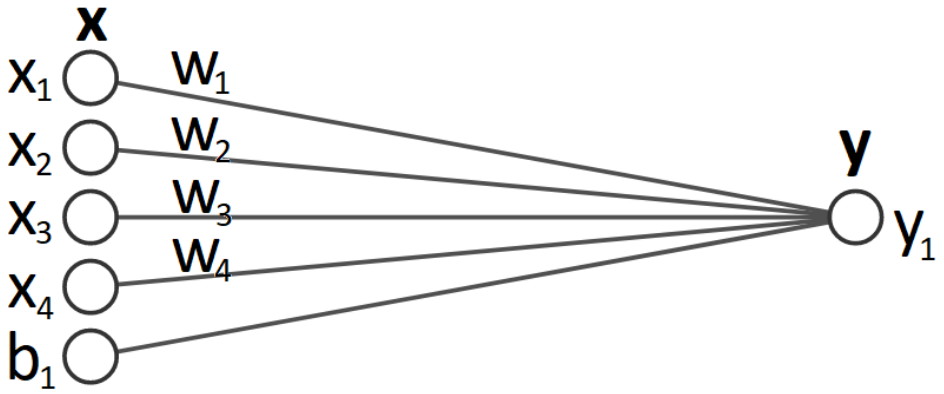

Figure 2.1: Illustration of a fully connected neural network with 4 input features, a bias and one output value.

One way to define a basic neural network is on the following form:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.1}$$

Here $\mathbf{x}$ is some input data on vector form. Every element of $\mathbf{x}$ has a neural connection with every element of $\mathbf{y}$[7] If $\mathbf{y}$ has only one element, like the one seen in figure 2.1, then $\mathbf{y}$ is simply a weighted sum of the elements in $\mathbf{x}$. These weights are stored in $\mathbf{W}$ and if $\mathbf{y}$ has more than one element then $\mathbf{W}$ is a matrix describing all possible connections between $\mathbf{x}$ and $\mathbf{y}$. The elements of vector $\mathbf{b}$ are called biases which allow $\mathbf{y}$ to be non-zero even if $\mathbf{x}$ is the zero vector.

Far from being a general function the matrix multiplication in eq. 2.1 can only describe linear relationships between $\mathbf{x}$ and $\mathbf{y}$. One idea to increase the complexity of the function is to add more layers between the input and output. To increase generality however it is not enough to simply introduce a hidden intermediate layer with another matrix multiplication as this is mathematically indistinguishable from the original case. Proof:

$$\mathbf{h} = \mathbf{W_1 x} + \mathbf{b_1},\ \mathbf{y} = \mathbf{W_2 h} + \mathbf{b_2} \Rightarrow \tag{2.2}$$

$$\mathbf{y} = \mathbf{W_2 W_1 x} + \mathbf{W_2 b_1} + \mathbf{b_2} \Rightarrow \mathbf{y} = \mathbf{W'x} + \mathbf{b'}$$

If the true relationship between $\mathbf{x}$ and $\mathbf{y}$ is non-linear then the network requires a non-linear step after matrix multiplication, commonly referred to as the activation function[7][8]. If $\mathbf{h}$ is passed through an activation function then the result in eq. 2.2 is no longer true.

Two common activation functions are tanh and the so called rectified linear unit or ReLU[8]:

$$\text{ReLU}(x) = \{\, 0\ x < 0 x x \geq 0 \tag{2.3}$$

The input to the activation function, $\mathbf{h}$ is typically referred to as the pre-activation and the output is referred to as the activation, e.g. notated as $\mathbf{a}$ except for in the final layer where the activation is simply the output $\mathbf{y}$[9].

The aim in deep learning is to determine the optimal values for the weights so that any input data $\mathbf{x}$ gives a $\mathbf{y}$ as close as possible to the desired output[8]. This requires training data which consists of input vectors $\mathbf{x}$ and the corresponding desired output $\mathbf{d}$[8]. The training problem is then framed as minimizing the error in the output, and in deep learning this error is typically referred to as the loss[8]. There are different types of loss functions which can be preferable in different circumstances. For example the mean squared error can be used as a loss function, however for classification problems with multiple possible classes it is common to use the categorical cross-entropy loss:

$$L(\mathbf{y}, \mathbf{d}) = \sum_{i=1}^{n} \sum_{j=1}^{m} -d_j^i log(y_j^i)/n \tag{2.4}$$

Where n is the number of samples in the data set and m is the number of elements in $\mathbf{y}$ and $\mathbf{d}$. The categorical cross-entropy is used for classification problems and assumes that $\mathbf{y}$ is in the interval $0 < y_j < 1$, interpreted as the certainty that the input x_i belongs to class j. It also assumes that the target $d_j^i = 1$ for only one j per i, the correct class for sample i. All other elements of d^i are 0. This means that the inner sum will only have one non-zero element. For this thesis all networks will be trained with the categorical cross-entropy, as it has been shown to lead to faster training than the mean squared error for classification problems[9].

To ensure that $0 < y_j < 1$ and $\Sigma_{j=1}^{m} y_j = 1$ for classification problems the softmax function is used as activation function in the final step, to normalize y[8]:

$$y_j = softmax(h_j) = \frac{e^{h_j}}{\Sigma_{k=1}^{m} e^{h_k}} \tag{2.5}$$

As a consequence $\mathbf{y}$ can be interpreted as a list of probabilities that a given input $\mathbf{x}$ belongs to a certain class j.

2.2 Convolutional Neural Networks

Convolutional neural networks function very similarly to the networks seen in the previous section but instead of matrix multiplications in each layer discrete convolutions are used[9]. This has the benefit of reducing the number of weights required for large input vectors. It also takes advantage of the fact that for images, nearby pixels are more strongly correlated than distant pixels, and the same local patterns may be useful to detect in any part of the image[8]. Let's first consider a 1-dimensional discrete convolution which is defined in the following way[7]:

$$h(t) = f(t) * g(t) = \sum_{\tau=-\infty}^{\infty} f(\tau) \cdot g(t-\tau) \tag{2.6}$$

In general the two functions f(t) and g(t) need to be defined for all integer-values on the interval $[-\infty, \infty]$. In practice however f and g can be set to zero outside of a finite interval which results in a finite sum, and h(t) then also becomes non-zero for only a finite interval. In convolutional neural networks, only the non-zero elements are stored which means that a finite amount of data is required.

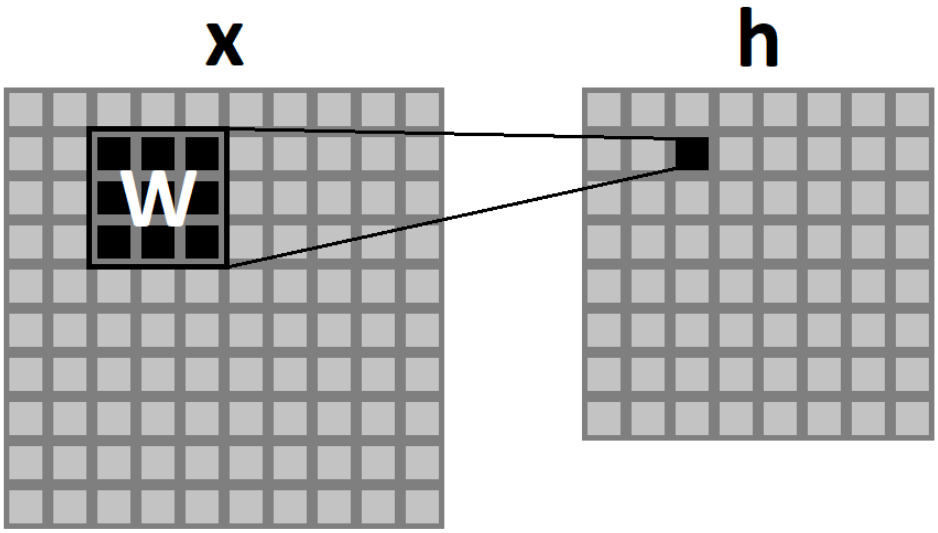

Figure 2.2: Illustration of a single filter in the convolutional layer.

From now on we assume that the input data $\mathbf{x}$ has the form of a 28 by 28 pixel image. Mathematically, let $\mathbf{x}$ be represented as a 3-dimensional tensor of dimensions (28,28,1), where the first two values are the width and height of the image and the third is the number of stacked images. In the input layer there is only one image but from this image multiple output images called *feature maps* will be generated and these are then sent in a stack to the next layer[9]. Values outside the volume of defined pixels are interpreted to be zero.

In the simplest case the pre-activation for the first hidden layer can be written on the following form with a single 3-dimensional convolution:

$$\mathbf{h_1} = \mathbf{x} * \mathbf{W_1} + \mathbf{b_1} \tag{2.7}$$

W_1 is a kernel, sometimes referred to as a filter. It is also 3-dimensional, but notably smaller than the input to the layer. It could for example have the dimensions (3,3,1). The b_1 tensor contains the biases. It may be computationally beneficial to stray away from the exact mathematical definition of a convolution when defining h_1. One property of a correct convolution is that h_1 would require a larger tensor than $\mathbf{x}$ to store its potential non-zero elements. This is because the (28,28,1) image and the (3,3,1) kernel share some overlap in a (30,30,1) volume. It may be more useful to only include output where the entire kernel overlaps with the image. This ensures that h_1 has a smaller dimension, (26,26,1) in this case. Other ways of decreasing the size of h_1 is to take larger strides during convolution which means that the filter is moved more than a single pixel at a time.

There are severe limitations to what information a single (3,3,1) filter can extract from an image and therefore a single layer usually has multiple filters. For the example above each kernel may output a 26 by 26 feature map. These feature maps are then stacked on top of each other which results in a h_1 of e.g. dimensions (26,26,10) if 10 filters are used. For the layer that follows each kernel may then have the shape e.g. (2,2,10) which allows information from all 10 feature maps to be combined.

2.2.1 Max-pooling

Another important step for CNNs is sub-sampling in the form of max-pooling[8] This technique reduces the size of the images but it also makes the network more flexible to slight positional shifts in the input data[9]. It works similarly to a convolution in the sense that it iterates a quadratic region over the image, but instead of multiplying the pixel values with weights and summing the terms together it simply finds the maximum pixel value and sends it forward. One possible max-pooling size is a 2 by 2 area.

Returning to the example above, after the output h_1 has been computed it has the dimensions (26,26,10). Max-pooling is performed separately on each of the 10 images. If the max-pooling size used is 2 by 2 and the stride is 2 then the data will be reduced to size (13,13,10) after max-pooling.

The robustness towards shifts in input data comes from the fact that the maximum pixel value in a 2 by 2 area may be unchanged even if the pixels are shifted one step in some direction[9].

2.2.2 Flattening

At some point down the line in a CNN a classification must be made; the probabilities for each class y_j have to be computed. This is done by so called flattening which simply means that the output from the last max-pooling layer is reorganized on vector form and sent into one or more fully connected layers eventually connecting to the output $\mathbf{y}$[9][8].

2.3 Training the Network

2.3.1 Back-propagation

The conventional method for training deep neural networks is with gradient descent[7][8][9]. The details of how the gradient is computed are beyond the scope of this thesis however the principles are as follows.

To maximize the ability of the network to predict the correct class $\mathbf{d}$ given some input data $\mathbf{x}$ is equivalent to minimizing the loss function with respect to the weight variables (and biases) in the network [7][8][9]:

$$\min_{\mathcal{W}} L(y(x, \mathcal{W}), d) \tag{2.8}$$

Here, let $\mathcal{W}$ denote all weights and biases in the network and let L be the categorical cross-entropy loss function.

Back-propagation is a technique for calculating gradients efficiently in Deep Learning. The back-propagation process emerges directly from the chain-rule when computing the gradient of the loss function[7]

It is called back-propagation because the gradient computations start at the loss function and propagate backwards toward the input layer. For example, consider a weight $w_{k,l}$ in the last layer of a network, where k denotes the row and l the column of $w_{k,l}$ in the matrix W. The partial derivative with respect to $w_{k,l}$ can be computed through the chain rule[7]:

$$\frac{\partial}{\partial w_{k,l}} L(y, d) = \sum_{j=1}^{m} \frac{\partial}{\partial y_j} L(y, d) \frac{\partial y_j}{\partial w_{k,l}} = \tag{2.9}$$

$$\sum_{j=1}^{m} \frac{\partial}{\partial y_j} L(y, d) \frac{\partial y_j}{\partial h_k} \frac{\partial h_k}{\partial w_{k,l}} = \sum_{j=1}^{m} \frac{\partial}{\partial y_j} L(y, d) \frac{\partial y_j}{\partial h_k} a_l$$

Here h is the final pre-activation and a is the activation from the previous layer. We utilize the fact that only the element h_k depends on $w_{k,l}$ and then from the matrix multiplication in equation 2.1 $\frac{\partial h_k}{\partial w_{k,l}} = a_l$.

As is evident from the derivation above many factors in the gradient computation are independent of k and l and only need to be computed once per weight update. For weights in other layers the chain rule is simply repeated for the subsequent activations until all weight gradients have been computed. For convolutional layers the same weight will appear in more terms but the principle is the same[9].

2.3.2 Gradient Descent

Once the gradients have been computed they can be used to make small updates to the weights and biases of the network in the descent direction. Let t denote the iteration of a weight w. The basic gradient descent update rule is as follows:

$$w^{t+1} = w^t - \lambda \frac{\partial L}{\partial w} \tag{2.10}$$

Here λ is known as the learning rate which controls how large updates are to be made to the weight per iteration. If the learning rate is too large then the weights may be changed too much to find any local minimum. If it is too small then it may take too many updates for the network to converge.

Parameters in deep learning which are not trained through gradient descent but determine some other aspect of how a network is trained or structured are called hyperparameters. The learning rate is an example of a hyperparameter.[9]

2.3.3 Momentum and ADAM

Instead of using the gradient directly to update the weights there are alternative algorithms which can be used. Despite an appropriate learning rate the gradient may sometimes become vanishingly small leading to insignificant updates of some weights.

One way to address this is to use a running average of the gradient instead of the gradient itself. This way the algorithm will keep updating the weights in the direction they were going even if the magnitude of the partial derivative starts to decrease. Inspired by the analogy to physics such a running average is referred to as momentum. [8]

A more involved way to improve learning is to use individual adaptive learning rates for each weight. One such algorithm is adaptive moment estimation (ADAM)[8]. Here ADAM will only be used for comparison, as it introduces many extra trainable parameters per weight and therefore is not feasible if the intent is to train using binary weights. The details of how ADAM is implemented are outside the scope of this thesis.

2.3.4 Stochastic Gradient Descent

Instead of computing the gradient for all training data every update it is more efficient to divide the data into mini-batches, and to perform one update iteration by computing the gradient for a single mini-batch. The resulting gradient is not truly the gradient of the loss function in equation 2.8 and has a stochastic element as it is dependant on what random mini-batch is being used. This is called stochastic gradient descent and will generally lead to significantly faster convergence[8]. When sending a mini-batch into the loss function the output is no longer the exact loss as previously defined but it can instead be referred to as the batch loss.

The mini-batch size used here is 128 images.

2.4 Input Data Pre-processing

It is common practice to normalize input data so that each pixel has the variance 1 and expected value 0. This is useful because the importance of a pixel may not be equal to the magnitude of its variance between different classes[9]. In this project all pixel values are normalized to within the interval [-1,1].

Chapter 3
Binarized Neural Networks

As mentioned in the introduction binarized neural networks, or BNNs, are deep neural networks which restrict the weights of the network to two values, typically +1 and -1. There exists a whole subset of BNNs where this is the only restriction [10] however most BNN implementations have in addition to this also restricted the activation functions to +1 and -1. The advantage of this in traditional hardware is that multiplications with activations and weights can be performed with bit-wise operations. Additionally, after multiplication accumulating the terms can also be done with specialized bit counting instructions[2]. Binarizing the weights and activations in this way poses a problem for how to train the network as neither the functions or variables are continuous. This has to be addressed if one wants to be able to train a BNN.

3.1 Gradient descent in BNNs

Much prior work in training BNNs, notably Courbariaux et. al. [2] rely on some variation of representing the binary weights with continuous counterparts during training[10]. This allows the continuous weights to accumulate small changes over time which can then be used to determine the optimal binary state of the weights. The sign of any continuous weight, or pre-activation determines if the corresponding binary weight/activation should be set to +1 or -1.

To make gradient descent possible the loss function (and by extension the activation function) need to be differentiable. The signum function:

$$\sigma(x) = \{ -1, \mathrm{x} < 0 + 1, \mathrm{x} \geq 0 \tag{3.1}$$

has the slope 0 for all $x \neq 0$, and an undefined slope at $x = 0$. This is a problem for gradient descent since it leads to a completely vanishing gradient. One way to address this is to substitute the slope with a different function during training. In their 2016 paper Courbariaux et al. [2] propose the use of a variation of the straight-through estimator. The straight-through estimator simply means removing the $\sigma'(...)$ factor from the chain rule (or equivalently to define $\sigma'(x) = 1$). Their altered version takes the following form[2]:

$$f'(x) = \{ 0 \ , x < -1 + 1, -1 \leq x < 10, x \geq 1 \tag{3.2}$$

In practice this amounts to applying the straight-through estimator if $-1 < x < 1$ and if the input value x is e.g. $x > 1$ then we recognize that the signum function $\sigma(x) = 1$ and

will remain unchanged even if x increases further. Suppressing the derivative for large $|x|$ in this way has been shown to improve training time significantly [2].

For this project, due to ease of implementation tanh was used to approximate the signum function during gradient computations since tanh shares the quality of vanishing the gradient for large $|x|$.

In addition to using the signum function as an activation function it is also used to binarize the continuous weights during training. It is important that the weights are binarized in this way because the gradient of each layer depends on the activation from the previous layer as was seen in equation 2.9 and the activations may change drastically if the continuous weights are used instead of the binary weights, leading to incorrect gradients.

3.2 Batch normalization

A technique of great significance in the field of BNNs is batch normalization. In their review of over 80 different articles on BNN research Simons and Lee note that batch normalization is considered essential when using binary weights[10]

It works essentially by normalizing the input to each hidden layer of the network, similarly to how pre-processing has been used to great effect on the input to the first layer. [8]

This means that every feature in every layer, that is, each separate pixel of every feature map requires two additional continuous values to be computed: its mean value and variance. Different values are used during training and when the network is in use. During training the mean and variance is computed for each mini-batch and these values are then used as approximations for the real mean and variance of the entire data-set. When the network is in use, stored values for the mean and variance are used which have been computed as a part of training for example as a running average of the values from each mini-batch.[10].

In typical BNN implementations intended to be trained on traditional hardware the introduction of additional trainable parameters is not a problem since the performance gained from reducing weight multiplications to bit-wise operations is so great and the total number of continuous variables has been significantly reduced[2]. For this project, however, batch normalization was not used due to a couple of reasons. One reason was because it would add additional complexity when developing the new training algorithms. But the main reason is that any hardware which could allow the introduction of what amounts to more continuous parameters than there are binary weights could just as well use these values for continuous weights during training which would remove the need for a new training algorithm utilizing the stochastic switching of the weights.

3.3 Bias Weights

It may be possible in some hardware applications to include a trainable bias in the activation function with greater than a single bit in precision, however if the bias is binary like the other weights in a kernel then this raises questions about its usefulness.

As an example, the second layer of the network to be trained in this thesis has 64 kernels of size 2x2x32 = 128 weights per kernel, all set to either +1 or -1. A bias with a byte of precision would be more than enough to have meaningful significance to such a

sum, however a bias with a single bit of precision could hardly make a dent. Informed by this observation an additional simplification was made by omitting bias weights from the network architecture.

3.4 Stochastic Switching

The central mechanic examined in this thesis is stochastic switching. This is a property found in some memory elements (memristors) such as the Ferroelectric Tunnel Junction [5] and Spin-Transfer Torque Magnetic Memory [4], where the probability of switching between their two unique resistance states is a function of the voltage and length of an electric pulse sent through the memristor[5].

As a constraint the exact switching kinetics of the memristors will not be considered and it is assumed that any calculated switching probability is usable for the training algorithm.

The potential usefulness of stochastic switching for implementing a training algorithm is that it allows control over what percentage of weights will be updated. The hypothesis is that if the gradient is very large for many weights in the network, then switching them all at the same time could make training unstable, but if all such weights are given a certain switching probability lower than 1 then this can be used to make a smaller update to the network and this may lead to more stable training.

Chapter 4
Method and Results

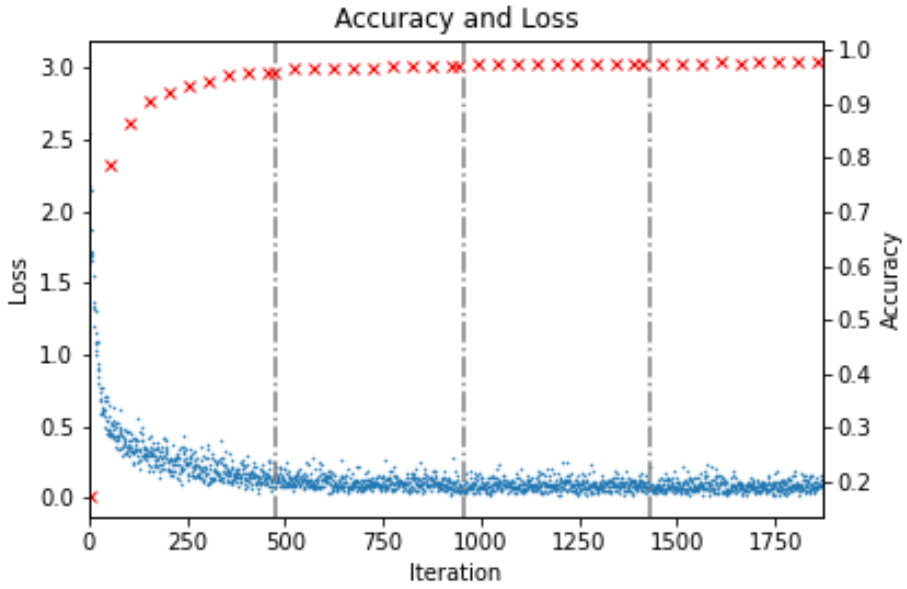

Figure 4.1: Network I. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network I. The vertical lines mark the start of a new epoch.

All networks in this work are evaluated on the MNIST numbers data set, and have the same architecture. The first layer consists of 32 3x3 kernels, followed by 2x2 max-pooling with stride 2. The second layer consists of 64 2x2(x32) kernels, followed again by 2x2 max-pooling with stride 2. Finally the network is flattened with one fully connected layer to the output. The total number of trainable parameters is 288 weights in layer 1, 8192 weights in layer 2 and 23 040 weights in layer 3. This is a moderate size for a network.

Furthermore the networks are implemented in Python 3 using Tensorflow 2. The MNIST data set consists of 60000 training images and 10000 validation images of hand-drawn numbers. One epoch consists of iterating over the 60000 training images where each iteration uses a mini-batch of 128 images. The validation accuracy is computed once every 50th iteration over all 10000 validation images by counting how often the

network's best prediction is correct. The network never makes any updates based on the validation images so to classify them well it must detect general patterns in the training images.

4.1 CNN with Continuous Weights. Network I

The classic CNN, Network I was trained using the ADAM algorithm. Tanh was chosen as activation function in place of the ReLU function since it is symmetrical and more closely comparable to the sign-function used in the BNNs.

The results of the training are displayed in figure 4.1. Both the accuracy and the batch loss can be seen rapidly converging within the first epoch. The accuracy converges to within 98%.

This is the base-line performance with which to compare all the other networks.

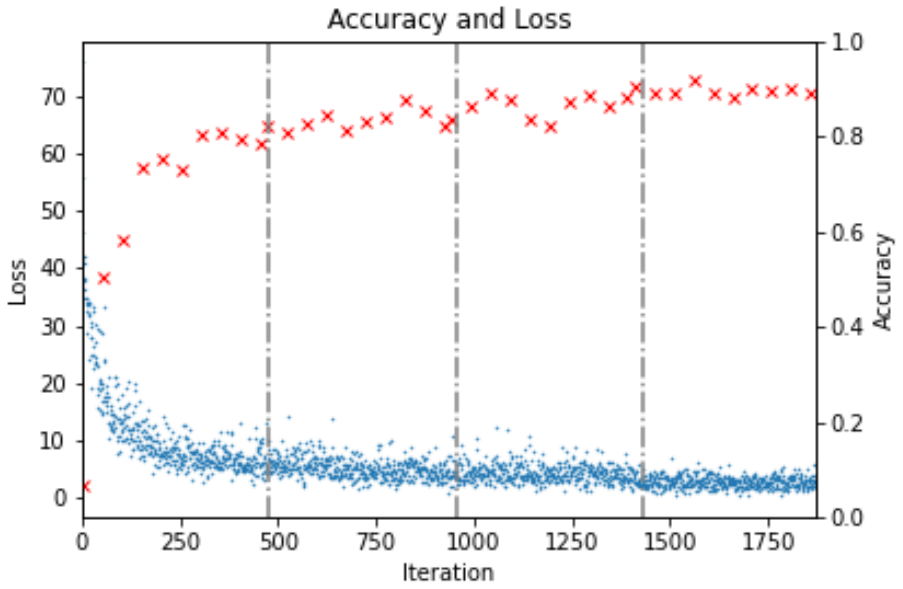

Figure 4.2: Network II. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network II. The vertical lines mark the start of a new epoch.

4.2 BNN with Continuous Weights During Training. Network II

Network II is a BNN which was trained with conventional gradient descent using hidden continuous weights. The architecture was changed from Network I in the following way: Before each convolutional layer, and in the final flattened, fully connected layer the continuous weight values were passed through the binarization function. In addition

the weights were locked inside the interval [-1,1] to prevent weights from accumulating unnecessarily high values which would not affect their binary state, but may make the network slower to update weights which have retained the same binary value for a long time. In addition the activation function was also replaced with the binarization function.

The batch loss and accuracy of Network II can be seen in figure 4.2. Though it cannot be seen in the figure, allowing the algorithm to run for 10 epochs or longer with appropriately declining learning rates, this algorithm has been made to reach an accuracy of 93%. In figure 4.3 one can see the amount of weight switches between -1 and 1 over time. Each layer has its own curve however the results for layer 2 and layer 3 happen to be superimposed in this graph. To allow better comparisons between the layers the number of changes is expressed as a fraction of the total number of weights in that layer.

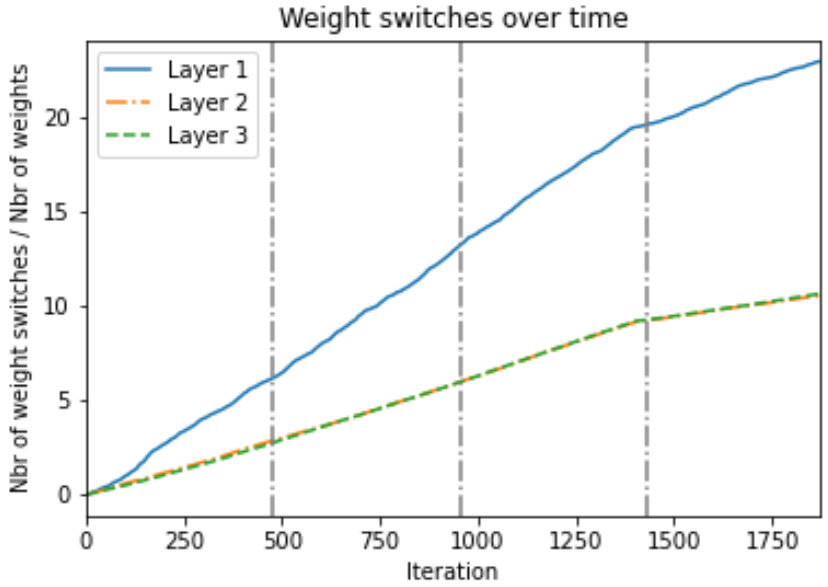

Figure 4.3: Network II. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

4.3 Deterministic Switching. Network III

This is the first network in this paper which utilizes no hidden continuous variables for the weights. All weights are binary and training is done directly on the binary data. The algorithm still relies on backpropagation and uses the gradient to decide which weights to update. For Network III the algorithm was kept fully deterministic and the stochastic switching property is not used. The idea is that a network can be improved by switching the weights with the largest partial derivatives. Algorithm 1 shows how the indices for these derivatives are found.

Net.	Param.	Epoch 1			Epoch 2			Epoch 3			Epoch 4		
IIIa	$\sigma =$	0.8	0.8	0.8	0.95	0.95	0.95	0.95	0.95	0.95	0.99	0.99	0.99
IIIb	$\sigma =$	NaN	0.8	0.8	NaN	0.95	0.95	NaN	0.95	0.95	NaN	0.99	0.99
IV	$\sigma =$	0.8	0.8	0.8	0.95	0.95	0.95	0.95	0.95	0.95	0.99	0.99	0.99
2*V	$\sigma =$	0.8	0.8	0.8	0.95	0.95	0.95	0.95	0.95	0.95	0.99	0.99	0.99
	$\lambda =$	0.1	0.9	0.9	0.01	0.9	0.9	0.003	0.1	0.9	0.0001	0.01	0.1
2*VI	$\sigma =$	0.6	0.7	0.7	0.6	0.85	0.85	0.7	0.95	0.98	0.4	0.9	0.95
	$\lambda =$	0.1	0.8	1.0	0.03	0.5	0.9	0.03	0.5	1.0	0.001	0.03	0.5
2*VII	$\sigma =$	0.6	0.7	0.9	0.6	0.85	0.9	0.7	0.95	0.95	0.95	0.95	0.95
	$\lambda =$	0.1	0.1	0.1	0.03	0.1	0.1	0.03	0.5	0.5	0.01	0.03	0.5
2*VIII	$\sigma =$	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95
	$\lambda =$	0.03	0.03	0.03	0.01	0.01	0.01	0.001	0.001	0.01	0.0001	0.0001	0.01

Table 4.1: This table contains all of the values used for the σ and λ hyperparameters when training Networks IIIa through VIII. For each epoch the values for layer 1 to layer 3 are listed from left to right.

Algorithm 1: Selects indices for weights to switch, based on gradient information.

Data: The weights $\mathbf{W_k}$ and their corresponding gradient $\mathbf{g_k}$ for the current network layer. A cut-off parameter $0 \leq \sigma \leq 1$.

Result: The indices $\mathbf{inds}$ of the weights to be switched.

$\mathbf{p} \leftarrow$ multiplyElementWise($\mathbf{W_k}$,$\mathbf{g_k}$)

$\rho \leftarrow \max(\mathbf{p})$

$\mathbf{inds} \leftarrow \mathbf{p}[\mathbf{p} \geq \sigma \times \rho]$

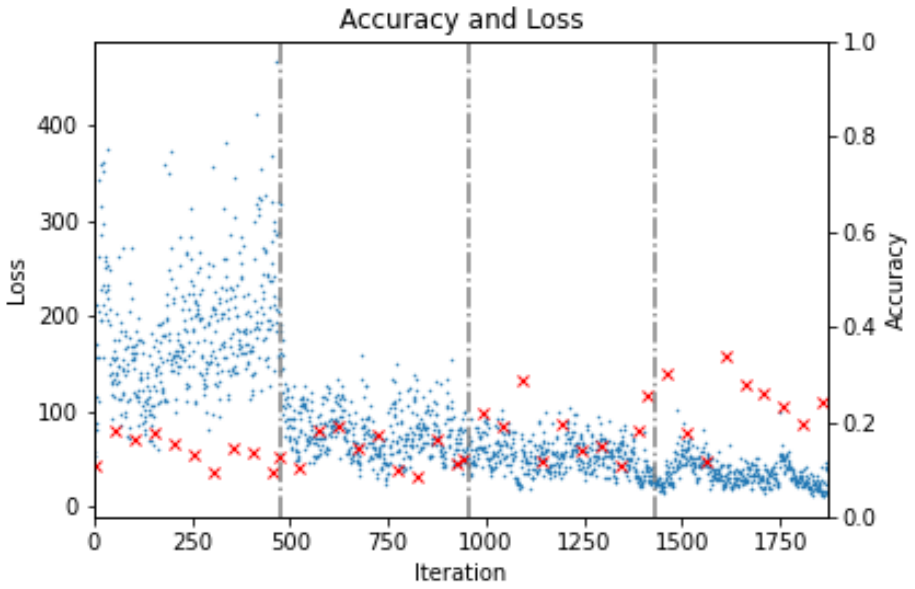

Figure 4.4: Network IIIa. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network III. The vertical lines mark the start of a new epoch.

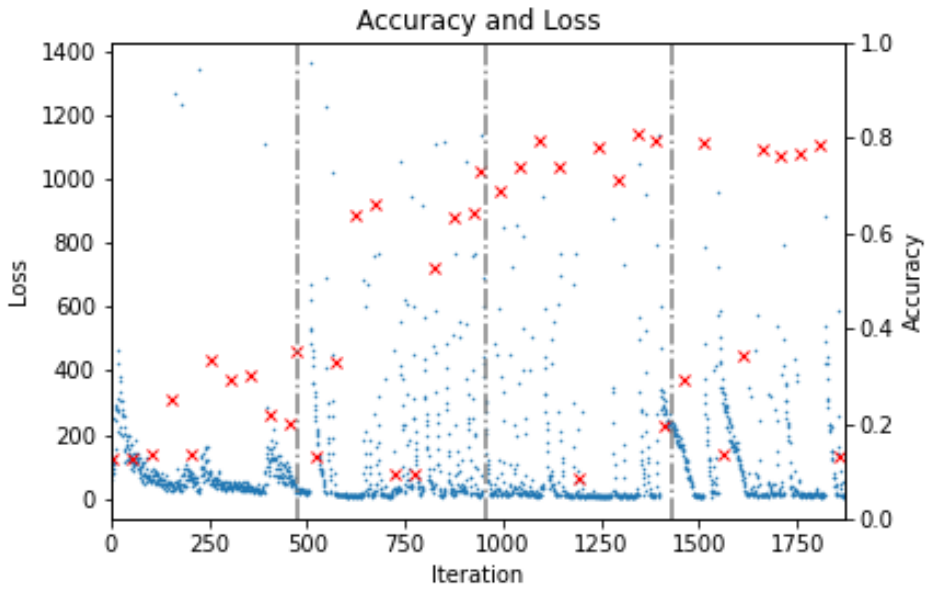

Figure 4.5: Network IIIb. with layer 1 weights locked. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network III. The vertical lines mark the start of a new epoch.

The first step of the algorithm is to compare the sign of each weight with its partial derivative. If they have opposite signs then the weight cannot be incremented further in the descent direction and should instead be ignored. In the algorithm a simple trick is used to perform this selection. By multiplying the weights and gradient element wise the result $\mathbf{p}$ will contain positive values only for weights which can be updated in the descent direction.

The next step is to determine the maximum element of $\mathbf{p}$. For a weight to update the corresponding element of $\mathbf{p}$ must be greater than some fraction $0 < \sigma < 1$ of this maximum ρ.

Network III was trained by repeating algorithm 1 for each layer and switching the sign for the selected weight indices.

Two versions of this network were trained for comparison. Figure 4.4 shows the loss and accuracy results of Network IIIa, which applies the algorithm for all layers. Network IIIb was trained using the same hyperparameter values for σ as Network IIIa, which are all listed in table 4.1, however here no updates are made to the first layer. Instead the weights in the first layer are kept on their random initialization.

Only the weight changes for Network IIIa are included but can be viewed in figure 4.6.

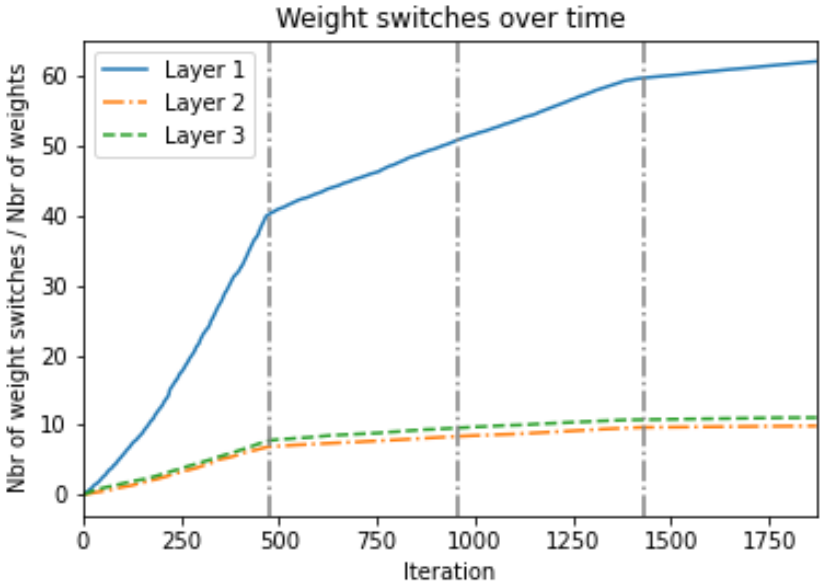

Figure 4.6: Network IIIa. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

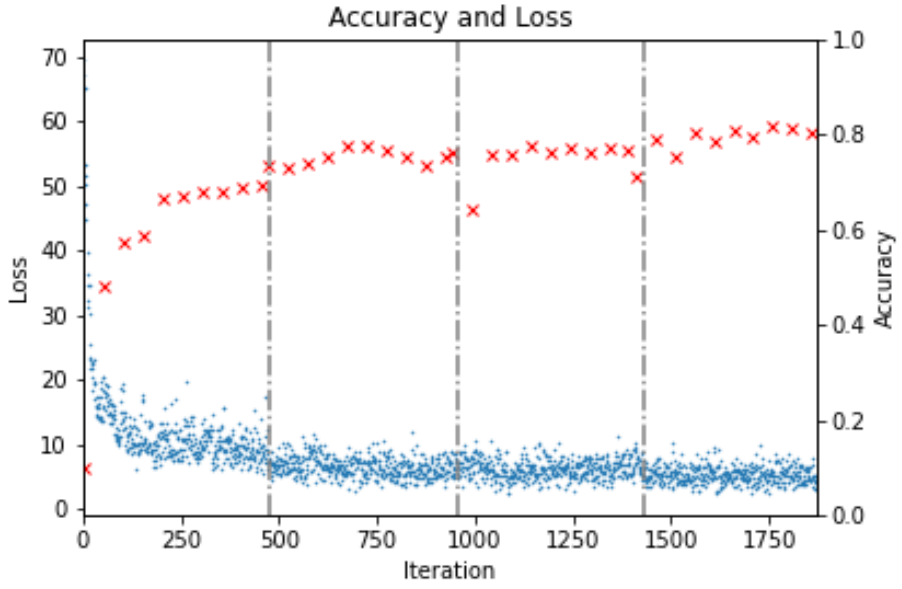

Figure 4.7: Network IV. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network IV. The vertical lines mark the start of a new epoch.

4.4 Deterministic Switching - with Undo Function. Network IV

For Network IV an additional step was added to the training algorithm in the form of an undo-function.

The undo-function consists of a simple check: If the loss on the current batch is increased after updating the weights, then the network is assumed to have made a bad update and the weights are reverted to their previous state. This is applied separately to each layer which minimizes the amount of progress being lost.

As can be seen in table 4.1 the hyperparameters are the same as for the previous network.

The accuracy and loss for this network are displayed in figure 4.7 and the weight changes are displayed in figure 4.8.

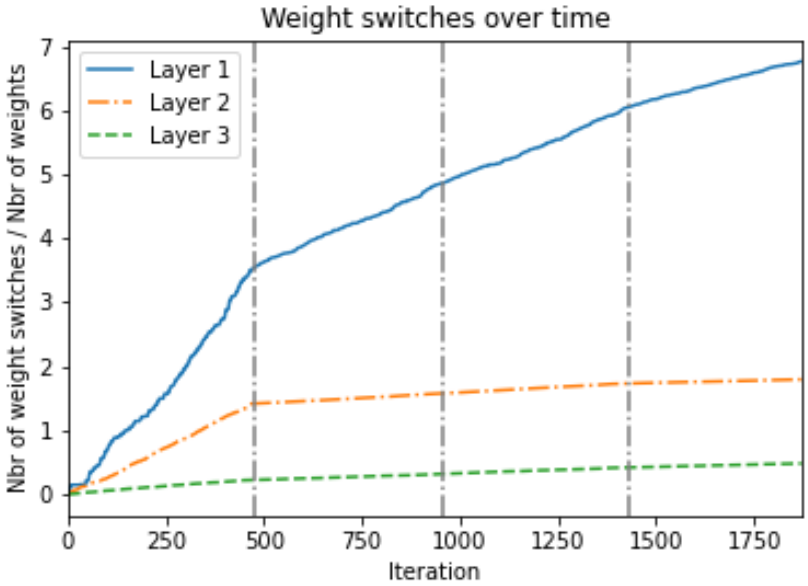

Figure 4.8: Network IV. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

4.5 Stochastic Switching - with Undo Function. Network V

To switch weights stochastically one must first compute a probability of switching for each weight. One possible solution is to simply take the indices generated in algorithm 1 and give them all the same switching probability. The solution proposed here goes one step further and alters algorithm 1 so that the switching probability is tied to the magnitude of the gradient.

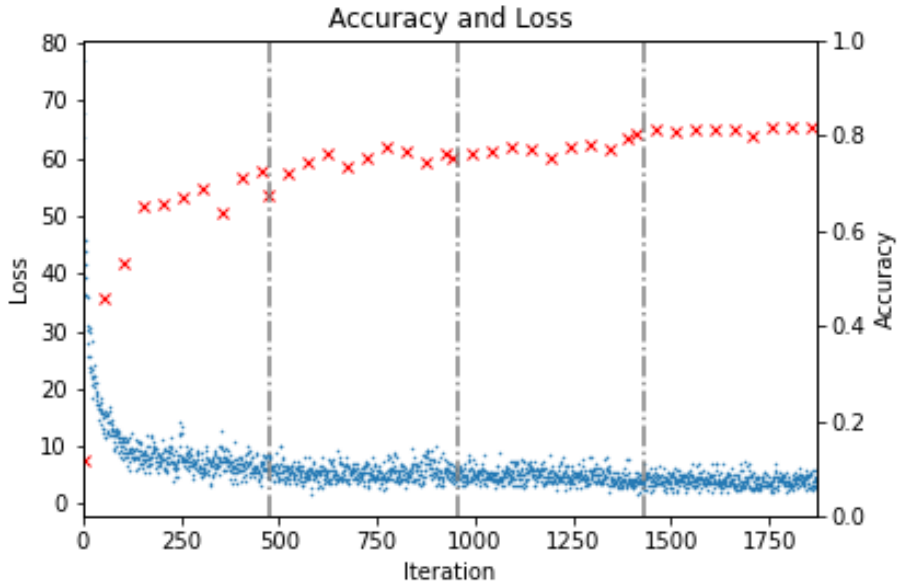

Figure 4.9: Network V. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network V. The vertical lines mark the start of a new epoch.

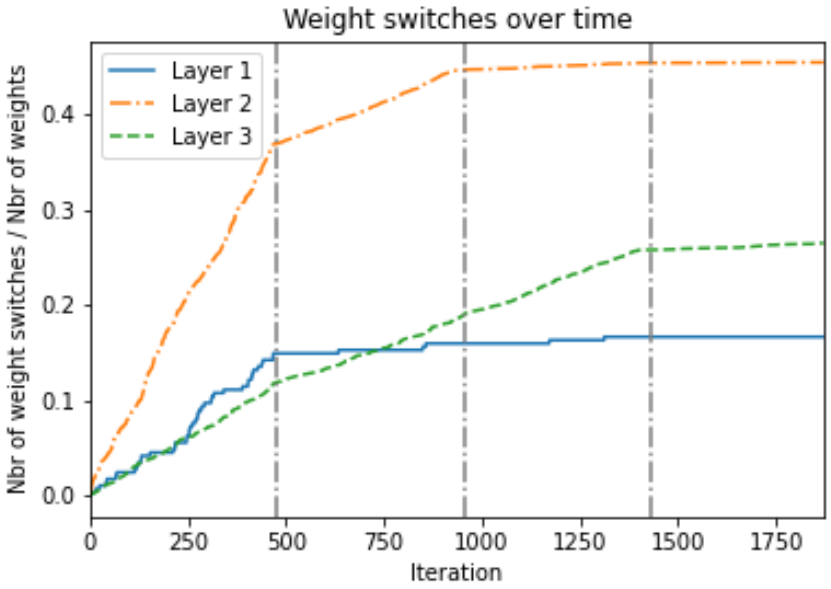

Figure 4.10: Network V. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

In algorithm 2 this probability is derived with two hyperparameters, σ and λ. σ serves the same role as in algorithm 1. For a weight to be eligible to update it must have a partial derivative of minimum size $\sigma \times \rho$. As the elements of $\mathbf{p}$ vary linearly from $\sigma \times \rho$ to ρ the corresponding switch probability varies linearly from 0 to λ.

Using this algorithm to generate switching probabilities Network V was trained, and the results can be seen in figures 4.9 and 4.10.

Algorithm 2: Selects indices for weights to switch, based on gradient information.

Data: The weights $\mathbf{W_k}$ and their corresponding gradient $\mathbf{g_k}$ for the current network layer. A cut-off parameter $0 \leq \sigma \leq 1$. A learning rate λ

Result: The probabilities $\mathbf{p}$ of the weights to be switched.

$\mathbf{p} \leftarrow$ multiplyElementWise($\mathbf{W_k}$,$\mathbf{g_k}$)

$\rho \leftarrow \max(\mathbf{p})$

$\mathbf{p} \leftarrow \lambda \frac{\mathbf{p}/\rho - \sigma}{1-\sigma}$

$\mathbf{p} \leftarrow clip(\mathbf{p}, 0, 1)$

4.6 Momentum-Based Stochastic Switching - with Undo Function. Network VI

This implementation is almost identical to the previous implementation with the exception that all gradient information used in the algorithms has been replaced with momentum updated according to 95% of the previous iteration momentum + 5% of the gradient. The results for network VI can be viewed in figures 4.11 and 4.12.

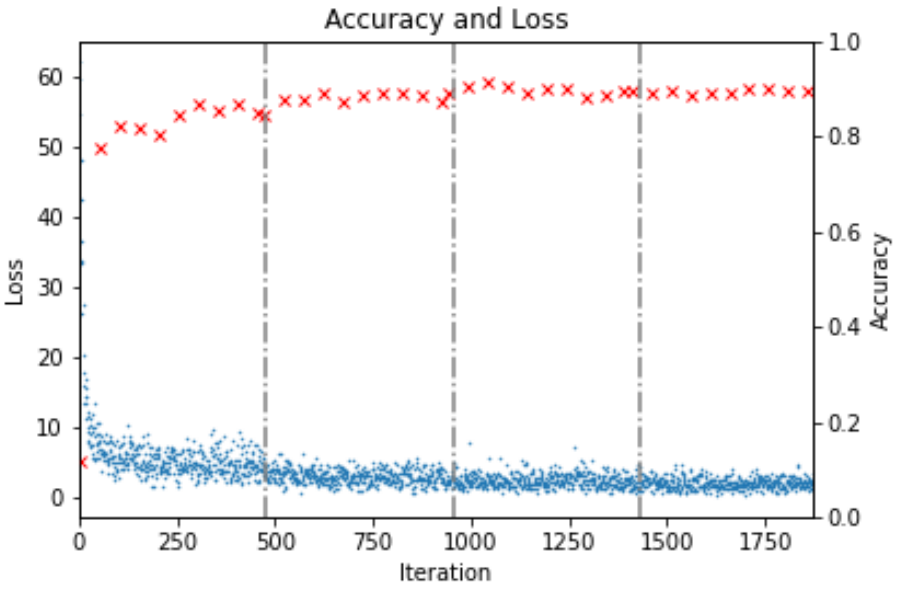

Figure 4.11: Network VI. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network VI. The vertical lines mark the start of a new epoch.

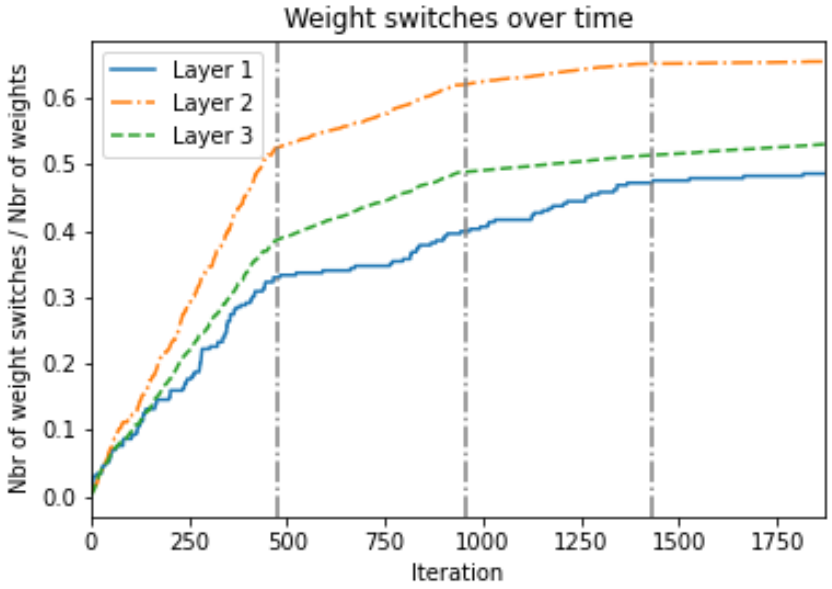

Figure 4.12: Network VI. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

4.7 Integer Accumulation-Based Stochastic Switching - with Undo Function. Network VII

This algorithm uses a different approach to storing information between iterations which requires less than a byte of extra storage for each weight. After computing the batch gradient only the sign of each partial derivative is used to increment or decrement an integer accumulation variable. This accumulation variable then replaces the partial derivative in algorithm 2. This also introduces a new hyperparameter in the form of the accumulation cut-off. For example if the cut-off is 20 then the accumulation variable cannot be increased beyond 20 or decreased beyond -20. The size of the cut-off determines how many bits of storage are required and for Network VII it was set to 50.

The results for network VII can be viewed in figures 4.13 and 4.14.

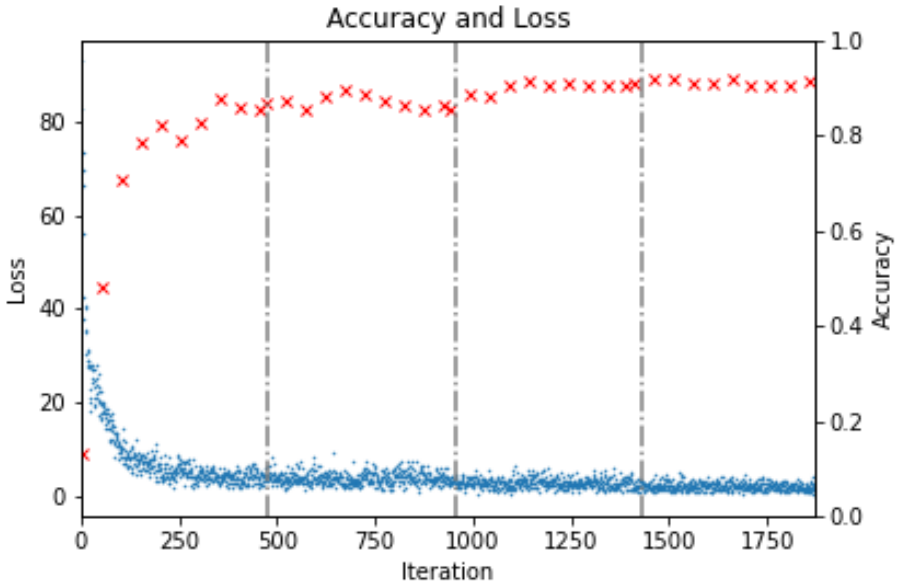

Figure 4.13: Network VII. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network VII. The vertical lines mark the start of a new epoch.

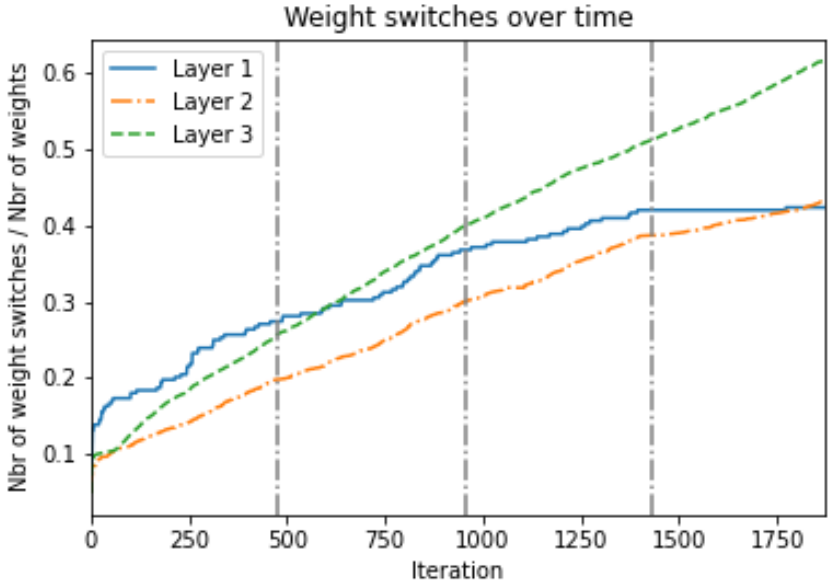

Figure 4.14: Network VII. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

4.8 Low Resolution Gradient Sign Accumulation Stochastic Switching - with Undo Function. Network VIII

This evolution of the previous algorithm addresses the gradient computation stage. Instead of computing the gradient for an entire batch of 128 images this version handles each image individually and accumulates the sign of 128 gradients before updating the weights. Additionally, to allow for some form of weeding out of insignificant gradients the algorithm introduces a new hyperparameter $0 \leq \sigma' \leq 1$, to ensure that we only count partial derivatives with a sufficiently large magnitude compared to the greatest partial derivative. This σ' replaces σ in algorithm 1 which is used to determine which partial derivatives to use and which ones to ignore for the current image. Then the accumulated integer values are sent to algorithm 2 where σ is used to determine which weights to switch.

For Network VIII the new hyperparameter was set to $\sigma' = 0.9$ for all layers and the accumulation cut-off was set to 200.

The results for network VIII can be viewed in figures 4.15 and 4.16.

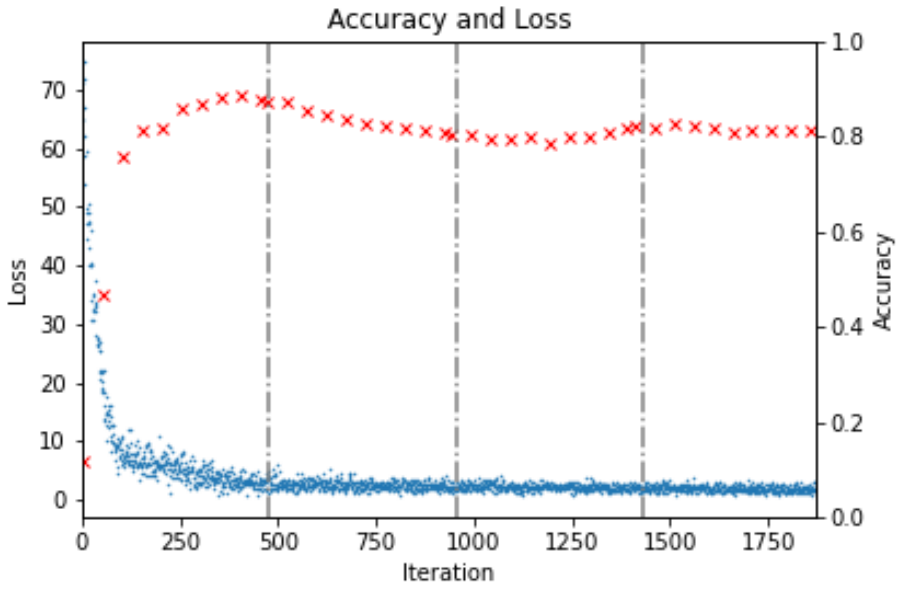

Figure 4.15: Network VIII. The blue points mark the batch loss and the red crosses mark the validation accuracy for Network VIII. The vertical lines mark the start of a new epoch.

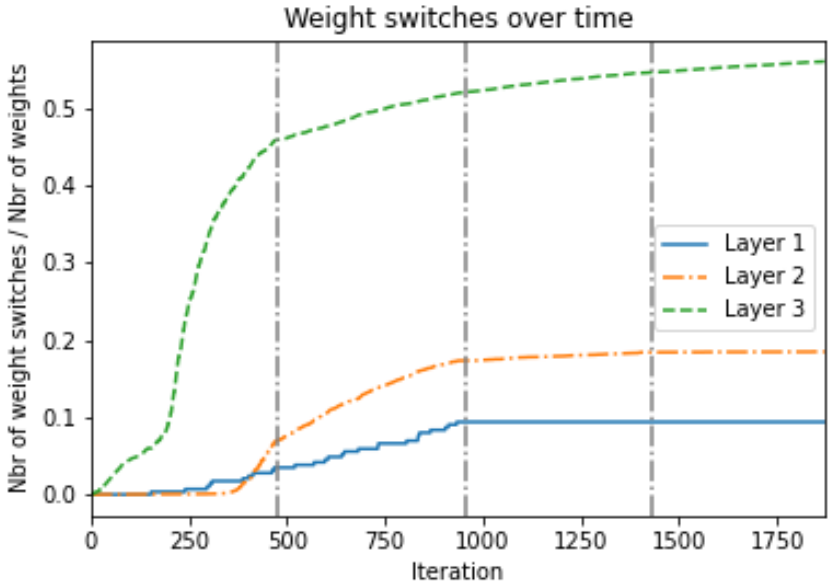

Figure 4.16: Network VIII. The number of changed weights in each layer over time, as a fraction of the total number of weights in each layer.

Chapter 5
Discussion

5.1 Comparing the algorithms

5.1.1 Best and Worst BNN Performance

It turned out to be very difficult to achieve comparable results to the continuous CNN (Network I) from a BNN which does not use batch normalization even when training with conventional gradient descent (Network II). The disparity in the results can be seen both in the accuracy and the batch loss for the corresponding networks. The BNN trained using continuous weights for accumulation reached 90% accuracy in 4 epochs... and as noted it has the ability to reach 93% if given more training time. Even though the variables are continuous the objective function is not which could explain why the accuracy oscillates more for Network II instead of converging to a single value in figure 4.2.

Network IIIa displays the worst convergence in accuracy and batch loss of all the networks and when comparing the weight changes (in figure 4.6) for this network with any other implementation it is clear that this network updates the weights by far the most often. The most likely explanation is that algorithm 1 is built in such a way that some weight is absolutely required to be switched in each iteration. But there may be many times when layer 1 shouldn't update any weights and the network is most sensitive to changes in the first layer since the optimal states of subsequent weights are affected.

5.1.2 Increasing Training Stability

To address this flaw we have seen two possible approaches. The first approach seen in Network IIIb addresses the problem by not updating the first layer and simply leaving it in a randomly initialized state. The resulting performance, while not completely stable, at the very least shows that such a network can reach an accuracy above 80%. The remaining instability can be explained by applying the same logic to layer 2; sometimes the algorithm forces updates when none are good. It would most likely not be useful to lock the weights in layer 2 as the possible weight constellations in this layer are much more complex. It may be feasible to use pre-programmed filters in layer 1.

One idea might be to attempt to change the weight update algorithm so that it doesn't always attempt bad weight updates. The problem is that the only available information is gradient information and the algorithm attempts to make large changes to the weights

based on this local information. There is no way of securing that the gradient will be a good indicator that a weight is better off in the opposite state.

The second proposed solution seen first for Network IV is to introduce the undo-function. This allows more bad inferences to be made from the gradient without the same consequences. As can be seen in figure 4.7 the resulting accuracy converges to above 80% in a much more stable manner and the number of weight updates is significantly reduced. Whereas Network IIIa (figure 4.6) updates the weights a minimum of 10 times the number of weights in a layer (which equates to over 200 000 changes for layer 3) Network IV doesn't even update all weights in layer 3 (figure 4.8) and a good algorithm shouldn't need to update all weights since 50% of weights should be initialized to their optimal state.

5.1.3 Stochastic Switching Algorithms

Network V introduces stochastic updates to the weights which allows for the reintroduction of learning rates. The ability to decrease the learning rate for later epochs allows for more fine tuning where fewer weights are updated at a time, leading to a smoother convergence. It also puts less reliance on the undo-function as e.g. layer 1 can be given a low learning rate which better reflects how often changes to this layer may be beneficial. Despite this even the best attempt couldn't produce an accuracy above 83% and it is difficult to motivate why stochastic switching would be beneficial as this level when comparing this network to the deterministic Network IV.

5.1.4 Reintroducing Accumulation Variables

It is a great weakness for Networks III through V that they must perform large changes to the objective function based on local gradient information. For example if the ideal value for a weight in fact is 0 then it may have a great slope for both of the binary states -1 and +1. Switching between these two values may serve little practical use for the network but will be prioritized by any algorithm which only regards the size of the partial derivative. To explore the extent of this effect the proceeding networks reintroduced accumulation of gradients.

Notably Network VI is the first network to get above 90% accuracy since the classically trained BNN (Network II). This is to be expected considering the similar quality of information available during training. The running average for the gradients serve as a form of accumulation of similar precision as using continuous weights during training. As such it has not utilized stochastic switching to any great effect in replacing any part of the conventional training method.

Network VII is a step further in the direction of substituting expensive calculations with cheaper ones, facilitated by stochastic switching. Here much information is thrown away pertaining to the gradient without significant loss of performance. The results for Network VII seen in figure 4.13 were facilitated by the stochastic switching property as this allowed fewer weights to be updated at a time despite the lower resolution in information, however it is unclear if a deterministic approach could yield similar results since the clear source of the improved performance comes from being able to accumulate gradient information. The main issue still is that the gradient must be calculated for an entire batch, accumulating results for 128 images before the information is simplified.

Finally Network VIII addresses this by handling each image individually. It was crucial to include the additional hyperparameter σ' for this algorithm to work at all as it makes sure that small gradient noise can be filtered out before accumulation. The resulting algorithm does not reach the same accuracy as that of network VII. The accuracy peaks early during training at around 85% after half an epoch and then decreases over time. Notably the maximum accuracy and minimum loss do not coincide, and the network actually performs better in the first epoch than in later epochs.

5.2 Mapping Algorithms to Architecture

5.2.1 Finding the Maximum Partial Derivative

One of the most problematic steps in algorithms 1 and 2 is the requirement to compute the maximum element ρ of the gradient. To do this accurately one would be required to iterate over all elements of the current gradient which may present many problems for a large network.

It is clear that, though this value is simple to comprehend and to write in psuedo-code a practical implementation would have to replace it with some approximation. Knowledge of the typical magnitude of the gradient may be sufficient to replace ρ with a constant value. For implementations which accumulate gradient signs such as Network VII and VIII there is a maximum accumulation point which could potentially be used in place of ρ with similar results.

5.2.2 Hyperparameters

The algorithms presented require the implementation of a couple of hyperparameters. Out of these hyperparameters the learning rate λ may be the most straight forward as it directly corresponds to the switching probability of the memory elements used to implement the binary stochastic weights.

The two cut-off parameters σ and σ' are less obvious to implement as they imply comparisons between different floating point values.

The integer accumulation variables used in Network VII and VIII would have to be implemented using a byte of conventional memory in the peripheral circuit. Since this is equivalent to storing a byte worth of resolution for each weight it puts the feasibility of these Networks into question. Why not simply use this byte to train the network using traditional gradient descent if it is available?

5.2.3 The Undo Function

As has been shown the Undo Function has a crucial role for stability during training.

One possible way to implement this in hardware could be to group memristors in pairs where one memristor is used for saving the previous state of the weight before an update. The undo could then be performed by writing back from this previous state.

One important detail to note here is that this implementation could double the number of writes done to every memory element. If the training of a network requires frequent use

of the undo-function then the effective number of changes during training may be much higher than the number of accepted updates to a weight.

5.3 Are these BNNs feasible?

If feasible means achieving comparable results to a conventional CNN with continuous weights then none of the BNNs featured in this paper are feasible. Based on the review by Simons and Lee [10] where so much prior research has necessitated batch normalization it is unlikely that comparable results can be achieved without leaving the realm of pure BNNs and introducing many continuous parameters.

There may be applications where a worse accuracy is acceptable if it means that training can be performed on specialized low-power hardware and the ability to handle rudimentary deep learning without significant complications could be enough of a bonus to render results of this nature feasible. Disregarding the complications that may arise when developing such a hardware implementation it still remains to be determined if stochastic switching has been shown to be potentially useful for training the networks.

While Networks V through VIII were inspired by the stochastic switching property and integrates it into the training it is unclear if the algorithms presented could be tweaked in such a way that completely deterministic switching would yield equivalent results. At the very least it has been proven that some approaches like the training of Network VIII which use very low resolution gradient information can be implemented with stochastic switching, despite there being no obvious translation to a deterministic implementation.

The benefit of Network VIII is also that it opens up the door for many simplifications in the gradient computation process as it primarily relies on the sign of each partial derivative, combined with some measure of its magnitude which perhaps need not be very precise at all.

5.4 Further Research

5.4.1 Solving the Accumulation Problem

One possible direction to develop these algorithms is to look into chaining several stochastically switching elements together to implement accumulation. As Courbariaux et al. mention [2] a key reason why high precision accumulation of gradients is so important is because it allows small subtle trends in the gradients to accumulate over time, something which the networks in this project have struggled with. However it might be possible to use stochastically switched bits to solve this problem.

5.4.2 Continuous Activation Functions

It may be possible to improve the performance of the networks by using continuous activation functions. In BNNs implemented on traditional hardware discrete activations are necessary to allow multiplication with weights to be done with bit-wise operations, however in the hardware context of using binary weights with stochastic switching only the state of the weight is necessarily limited to two states.

5.5 Conclusion

This thesis set out to examine the viability of using stochastic switching to train Binarized Neural Networks. The resulting networks do not perform as well as their conventionally trained counterparts without reintroducing some element of accumulation, however this could be an issue of not having found the right algorithm yet. Based on the algorithms which were tested it is clear that BNNs require some form of accumulation of gradients for training to work and stochastic switching shows potential to allow the precision of the accumulation to be relaxed. It is clear that stochastic switching introduces some stability issues during training and the suggested technique to fix this, implemented here with great effect, is to use an undo function which can check if a weight update impairs the network and which reverts the network to its previous state when needed.

Anyone attempting a hardware implementation would still have to consider the requirement for accumulation and therefore it is of great interest to attempt to implement accumulation with stochastic switching.

References

[1] Sevilla, J. et al., "Compute Trends Across Three Eras of Machine Learning", arXiv **2022** arXiv:2202.05924

[2] Courbariaux, M.; Bengio, Y. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1 .arXiv **2016**, arXiv:1602.02830.

[3] Roy, K., Jaiswal, A. Panda, P. Towards spike-based machine intelligence with neuromorphic computing. Nature 575, 607–617 **2019** . https://doi.org/10.1038/s41586-019-1677-2

[4] A. F. Vincent et al., "Spin-Transfer Torque Magnetic Memory as a Stochastic Memristive Synapse for Neuromorphic Systems," IEEE Trans. Biomed. Circuits Syst., vol. 9, no. 2, pp. 166–174, Apr. **2015**, doi: 10.1109/TBCAS.2015.2414423.

[5] H. Mulaosmanovic et al., "Switching Kinetics in Nanoscale Hafnium Oxide Based Ferroelectric Field- Effect Transistors," ACS Appl. Mater. Interfaces, vol. 9, no. 4, pp. 3792–3798, Feb. **2017**, doi: 10.1021/acsami.6b13866

[6] Garcia, V. and Bibes, M. Ferroelectric tunnel junctions for information storage and processing. *Nat. Commun.* 5:4289 doi: 10.1038/ncomms5289 **2014**.

[7] Russel, S.; Norvig, P. *Artificial Intelligence: A Modern Approach*, Third Edition, Pearson Education Limited **2010** pp.726-736, p. 938

[8] A. Shrestha and A. Mahmood, "Review of Deep Learning Algorithms and Architectures," in IEEE Access, vol. 7, pp. 53040-53065, **2019**, doi: 10.1109/ACCESS.2019.2912200.

[9] Bishop, C. M.; *Pattern Recognition and Machine Learning*, Springer Science + Business Media **2009** p. 2, p. 235, pp. 267-269

[10] Simons, T.; Lee, D. A Review of Binarized Neural Networks, Brigham Young University, **2019**