Building a Deep Neural Network From Scratch

Fredrik Sundström and Samppa Raittila

***Abstract*—Machine learning is becoming increasingly common in our society and is predicted to have a major impact in the future. Therefore, it would be both interesting and valuable to have a deep understanding of one of the most used algorithms in machine learning, deep neural network. In this project, the goal is to implement a neural network from scratch and use it to solve an image recognition problem. Specifically, instead of using common machine learning libraries, such as Pytorch or TensorFlow, we only use a linear algebra library (numpy) and implement the deep learning models and algorithms ourselves. In addition to a fundamental implementation of a neural network we implement performance enhancing algorithms and study the effects these have on our neural network.**

The finished program is a neural network that can be customized to any size and can implement different activation functions. It also has several options for different training features, called hyperparameters, such as batch-size, learning rate and momentum which can be tweaked to optimize the training process. Furthermore, we train neural networks to solve two different image classification problems each with a different complexity. On the easier problem MNIST, our neural network correctly classifies handwritten digits with a test accuracy of 95.75% . On the more complex dataset CIFAR10, the neural network achieved an test accuracy of 41.94%.

***Sammanfattning*—Maskininlärning blir alltmer vanligt i vårt samhälle och förväntas att ha en omfattande samhällspåverkan. Skulle det då inte vara intressant att ha en djup förståelse av en av de mest använda maskininlärnings algoritmerna, deep neural networks? I det här projektet är målet att implementera ett neuralt nätverk från grunden i Python och använda det för att lösa ett bildigenkänningsproblem. Med en implementation från grunden menar vi, i stället för att använda vanliga maskininlärningsbibliotek så som Pytorch eller TensorFlow, kommer vi att använda oss av ett bibliotek innehållande linjär algebra funktionalitet. Utöver att implementera ett neuralt nätverk från grunden kommer vi även att implementera några prestationsförbättrade algoritmer och studera deras effekt på vårt neurala nätverk.**

Det färdiga programmet är ett neuralt nätverk som kan anpassas till önskad storlek samt använda sig av olika aktiverings funktioner. Programmet har även flera alternativ för att optimera träningsprocessen genom att justera så kallade hyperparametrar, så som batch-size, learning rate och momentum. Vidare så användes det neurala nätverket för att lösa två olika bildigenkänningsproblem med olika svårighetsgrad. På det enklare problemet MNIST, klassificerade vårt neurala nätverk rätt i 95,75% av fallen. Och på det svårare datasetet CIFAR10, klassificerade det neurala nätverk rätt i 41,94% av fallen.

***Index Terms*—Machine Learning, Artificial Neural Network, Supervised Machine Learning, MNIST, CIFAR10, Backpropagation, Momentum, Mini-batch.**

Supervisors: Henrik Hellström

TRITA number: TRITA-EECS-EX-2023:178

I. Introduction

Machine learning (ML) can be described as the field of computer science where the goal is to create software that can perform on tasks that they have not been specifically programmed to do. Central features of these types of programs are that they can recognize patterns, learn from data, and adapt to new situation with little interference from humans.

ML has a wide variety of applications. Most people today have experienced some of them, such as email filtering, speech recognition, and algorithms aimed at content and ad recommendations. In the future, we can expect ML to have a massive societal impact. According to Ng [1], the change will be on the scale similar to how the electrification changed the society a 100 years ago. Some of the fields we can expect to have a significant change are healthcare, agriculture, retail, transportation, and finance [2].

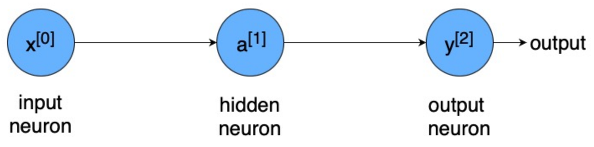

Fig. 1. The architecture of a simple neural network

One of the most successful ML methods is neural networks (NN) which is inspired by how brains work [3]. The simplest NN can be thought of as three nodes connected in a chain according to Figure 1. The node to the left is called the input node, the node to the right the output node, and the middle node is called the hidden node. All the nodes are connected via a parameter called weight. This weight will amplify or attenuate the data that is transmitted through it. Also, the hidden and output node have an "activation function". This function defines the output of the node according to the function's mathematical characteristics. Furthermore, to make the neural network perform as desired it must be "trained". During the training process the weights of the neural network change.

The development of neural networks can be traced back to as far as 1943, and over the years several scientist and mathematicians have contributed to the field. At 1957 Frank Rosenblatt built the first neural network called the perceptron [3]. Worth mentioning is that Rosenblatt´s primary goal with the perceptron was to gain a better understanding of how the human brain worked, rather than to make advancement in ML. Later in 1974 the backpropagation algorithm was introduced [3]; the algorithm is used to train neural networks and is a very popular method today. During the 1980 hidden neurons were introduced and gave neural networks the ability solve more complex problems [3]. Neural networks with more than one hidden layer are called deep neural networks (DNN). Furthermore, advancements in computation hardware, specifically GPU technology have contributed to the success of NN, since access to powerful computing power is important when training a NN. Therefore, we saw rapid advancement in the field of ML around 2010 since the GPU technology at the time offered the needed computing power [3].

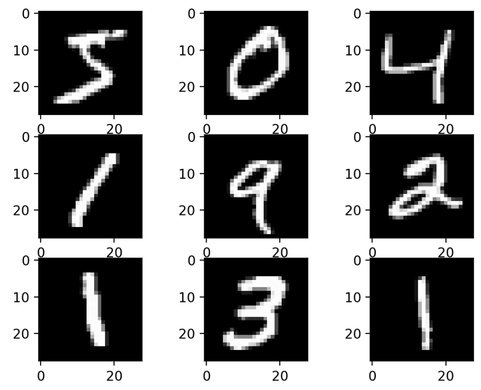
Fig. 2. MNIST dataset

Fig. 3. CIFAR10 dataset

The goal of this project was to implement a DNN from scratch in Python only using Numpy, a code library containing liner algebra capabilities. The goal for our NN is to be capable of solving classification problems after being trained with supervised learning. Our program will be trained to classify images of handwritten numbers, with a dataset called MNIST, Figure 7 that is a standardized dataset used for comparing performance of different NNs. Our aim was to achieve an accuracy of 80% on this dataset. Also, we have train an another NN on the dataset CIFAR10, Figure 3 and compered the results. Furthermore, we have implemented some performance enhancing algorithms and analyzed what effect they have on our NNs.

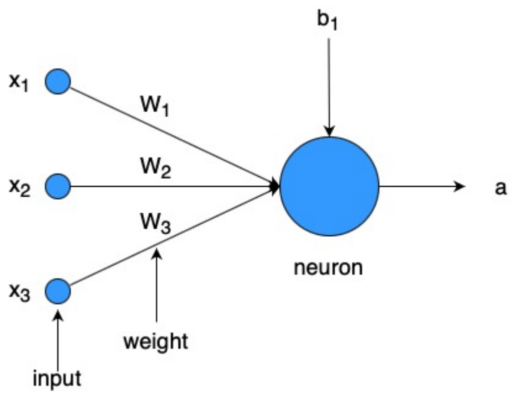

Fig. 4. Schematic of a neuron

II. Theory

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b_1 \tag{1}$$

$$a = \sigma(z) \tag{2}$$

A. Functionality of a single Neuron

A single neuron can be thought of as doing two calculations, pre-activation and activation. First, a sum of the input values is calculated. The input consists of two types of terms. First, a set of values calculated from the inputs x by multiplying with the weights, w which are connected according to Figure 4. The second type of term is the bias-term, b. All these terms are summed together according to Equation 1. This sum z is called pre-activation. Pre-activation is passed on as an input to the activation function, see Equation 2. The result of the calculation made by the activation function, a is called the activation and it is the output of the neuron. Activation functions are discussed in more detail in chapter II-D but for now it can be thought of as a function that outputs a value close to one if the input is large and close to zero if the input is small.

B. Neural Network Architecture

As previously mentioned, NNs consists of three types of layers, the input layer, the hidden layers, and the output layer. In fully-connected NNs, the layers are connected via weights structured like the connections in Figure 5. In the input layer, which only consists of one layer of neurons, the data is simply being passed into the model. In the hidden layer, which can consist of several layers of neurons, the neurons process the data that is being fed into them with an activation function and passes the output to the next layer. The activation function can take many different forms but is a mathematical function that provides a non-linearity to the NN [4]. In the output layer, the neurons also have an activation function, but the layer only consists of one layer with one or several neurons. Depending on what problem the NN is intended to solve, the

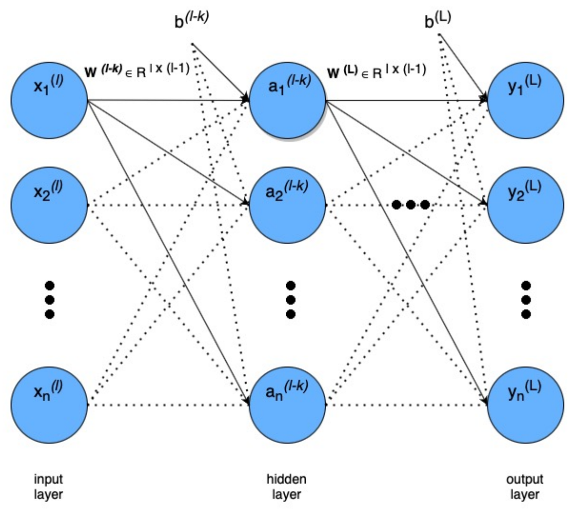

Fig. 5. The architecture of a general neural network

output neurons output will have different meaning. If the NN is attempting to estimate a value, the neuron will output a numerical value. If the NN is intended to solve a classification problem the output neurons will correspond to each possible class and the neurons will output a value between zero and one. The neuron with the highest value corresponds to the NN´s prediction.

C. Forward pass

The process where a neural network calculates an output or a prediction for a given input value is called forward pass or forward propagation [5]. The functionality of a single neuron in the first hidden layer is described in Equations 3 and 4 [5]. Here superscript denotes the layer and subscript the place of the neuron in the layer. First, a linear combination of the input **x** with weights $\mathbf{w}_1^{(1)}$ is calculated and a bias term $b_1^{(1)}$ connected to the neuron is added. The result of this calculation, z is called a pre-activation of the neuron. This pre-activation is given to a activation function σ in Equation 4. The result $a_1^{(1)}$ is called the activation and it is the neurons output. Activations are then used as inputs to the next layer in a similar manner as input **x** [4].

$$z_1^{(1)} = \boldsymbol{w}_1^{(1)}\boldsymbol{x} + b_1^{(1)} \tag{3}$$

$$a_1^{(1)} = \sigma(z_1^{(1)}) \tag{4}$$

Formulas for a single neuron can be generalized for whole layers of neurons as in Equations 5 and 6. Now the pre-activations and activations become column vectors and the whole layer can be calculated simultaneously using matrix multiplications. The functionality of a full layer is summarized in matrix form in Equation 7. Output of the whole network can be calculated iteratively using Equation 7. This is what the forward propagation algorithm does and the result is a prediction **y** as in Figure 5 [5].

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{5}$$

$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{z}^{(l)}) \tag{6}$$

$$\begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_m^{(l)} \end{pmatrix} = \sigma \left[\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix} \begin{pmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_m^{(l-1)} \end{pmatrix} + \begin{pmatrix} \mathrm{b}_1^{(l)} \\ \mathrm{b}_1^{(l)} \\ \vdots \\ \mathrm{b}_n^{(l)} \end{pmatrix} \right] \tag{7}$$

D. The Activation Function

To give a NN the ability to learn powerful operations, a non-linearity has to be introduced to the network, which is done with activation functions such as Logistic Sigmoid, Rectified Linear Unit (ReLU), and Parametric Rectified Linear Unit (PReLU). Without these functions and thus the introduction of the non-linearity, the NN would no longer be able to perform complex task such as image and speech recognition [6].

$$ReLU(x) = max(0, x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases} \tag{8}$$

$$PReLU(x) = \begin{cases} ax \text{ if } x < 0 \\ x \text{ if } x \geq 0 \end{cases} \tag{9}$$

$$Sigmoid = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{10}$$

The different activation functions Logistic Sigmoid, ReLU and PReLU have their pros and cons. ReLU is one of the simplest activation functions and it is a piecewise linear function that outputs zero if the input is negative, otherwise it outputs the input according to Equation 8. One drawback with ReLU is that the derivative of ReLU for negative x is zero and this can cause problem when training the network. The issue is that when the value of the gradient is zero it will prevent the value of the weights to be updated [7]. To avoid this problem a variation of ReLU, PReLU can be used, see Equation 9. With PReLU the gradient is non-zero for all values of x except for $x = 0$ where it is undefined. Here the gradient can be set to either 1 or 0. Another commonly used function is the Logistic Sigmoid function (Equation 10) which outputs between 0 and 1 and it derivative is defined everywhere. However, the sigmoid function is in practice computationally expensive to calculate due to it is an exponential function. Another drawback is that the derivative of the sigmoid function for large absolute values of x becomes very small which causes problems when training the NN similar to how the zero value of the derivative of the ReLU function prevent the weights from being updated [6] [8].

E. Supervised learning and Cost Func

The forward propagation algorithm outputs a value, a prediction, for every input, but in the beginning these predictions are random. The weights and biases of the neural network have to be changed so that the NN implements the desired function [4].

In supervised learning, input data is equipped with the corresponding desired output value [9]. For example if the data includes images of numbers, data needs to have the information of what number the image contains. From now on, it is assumed that data is labeled like this and supervised learning can be used.

To be able to change the parameters of the NN in a correct way, the difference between the actual prediction and the desired output has to be measured [4]. This measurement is called a cost function or a loss function.

The choice of cost functions depends on the problem. An example of a cost function is Cross-Entropy given in Equation 11. Cross-Entropy is commonly used in binary classification problems. Label y is either zero or one and prediction p is a floating value between zero and one [4].

$$J = -(y \log(p) + (1-y)\log(1-p)) \tag{11}$$

F. Gradient Descent

Gradient descent is an algorithm used for finding a local or global minima of a function. In the context of neural networks it is used for minimizing the cost function of a neural network [4]. The idea is to calculate the gradient of a cost function with respect to it´s weights and biases and change these parameters to the opposite direction of the gradient, as illustrated in the Figure 6.

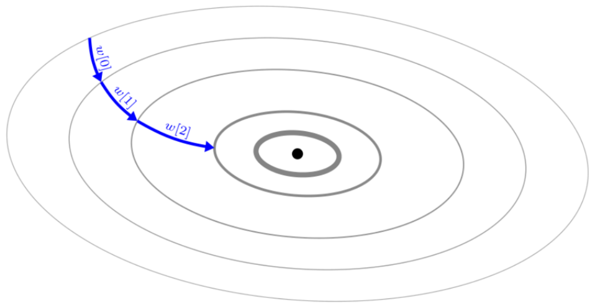

Fig. 6. Finding a minima with gradient descent.

How much the weights and biases are changed, depends on the steepness of the gradient and on the parameter called learning rate, which can be assigned manually by the programmer or by some other algorithm. Learning rate affects how fast the gradient descent algorithm converges to a minima. If the learning rate is too large, the algorithm might not converge [9].

A basic version of this algorithm called stochastic gradient descent with constant learning rate is described in Algorithm 1 [10]. X denotes the whole labeled dataset that is used for training or in other words, used to minimizing the cost function J. μ and W denotes learningrate and parameter values, respectively.

First, the gradient of a cost function is calculated with the current input x and the current weights and biases. Specifically in stochastic gradient descent, the weights and biases are updated after every input x. This process is continued until a minima is found or the performance of the neural network is good enough. An important thing to notice is that the algorithm is guaranteed to converge to a global minima only if the cost function is convex and the learning rate is sufficiently small, which is not the case for DNNs [11]. The initial values of weights and biases have thus an impact on how good the converged minima will be.

Algorithm 1 Stochastic gradient descent

```
Input: $X$, $\mu$, $W$, $J$
while not converged do
    for $x$ in $X$ do
        calculateGradient($x$, $J$, $W$)
        for $w$ in $W$ do
            $w \leftarrow w - \mu \times \frac{\partial J}{\partial w}$
```

G. Backpropagation

To be able to use the gradient descent algorithm in practice, an effective method to calculate gradients is needed. Chain rule of derivatives can be used to calculate partial derivative of a cost function with respect to any weight or bias term. Backpropagation is an effective algorithm to implement the chain rule for neural networks [4].

Backpropagation calculates each gradient in two parts [4]. First, gradients $\mathbf{g}^{(l)}$ are derived with respect to the pre-activation vectors $\mathbf{z}^{(l)}$ for each layer. For the output layer, Equation 12 is used. After that gradients can be calculated iteratively with Equation 13 [4].

$$\boldsymbol{g}^{(L)} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{p}} \odot \sigma'(\boldsymbol{z}^{(L)}) \tag{12}$$

$$\boldsymbol{g}^{(l)} = \left(\boldsymbol{g}^{(l+1)} \boldsymbol{W}^{(l+1)}\right) \odot \sigma'(\boldsymbol{z}^{(l)}) \tag{13}$$

Here $\mathbf{J} \in R^{N_o \times 1}$ denotes the cost function and $\mathbf{p} \in R^{N_o \times 1}$ the prediction or the output vector of the NN. Matrix derivatives are assumed to be in numerator layout, thus $\frac{\partial \mathbf{J}}{\partial \mathbf{p}}$ and $\mathbf{g}^{(l)}$ become row vectors. $\odot$ denotes element-wise multiplication and σ' denotes the derivative of a activation function.

It can be shown that the gradients with respect to weights and biases can be calculated with the Equations 14 and 15 [4]. To use these formulas in practice, activation vectors $\mathbf{a}^{(l)}$ and pre-activation vectors $\mathbf{z}^{(l)}$ have to be stored during the forward propagation. Also, derivatives of a cost function and activation functions have to be known. When these conditions are fulfilled, backpropagation can be implemented using only matrix multiplications and transposes.

$$\boldsymbol{\nabla}_{\boldsymbol{W}^{(l)}} \boldsymbol{J} = (\boldsymbol{g}^{(l)})^T (\boldsymbol{a}^{(l-1)})^T \tag{14}$$

$$\boldsymbol{\nabla}_{\boldsymbol{b}^{(l)}} \boldsymbol{J} = (\boldsymbol{g}^{(l)})^T \tag{15}$$

H. Momentum and batch size

Parameters that are chosen beforehand and that are not learned by the gradient descent algorithm are called hyper-parameters. One example is learning rate that was discussed together with gradient descent. Other common hyperparameters are momentum and batch size [4].

Momentum is a hyperparameter that gives gradient descent a short term memory [12]. In the standard gradient descent, only the current gradient is used when weights and biases are updated. With momentum, current gradient is summed with the previous gradient multiplied by a factor between zero and one, called momentum or decay factor. The update rule with momentum for weights is described in Equations 16 and 17, where β denotes momentum and λ denotes learning rate [12]. Formulas for updating the bias vectors are identical.

$$\boldsymbol{G}^{(k)} = \beta \boldsymbol{G}^{(k-1)} + \boldsymbol{\nabla}_{\boldsymbol{W}} \boldsymbol{J}^{(k)} \tag{16}$$

$$\boldsymbol{W}^{(k+1)} = \boldsymbol{W}^{(k)} - \lambda \boldsymbol{G}^{(k)} \tag{17}$$

Qualitatively speaking, momentum can help in situations where the cost function is flat and gradient descent would move in very small steps [12]. Momentum accelerates learning if the gradient points to the same direction several times in a row. It can also help in cases where gradient descent would otherwise end up jumping between valleys without making much progress. In some cases, it can even help gradient descent to escape a local minima [12].

Another hyperparameter worth mentioning is batch size. As mentioned earlier, stochastic gradient descent means that weights and biases are updated every time after the backpropagation algorithm has calculated gradients for an input, in other words, batch size is one. This means that the cost function of the whole training dataset doesn't necessarily get smaller after an iteration or at least stochastic gradient descent doesn't follow the straightest path to a minima [10].

Increasing batch size means that the weights are not updated as often. The idea is to first calculate gradients for several inputs and then take a mean of these gradients. This average value is then used for updating the weights as in Eqauation 18, where n denotes batch size and l denotes the number of the batch [13]. The number of the batch l goes from 0 to $\frac{N}{l}$ where N is the size of the training dataset.

$$\boldsymbol{W}^{(k+1)} = \boldsymbol{W}^{(k)} - \lambda \frac{1}{n} \sum_{i=n\cdot l}^{(l+1)\cdot n} \boldsymbol{\nabla}_{\boldsymbol{W}} \boldsymbol{J}^{(k)} \tag{18}$$

The other extreme is full gradient descent, where batch size equals the size of the training dataset. Gradients for the entire dataset are calculeted and stored and after that the mean of the gradients is used to update the parameters. Full gradient descent follows a straighter path to a minima, but it is not necessarily more effective than stochastic gradient descent. Discussion of the reasons for this is ongoing and a clear answer cannot be given [10]. It is common to use a batch size that is somewhere between these two extremes [13].

III. Simulation and Results

One goal of the simulation is to verify that the implementation of the neural network works correctly. The other goal is to study how different hyperparameters of the neural network affect the learning process. Simulations are mainly done using the MNIST dataset and we study how altering learning rate, momentum and batch size affect the performance of the neural networks.

A. MNIST dataset and system model

MNIST is a well-known database containing handwritten digits from zero to nine. The training data consists of 60000 images and the test data has 10000 images. The images are grey scale, and contain 28x28 pixels. The data is pre-processed so that the image is turned into a 784x1 column vector taking values between zero and one. Every image is labeled with the number that the image contains. The labels are one-hot encoded into a 10x1 vectors.

Fig. 7. A MNIST image.

Dimensions of the dataset determine the size of the input and output layers to be 784 and 10 neurons, respectively. We chose to use one hidden layer with 32 neurons. Both the hidden and output layer use the sigmoid as an activation function. The network is fully connected, and the initial weights of the model were randomly generated from a standard normal distribution. Every simulation with MNIST used the same initial weights. To classify an image, we took a maximum value of the output neurons. If the index number of the neuron with maximum value is the same as the number in the image, the image is considered to be classified correctly.

B. Effect of the learning rate

The first hyperparameter studied was the learning rate. In this simulation, stochastic gradient descent was used with zero momentum. Figure 8 displays how the learning rate affects the test accuracy of the network. In this context, test accuracy means the percentage at which the neural networked classified images correctly using the MNIST test data. One epoch means that the gradient descent algorithm has been calculated for the entire training data set.

After 35 epochs the best total performance was achieved with the learning rate of 0.1, which was 94.81% after 32 epochs. At the learning rate of 1, the neural network achieved over 90% accuracy already after first epoch but the difference between learning rates 0.1 and 1 was diminishing after the

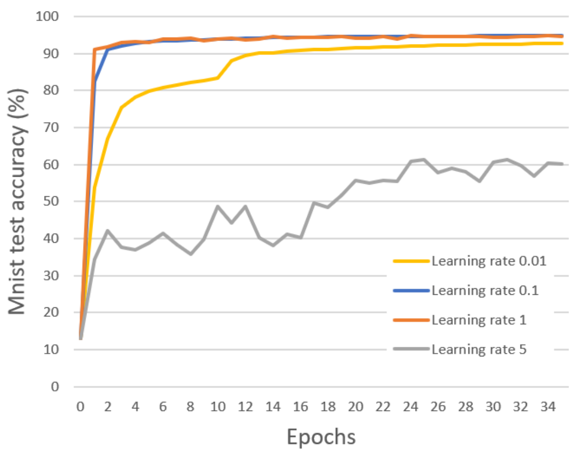

Fig. 8. Effect of the learning rate to the learning process.

second epoch. Learning rate 5 is clearly too large for this application and results in poor performance.

C. *Effect of the momentum*

To study the effect of momentum for the learning process, the learning rate was kept constant at 0.1. Simulations were done using stochastic gradient descent and the MNIST test accuracy was used for measuring the performance. Figure 9 illustrates the results for three different values of momentum.

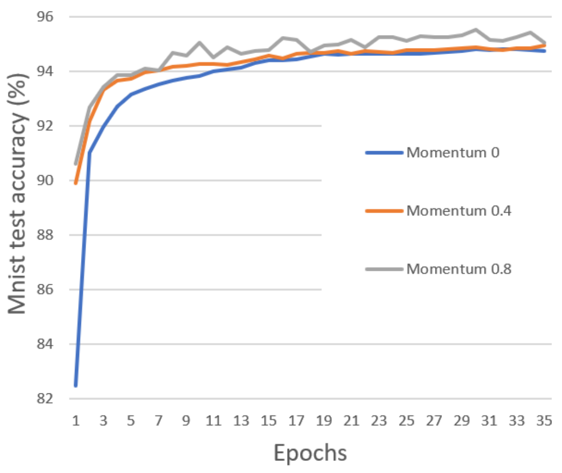

Fig. 9. Effect of the momentum to the learning process. Learning rate used was 0.1.

According to Figure 9, momentum of 0.4 leads to faster learning process. The accuracy is over 93.3% already after 3 epochs whereas this accuracy is achieved after 6 epochs without momentum. After 15 epochs momentum of 0.4 doesn't have a significant effect. Larger momentum of 0.8 on the other hand behaves much like 0.4 in the beginning, but after 7 epochs it seems to have a positive impact on the final accuracy. The best accuracy achieved was 95.5% after 30 epochs. As discussed in II-H, momentum should accelerate learning, which seems to happen in both cases. When momentum is 0.8, algorithm finds a better local minima. This might be because momentum might help gradient descent to escape a local minima as mentioned in II-H.

D. *Effect of the batch size*

To describe the effect of different batch sizes, a sample of one hundred iterations was calculated with batch sizes one, ten and fifty. The sample was taken after the neural network had been trained one full epoch with stochastic gradient descent. The results are illustrated in Figure 10. The mean square error of the MNIST training data was chosen as measurement in this case because the small sample wouldn't have a clear effect on the total test accuracy. According to [14], a good rule of thumb when increasing the batch size k times, is to increase the learning rate $\sqrt{k}$ times. This rule was applied here to minimize the effect of the relative difference in learning rates.

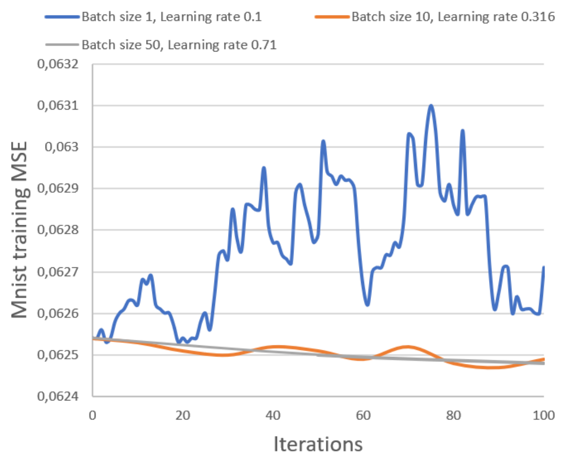

Fig. 10. Behaviour of the MSE of the training data with different batch sizes. The 100 sample iterations are taken after one epoch with stochastic gradient descent.

According to Figure 10, the mean square error varies significantly when using stochastic gradient descent. The total MSE is actually larger after this particular sample than it was in the beginning of the second epoch. As discussed in II-H, a larger batch size should minimize the cost function in a smoother way, which is the case for this sample. In this particular sample, larger batch size leads to faster learning.

To study if this is the case in general, 35 epochs were calculated with different batch sizes using the same rule of thumb for the relation between batch size and learning rate. No momentum was used in these simulations. Results are plotted in Figure 11.

Even though stochastic gradient descent minimizes the cost function in a more random way, it seems to lead to a overall faster training process, according to Fig 11. Difference is significant especially in the beginning of the process. According to [10], one possible explanation is that the noise in the steps of stochastic gradient descent may help the algorithm to escape a local minima easier.

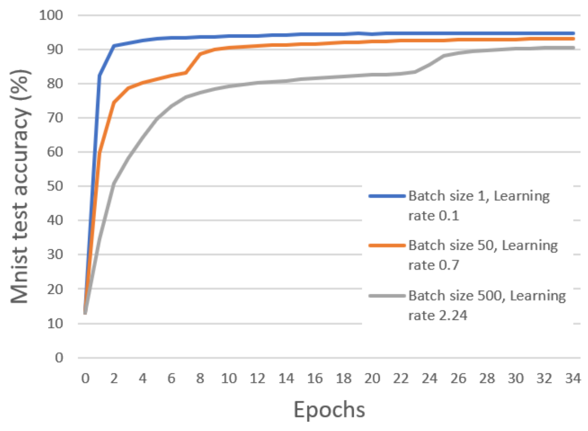

Fig. 11. Effect of the batch size to the learning process.

E. *MNIST performance*

According to Figure 11 , stochastic gradient descent works better than larger batch sizes in the beginning of the learning process. We tried to see if larger batch size would lead to better accuracy later in the process. A simulation was made so that the first 25 epochs were calculated with stochastic gradient descent, learning rate 0.1 and momentum 0.8. After 25 epochs, batch size 50 was used with learning rate 0.71 and momentum 0.8. Training and test accuracy of this simulation is plotted in Figure 12.

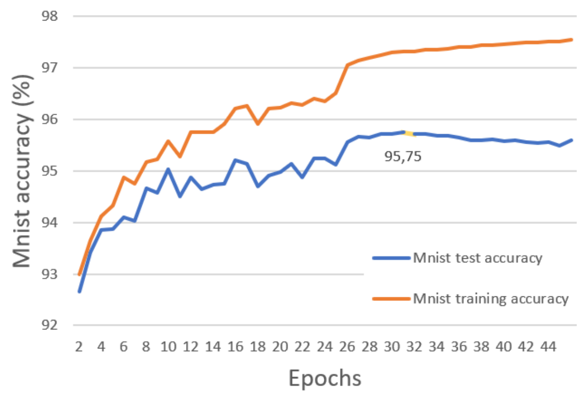

Fig. 12. MNIST test and training accuracy with combination of stochastic gradient descent and mini batch

Larger batch size seems to have positive effect on the accuracy. After the batch size is changed, the test accuracy increases 0.45% during one epoch and learning curve becomes smoother. Test accuracy of 95.75 is achieved after 31 epochs, which is 0.25% better than in simulation using only stochastic gradient descent, see Figure 8. After this, the training accuracy keeps increasing, but the test accuracy starts getting worse. In other words, the neural network starts overfitting after 32 epochs.

F. *CIFAR10*

As final simulation, we applied the neural network for a more complex image recognition problem, CIFAR10. Like MNIST, CIFAR10 has 10 classes, but instead of numbers it includes images of different animals and vehicles in color. The images are 32x32 pixels and there is 50000 training images and 10000 test images. An example of how CIFAR10 data looks like is given in Figure 13

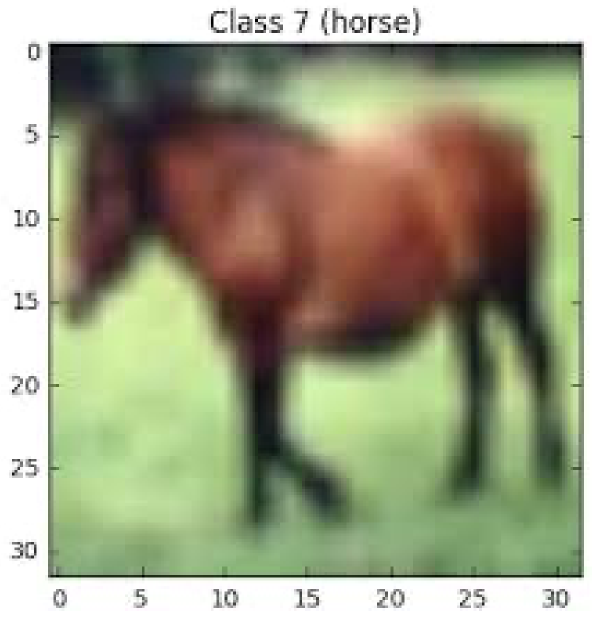

Fig. 13. A CIFAR10 image.

The preprocessing of data was done in a similar manner as with the MNIST-dataset. Images were turned into a one-dimensional vectors and labels were one-hot encoded. In this case the images had three color channels instead of one and because of that the input vector had a length of 3072. Thus the input layer had 3072 neurons and the output layer 10 neurons. Again one hidden layer was used but because of the larger input layer we chose a hidden layer with 200 neurons.

We attempted simulation with different learning rates and 0.01 gave promising results. One longer simulation was done starting with stochastic gradient descent, learning rate 0.01 and momentum 0.6. After 10 epochs batch size of 50 was used with learning rate 0.07 and momentum 0.6. Training and test accuracy of this simulation is plotted in Figure 14 and an accuracy of 41.94% was achieved.

IV. Discussion

Standard performence on MNIST usually achieve an accuracy of 98-99% using programming libraries like TensorFlow or PyTorch. Already in our first simulation, Figure 8, over 94% accuracy was achieved. As our implementation is simpler and not as optimized, lower accuracy was expected. Nevertheless, our implementation of a neural network works, and it clearly can learn using backpropagation.

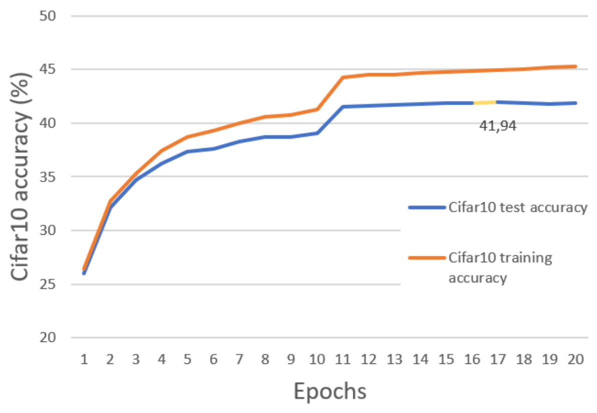

Fig. 14. CIFAR10 test and training accuracy with combination of stochastic gradient descent and mini batch

A. *Effect of hyperparameters*

The simulations with different hyperparameters gave mostly expected results. In Figure 8 we can see that learning rate 0.01 was unnecessarily small and it slowed down the learning process, at least when training on the MNIST dataset. At learning rate 5, accuracy seems to change quite randomly. This behaviour can be compared to what would happen with a one-dimensional convex cost function like x^2. If the learning rate is too large, the GD algorithm just ends up jumping from one side to the other never reaching the vertex of the parabola. Something similar seems to happen here in the higher dimensional case.

Momentum had a clear effect on the learning process. From Figure 9 it can be seen that a momentum of 0.4 leads to faster learning in the beginning and behaves similarly to the larger learning rate 1 depicted in Figure 8. When we use a momentum of 0.8, the results are a bit more surprising. In this case, the learning process is also accelerated, but now the gradient descent seems to jump over a local minima and it ends up finding a better minima. This behaviour is in line with the concept discussed in [12], but it doesn't explain why the accuracy changes non-linearly, similarly to the simulation with learning rate 5 in Figure 8. One possible explanation is that the gradient descent has reached some kind of steep valley in the cost function and that the learning rate is too large for further convergence.

The effect of the batch size is more difficult to analyze reliably. As mentioned in Section III-D, the relation between batch size and learning rate used in the simulations was more of a rule of thumb than a proven law and there is different opinions on the matter. Perhaps the results in Figure 11 are better explained by the difference in relative learning rates than by the difference in batch sizes. Same goes even to the simulations where batch size was changed after the model was trained with stochastic gradient descent. From Figures 12 and 14 it can be seen that increasing the batch size has an immediate positive effect on accuracy, but further studies would be needed to analyze if the real reason wasn't just a decrease in learning rate as a side effect due to a change in batch size.

Another interesting characteristic of simulations with different learning rates and different batch sizes is the non-linearity in Figure 8 with learning rate 0.01 after epoch 10 and a similar shape is observed in Figure 11 where larger batch sizes are used. The accuracy gets suddenly significantly better during only one epoch without any change in hyperparameters.

B. *CIFAR10*

When it comes to accuracy for the CIFAR10 dataset, Tensorflow has a tutorial model that achieves an accuracy of 72%. Standard implementations with a convolutional neural networks have usually accuracy between 80-90%. Our model had an accuracy of 41.94%, which is significantly below these values. One reason that the performance is so much worse than with MNIST is that CIFAR10 has much more complex data. Classes like automobile or cat are more general and have more variation than handwritten numbers. Also some classes like truck and automobile are quite close to each other and can be difficult to classify correctly.

A significant issue with our approach is that we reshape the two-dimensional images into one-dimensional vectors. As such, most of the spatial information is lost and the neural network has a hard time learning geometrical characteristics present in the training data. In contrast to our approach, convolutional neural networks represent images using matrices, which is more suitable for image classification. Nevertheless, an accuracy of 40% is a lot better than a random guess, which means that our implementation ,in principle, works even in this case.

V. Conclusion

The goal of this project was to reach a fundamental understanding of the inner workings of neural networks, through constructing one in Python with only the help of the linear algebra library Numpy. The project has resulted in an implementation of a neural network that consist of two main algorithms, forward pass, and gradient descent using backpropagation. The forward pass processes the input and produces an output ,i.e., a classification or a prediction. The backpropagation algorithm trains the neural network to be able to produce more accurate outputs. In addition to these two main algorithms, other features that work in conjunction with backpropagation and gradient descent were implemented: batch size and momentum. Furthermore, an analysis of the effect of these features was done. We found that the use of momentum during training improves the accuracy compared to the basic gradient descent. We also found that the use of momentum together with increasing the batch size later in the training process, had an overall improvement on the accuracy. With this method we achieved our best results which is an accuracy of 95,75% on MNIST and 41,94% on CIFAR10. However, exactly which one of these hyperparameters that have the dominant impact on the improvement is unclear and could be a subject for further studies.

Another interesting thing to investigate would be to measure the time the program has to run during each epoch. This way we could compare what effect the different hyperparameters have on the time it takes to compute an epoch and the accuracy. In addition to that, one could also examine the images the neural network makes faulty classifications on and speculate why that is. Also, we observed a nonlinearity between epochs when training which could be looked into.

Finally, a few suggestions for further development of our neural network: first, one could implement more performance enhancing features such as regularization and drop out which will add the ability to prevent overfitting of the neural network. Next, to significantly improve the performance on complex images such as those in CIFAR10, one would need to implement a convolutional neural network.

VI. Acknowledgement

We would like to thank our supervisor Henrik Hellström for his help, especially with the planning and scheduling of the project and providing resources so that we could do testing already for the early versions of the program. The meetings with Henrik were interesting and productive and he was fast to respond to emails. He also helped with the pre-processing of the MNIST and CIFAR10 data.

References

[1] S. Lynch. (2017, Mar.) Andrew ng: Why ai is the new electricity. . Andrew Ng citation. [Online]. Available: https://www.gsb.stanford.edu/insights/andrew-ng-why-ai-new-electricity

[2] D. Poulopoulos. (2022, Dec.) 5 surprising ways machine learning is revolutionizing the world. . 5 Ways ML is Revolutionizing the World. [Online]. Available: https://towardsdatascience.com/5-surprising-ways-machine-learning-is-revolutionizing-the-world-d1a0b94b3736

[3] K. D. Foote. (2021, Nov.) A brief history of neural networks. . History of NN. [Online]. Available: https://www.dataversity.net/a-brief-history-of-neural-networks/#

[4] (2022, Jan). Ml and neural networks. Material from course DD1420, KTH. [Online]. Available: https://dd1420.notion.site/ML-Neural-Networks-c89daef75b2a4d36b200bfa48af9badb

[5] R. Lamsal. (2021, Apr.) A step by step forward pass and backpropagation example. . Theory about the workings of NN. [Online]. Available: https://theneuralblog.com/forward-pass-backpropagation-example/

[6] T. Wood. (2023, Apr.) Activation function. . Theory regarding Activation Functions. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/activation-function

[7] K. Leung. (2021, Mar.) The dying relu problem, clearly explained. . Theory regarding The Dying ReLU Problem. [Online]. Available: https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24

[8] Wikipedia. (2023, Apr.) Vanishing gradient problem. . Theory regarding the Vanishing gradient problem. [Online]. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem

[9] V. Verdhan, *Supervised learning with Python : concepts and practical implementation using Python*, 1st ed. New York: Apress, 2020.

[10] (2022, Jan). Ml and optimization. Material from course DD1420, KTH. [Online]. Available: https://dd1420.notion.site/ML-Optimization-98e68f7cbfee49ff863d853a4f621b3e

[11] M. Danilova, P. Dvurechensky, A. Gasnikov, E. Gorbunov, S. Guminov, D. Kamzolov, and I. Shibaev, (2021, Nov.), "Recent theoretical advances in non-convex optimization." [Online]. Available: https://arxiv.org/abs/2012.06188

[12] G. Goh, "Why momentum really works," *Distill*, 2017. [Online]. Available: http://distill.pub/2017/momentum

[13] A. Oppermann. (2020, Apr.) Stochastic-, batch-, and mini-batch gradient descent. . Mini-Batch Gradient Descent. [Online]. Available: https://towardsdatascience.com/stochastic-batch-and-mini-batch-gradient-descent-demystified-8b28978f7f5

[14] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014. [Online]. Available: https://arxiv.org/abs/1404.5997