Implementation and Optimisation of a Neural Network

How stepping schedules affect learning

Axel Boivie, aboivie@kth.se
Victor Bellander, vicbel@kth.se

April 2021

SA115X Degree Project in Vehicle Engineering, First Level, 15.0 Credits
Supervisor: Anna Persson
KTH, Stockholm
Spring 2021

Abstract

As this project contains an implementation and optimisation of a basic neural network, it covers most of the essentials within the field of neural networks and machine learning. The focus however, lay on studying how stepping schedules affect learning. A stepping schedule is an in beforehand determined way to decrease the learning rate over the training process. It turns out that for the networks implemented, stepping schedules does not seem to make a significant difference in performance, although slight improvements from constant learning rates could be found in some cases. Different depths and widths for the neural network are studied, and the type of stepping schedule did not seem to have much of an impact. This result is somewhat surprising, as we in theory would expect to see a rise in performance when learning rate is gradually decreased over learning. We can conclude that correctly chosen parameters play a much more significant role in facilitating an effective learning process. The code for the implementation is found through the following link `https://github.com/axeboii/Neural-Network`.

Sammanfattning

Eftersom detta projekt innehåller en implementering och optimering av ett grundläggande neuralt nätverk, täcker rapporten det mest väsentliga inom neurala nätverk och maskininlärning. Fokus låg dock på att studera hur stegscheman påverkar inlärning. Ett stegschema är ett på förhand bestämt sätt att minska inlärningshastigheten (learning rate) under träningsprocessen. Det visar sig att för de implementerade nätverken verkar trappscheman inte göra någon signifikant skillnad i prestanda, även om små förbättringar från konstanta inlärningshastigheter kunde hittas i vissa fall. Olika djup och bredd för neuronnätverket studeras, och typen av stegschema verkade inte ha någon större inverkan. Detta resultat är något överraskande, eftersom vi i teorin kan förvänta oss en ökning i prestanda när inlärningshastigheten gradvis minskar över inlärningen. Vi kan dra slutsatsen att korrekt valda parametrar spelar en mycket viktigare roll för en effektiv inlärningsprocess. Koden för implementationen finns via följande länk `https://github.com/axeboii/Neural-Network`.

Preface

This report was written as part of a bachelor degree project in Vehicle Engineering at KTH, Royal Institute of Technology, Stockholm, Sweden. Before entering into this project we had no previous experience in the field of machine learning in general or neural networks in particular. The project has been a huge learning procedure for us where we have used knowledge from a mix of courses, mainly within mathematics, numerical methods and programming. Realising that there may be some new terminology introduced to the reader, we recommend keeping an eye on the word list found in appendix 7.1 for brief explanations of concepts. We hope that you will find the report interesting.

Axel Boivie & Victor Bellander, Stockholm, May 2021

Contents

1 Introduction

In today's society, machine learning plays an increasingly important role. Classification is an important machine learning task, whether it regards speech recognition, face detection, document classification or in the case of this project, handwriting recognition. A neural network is a machine learning algorithm well suited for such a task. In this project, a neural network will be implemented, trained and tested with labelled handwritten digits (MNIST dataset), with the goal of studying how stepping schedules implemented in stochastic gradient descent can help optimise training. A stepping schedule is an in beforehand determined scheme to decrease the learning rate over a training process, which can help maximising progress in training. A challenge for neural networks today is optimising training, as accurate networks with reasonable computational demands are desirable. Hopefully, a stepping schedule could help achieving this. In section 1.2 a more rigorous account of the issue dealt with in this report is presented.

1.1 Background

As the field of neural networks is so comprehensive, this background section will focus on theory related to the implementation of a neural network with the objective of classifying images from the MNIST Dataset.

1.1.1 Neural networks

A neural network (NN) is a complex type of machine-learning algorithm. The name comes from its resemblance of a biological neural network, such as those found in our own brains, where neurons are linked together. Similarly, a neural network consists of nodes linked together in a network-like fashion. The idea is that these nodes are organised in layers, with an input layer, an arbitrary number of hidden layers and an output layer. These layers consist of a number of nodes, and connections to the next layer. There are too many types of neural network structures for this report to handle, so this report is limited to fully-connected neural networks (FCNN:s). This means that every node at a specific layer is connected to all the nodes at the next layer. Networks in this report will rely on supervised learning, meaning all training data are labelled with its correct answer. (Goodfellow et. al, 2016)

In a mathematical way, a general neural network can be described as a function $\mathbf{f}(\mathbf{x};\theta) : \mathbb{R}^m \to \mathbb{R}^n$, where $\mathbf{x}$ is the input to the function, $\mathbf{f}$ the output and θ are the parameters that determines the network. In the process of resolving which hand-written digit is on a 28 by 28 pixel image, this function can be further clarified as the function $\mathbf{f}(\mathbf{x};\theta) : \mathbb{R}^{784} \to \mathbb{R}^{10}$. The input is of dimension 784, as there are 784 input pixel values which ranges from 0 to 1 depending on its brightness. The output is of dimension 10 as there are 10 digits 0-9, with each output node displaying a number 0-1 depending on how certain the network is on which digit the image depicts. Furthermore, the parameters θ are determined by the architecture of the network, described in the following section.

1.1.1.1 Architecture of a neural network

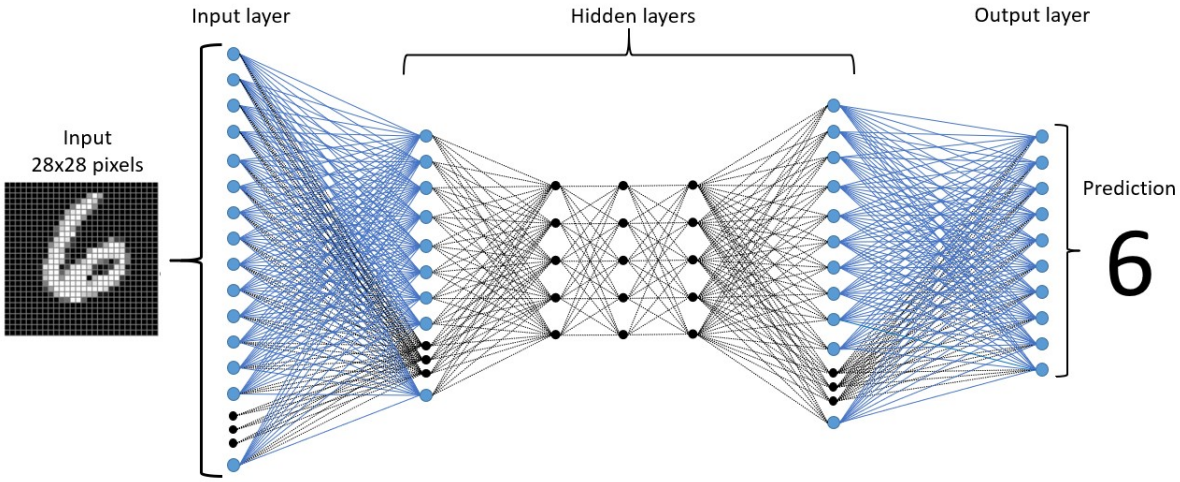


Figure 1: This figure illustrates how a basic neural network is built.

The basic architecture of a fully connected neural network is showcased in the figure above. The number of hidden layers and nodes in each of them is arbitrary, although an input layer with 784 nodes and and output layer of 10 nodes are required for this particular problem. A node in this case is simply a value, usually ranging from 0 to 1. This value is referred to as its *activation*.

The connection of one node to another node requires a *weight*. The weight determines how much the previous node's activation affects the current node's activation, mathematically described as $a^l = w \cdot a^{l-1}$, with the connection's weight as w^l, and activations given as a, with superscripts l and $l-1$ indicating which layer the nodes belong to. A common initialisation of the weights, for a network with m inputs and n outputs using a uniform distribution U, is to use the Glorot initialisation, presented below. (Goodfellow et. al, 2016)

$$w \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \approx U\left(-0.087, 0.087\right) \tag{1}$$

Additionally, we might require a threshold, under which no activation is given at the current node. This is regulated using a *bias*, which using previous notation gives $a^l = w^l \cdot a^{l-1} + b^l$, where b^l is the current node's bias. It is not uncommon for networks to not have biases at all, although the one used in this problem has. Biases are usually initialised to zero. (Goodfellow et. al, 2016)

Lastly, the current node's activation needs to be controlled to land between 0 and 1 again, so we run this value through an *activation function*, which accomplished this. The formula for the activation of one node can now be completed, with indexes j and k added to keep track of which nodes are used at the current and previous layers. With σ representing the activation function, the current node's activation is calculated through

$$a_j^l = \sigma(a_k^{l-1} \cdot w_{j,k}^l + b_j^l). \tag{2}$$

One common activation function is the sigmoid function, which is defined by

$$\sigma_{sigmoid}(z) = \frac{1}{1+e^{-z}}. \tag{3}$$

The sigmoid function gives a number close to 0 for small, or negative input z, and a value close to 1 for large z. Another common choice for an activation function is the rectified linear unit function

(ReLu). This is defined as

$$\sigma_{ReLu}(z) = max(0, z). \tag{4}$$

The ReLu function cannot give negative values, but can give infinitely large positive values, so the output range is not 0-1 as it is for the sigmoid function. This may seem problematic, but in practice it works fine as the output is still compared with the correct value, which is within the range 0-1. In modern neural networks the default recommendation is to use ReLu (Goodfellow et. al, 2016, p.171). ReLu is more computationally efficient as it just needs to pick $max(0, z)$, compared to the exponential operations for sigmoid. Additionally, the gradient of the sigmoid function vanishes for very small or large input, which is not the case for ReLu as it is either 0 or 1. The advantage of the sigmoid function is, as previously mentioned, that activation is limited between 0 and 1, and the risk of blowing up activation is avoided.

If equation (2) is used for all activations in a layer, it can easily be written using matrix notation, with the current layer's activations as $\mathbf{a}^l$, the previous layer's activations as $\mathbf{a}^{l-1}$, all the weights of the connections between the nodes in each layers as $\mathbf{w}^l$ and all the biases at the current layer as $\mathbf{b}^l$, as

$$\mathbf{a}^l = \sigma(\mathbf{w}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l). \tag{5}$$

Each node, except for the input nodes, is associated with its own bias, and each connection with its weight. Collectively, all the weights and biases of a network forms the parameters θ referenced earlier. The dimension of θ can easily grow very large, as showcased in a few examples below.

If the NN used in the current task of image classification and the NN had no hidden layers, but just an input layer of 784 nodes and output layer of 10 nodes, this would give $784 \cdot 10 = 7840$ weights and 10 biases, and the dimension of θ would be **7850**. If one hidden layer of 32 neurons are added between them, this would give $784 \cdot 32 + 32 \cdot 10 = 25408$ weights and $32 + 10 = 42$ biases, giving the dimension of θ as **25450**. If a NN with 2 hidden layers of 400 neurons are used this gives $784 \cdot 400 + 400 \cdot 400 + 400 \cdot 10 = 477600$ weights and $400 + 400 + 10 = 810$ biases, giving θ a dimension of **478410**. You easily see how the number of parameters grows rapidly with larger networks. This plays a large part in the computational demands of training a NN, as can be seen in the following section which describes this.

1.1.1.2 Training

The goal of this NN is predicting which digit is written on a given image. To accomplish this, the NN obviously needs training. The process of training a NN is the central problem in both implementation and optimisation of a NN. By 'training' the network, we mean the process of fitting all the weights and biases (gathered in θ) of the network to a given set of data. As the parameters θ are initialised randomly, an untrained NN will give random results when an image is put through the system. Now, training is done by putting images through the network and comparing the output with the desired results, which is possible as each image is labelled with its supposed digit. If the result is close to the desired result only small tweaks are made to the network, and vice versa if the result is far from the desired result. The comparison in done through a *loss-function*, also called a cost-function. One such loss-function is mean square error (MSE). With the last layer's activations in $\mathbf{a}^L$ (output of the network) and the desired values in $\mathbf{y}$, the cost C using MSE in this problem is calculated through

$$C = \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{a}_i^L - \mathbf{y}_i||^2. \tag{6}$$

Here, N is the number of images in the training dataset, which will be 60000 for this particular problem. The values in $\mathbf{y}$ will be 0 for all values except for the correct one, which will be 1. The values in $\mathbf{a}^L$ between 0 and 1 describes how likely the network thinks the given image is any of the numbers 0-9. The problem of training the network now becomes a minimisation problem of the cost C. If C is thought of as a function $C\left(\mathbf{f}(\mathbf{x}; \theta), \mathbf{y}\right)$, as $\mathbf{a}^L = \mathbf{f}(\mathbf{x}; \theta)$, the problem of minimising C

with respect to parameters θ can be accomplished through *gradient descent*. The idea of gradient descent is that the gradient of the function C with respect to parameters θ gives the direction of the steepest increase of the function C. This gives the direction of the steepest decrease as the negative of the gradient. Iterating this process through all data means that one step in the training process is calculated through

$$\theta_t = \theta_{t-1} - \epsilon \cdot \nabla_\theta \left(C \left(\mathbf{f}(\mathbf{x}_t; \theta_{t-1}), \mathbf{y}_t \right) \right). \tag{7}$$

Here, ϵ denotes the learning rate, or step size of the training process. This parameter is also the main focus of this report, and will be investigated in much further detail further on. Arrays $\mathbf{x}_t$ and $\mathbf{y}_t$ denotes the current image's pixel values as well as its label. If this process is completed for all training images, we should theoretically reach some minimum value of the cost function. Computing the gradient of C for 60000 training images, will not prove effortless. A lot of optimisation strategies must be deployed to make this problem computationally achievable, the most rudimentary of which is *stochastic gradient descent*, described in the following section. (Goodfellow et. al, 2016)

The goal of training is obviously for the network to be able to perform well on previously unseen data, the testing dataset in this case. When a network becomes 'too' well-trained over the training dataset, to the detriment of testing performance is when *overfitting* occurs. A way to illustrate overfitting is studying the gap between testing and training error, which becomes very large when the network is overfitted. The opposite of overfitting is *underfitting*, which is simply when the network is not well-trained enough. This theory explains why a network cannot simply keep training over the same training dataset to improve real world performance. (Goodfellow et. al, 2016)

1.1.2 Stochastic gradient descent

Stochastic gradient descent (SGD) is the most commonly used optimisation algorithm in deep learning today. Using SGD is a way of making a compromise between accuracy and speed when it comes to calculating the gradient of the cost function at a specific location. (Goodfellow et. al, 2016)

In theory it is possible to calculate the exact value of the gradient, and thereby direction, of every step supposed to be taken in trying to find a minimum of the cost function. This would be done simply by using all data available in a dataset when calculating the gradient in the training of the network. However, as datasets used to train neural networks are often large, it is costly to perform this type of calculation. In summary this approach leads to a very 'cautious' training algorithm, taking slow and steady steps in exactly the right direction before eventually finding a minimum of the function.(Goodfellow et. al, 2016)

The opposite philosophy by which to find the minimum would be to, instead of using the whole dataset to compute the the exact gradient at every step, only use a single data point to execute a very rough approximation of the gradient at this location. This approach makes for a fast but inaccurate algorithm. One could hope that as more and more steps are taken, the algorithm converges to a minimum even though the steps individually are not presentable as the correct direction of the gradient. In summary this leads to a fast, but somewhat uneven stepping pattern to the minimum compared to the first described approach.(Higham et. al, 2019)

SGD is a blend of the two algorithms above. With SGD every gradient is calculated using a *mini-batch* of data. A mini-batch is a share of the data in the dataset used to give a good representation of the correct gradient, without having to use all the data in every step. This makes for a faster training session as less data has to be analysed. Simultaneously, the path towards the minimum is hopefully shorter and more accurate than if only one data point is used to determine the gradient. (Goodfellow et. al, 2016)

A mini-batch consists of a number of randomly chosen data points from the dataset. The mini-batch can be created with different philosophies. One is to build it up by first shuffling the original dataset and thereafter picking the m first data points to perform the gradient approximation before taking the first step towards the minimum. For the second step, a mini-batch consisting of data points $m+1$ to $2m$ is used, and so fourth. Another course of action is to randomly select each data point into each mini-batch. This is analogue to drawing one card, m number of times from an ordered deck of cards. After each card has been drawn it is put back into the pile again. The most prone difference of the two then is the fact that first shuffling the dataset will only make use of a single data point once, whilst randomly picking from the 'deck' would make it possible for one data point to be present in several mini-batches, and some not present at all. It would also, though not very probable, be possible that the same data point is used several times in one mini-batch. (Goodfellow et. al, 2016)

1.1.2.1 Back propagation

When performing SGD, the gradient of the cost function with respect to biases and weights needs to be calculated. This is usually done by using the *back propagation algorithm*.

Wanting to find the gradient of the cost function, the partial derivatives of this function with respect to each w^l_{jk} and b^l_j in the network, has to be calculated. Here, l represents the layer in which the weight or bias is present. In the case of the bias, the j represent node number in layer l. In the case of the weights, w connects node k in layer l with node j in layer $l+1$. (Higham et. al, 2019)

Recalling from section 1.1.1.2, the expression of the cost function in equation (6) gives us the dependence of the cost in regard to the value of the nodes at the output layer. From equation (6) it is seen that the cost is only dependent on the weights and biases through the value of the output layer a^L (Higham et. al, 2019). Further, it is convenient to define a new variable z as the value of a node in layer l contributed by the value of the corresponding nodes from the previous layer, weights and biases

$$\mathbf{z}^l = \mathbf{w}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l. \tag{8}$$

Further the definition of a^l is remembered from equation (5), giving

$$\mathbf{a}^l = \sigma(\mathbf{z}^l) \tag{9}$$

with σ as the activation function. At last the node error is defined as δ^l_j

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \tag{10}$$

to describe how sensitive the cost function is to a change in the value of z of node j at layer l. Finally, this yields the four equations of back propagation

$$\begin{aligned}
&\delta^L = \Delta_a C \odot \sigma'(z^L) \\
&\delta^l = ((w^{l+1})^T)\delta^{l+1}) \odot \sigma'(z^l) && \text{for} \quad 2 \leq l \leq L-1 \\
&\frac{\partial C}{\partial b^l_j} = \delta^l && \text{for} \quad 2 \leq l \leq L \\
&\frac{\partial C}{\partial w^l_{j,k}} = a^{l-1}_k \delta^l && \text{for} \quad 2 \leq l \leq L,
\end{aligned} \tag{11}$$

all derived from the chain rule. For the full derivation of the equations (11), see reference (Higham et. al, 2019).

Lastly, the notation $\odot$ needs to be described. The sign denotes a *Hadamard product* which is an element wise multiplication of two vectors of the same size. This means that in a Hadamard multiplication of vectors x and y ($x \odot y$), the component i of the Hadamard product is calculated though $(x \odot y)_i = x_i \cdot y_i$. (Higham et.al, 2019)

1.1.2.2 Learning rate and stepping schedules

One of the most important parameters when setting up the training of a neural network is the learning rate ϵ. The learning rate is the parameter deciding the step size by which to go in the opposite direction of the gradient. The value of the learning rate is important in the way that a well chosen learning rate will be beneficial for the learning procedure. An ϵ too large will make the learning procedure very oscillative. On the other hand, a small ϵ makes the learning slow, though delivering a relatively smooth path towards the minimum.

An issue when dealing with SGD in tandem with a constant learning rate during a whole training session is the fact that SGD introduces noise in the gradient calculations. This is alright as long as the training algorithm is not close to a local minimum in the cost function. When closing in on a minimum though, the noise in the gradient calculation will be proportionally larger to the actual gradient leading to taking random steps around the minimum. This phenomenon is often called a *noise ball*. It can be explained as when the number of iterations in training goes to infinity, SGD converges to a noise ball. The step size is simply too large to enable SGD to get any closer to a minimum, and 'bounces' around the minimum. The size of the noise ball is proportional to the learning rate ϵ. Mini-batching can minimise the noise ball additionally. For accurate solutions, a small noise ball is ideal. There is however a trade-off, as a low learning rate increases convergence time. For the proof of noise ball, see reference (Cornell CS, 2017).

To counteract the noise ball, a none constant learning rate could be helpful. The denotation of ϵ could now be changed to ϵ_k, where k ranges from 1 to n during n iterations of the training process. How ϵ_k changes during training is defined by a *stepping schedule* providing a good set of values for the learning rate. More on this later. (Goodfellow et. al, 2016)

In theory, not using SGD but the whole dataset, to calculate the gradient in each step would make a stepping schedule superfluous. This is due to the fact that this method, though slow, will produce the exact value of the gradient at each location of parameter space. This means that close to a local minimum, the gradient will be small and at that minimum, the value of the gradient would be zero. (Goodfellow et. al, 2016)

There are a number of different stepping schedules providing more stability and reducing the impact of noise close to minimums of the cost function. To begin with there are a number of set schedules depending on mathematical functions. Examples of these types of schedules are *exponential decay* (equation (12)), *inverse time decay* (equation (13)), *piece wise constant decay* (equation (14)) and *polynomial decay* (equation (15)) (TensorFlow, 2021). Below, the formulas of calculating the present learning rate can be seen.

$$\epsilon_k = \epsilon_0 \cdot \gamma^k \tag{12}$$

$$\epsilon_k = \frac{\epsilon_0}{1 + \gamma^k} \tag{13}$$

$$\epsilon_k = (\epsilon_0 - \epsilon_n) \cdot (1 - \frac{k}{n}) + \epsilon_n \tag{14}$$

$$\epsilon_k = (\epsilon_0 - \epsilon_n) \cdot (1 - \frac{k}{n})^p + \epsilon_n \tag{15}$$

Above, ϵ_0 is the initial learning rate, ϵ_n is the end learning rate, γ is the *decay rate*, k is the present step in the training session, n is the total number of changes of the learning rate in the stepping

schedule and p is a selectable coefficient.

As a part of implementing a stepping schedule, it is central to know when these learning rates should be updated. This can be done in several ways. One way could be to update the learning rate after each mini-batch and therefore after each taken step. Another philosophy could be to update the learning rate after each *epoch*, which is another term needing an explanation. An epoch is defined as when an entire dataset has been passed through the neural network once in the training session (Sharma, 2017). Usually the training session consists of several epochs, meaning the whole dataset will have been used to train the neural network several times in trying to get a sufficiently accurate network.

It can be shown that inverse time decay, given by equation (13), is an optimal step size scheme for convex problems (Cornell CS, 2017). However, as neural network training is not ordinarily a convex optimisation problem, this stepping schedule might not be optimal for this particular problem.

1.1.2.3 Adaptive learning rate and ADAM

The stepping schedules presented in the sections above does not take into account what the surrounding of the cost function look like or what step sizes and gradients have been used before. From an optimisation stand point it could be beneficial to choose ϵ_k based on these criteria and not only follow a fix mathematical formula. This can be done using an algorithm with an adaptive learning rate.

One of the most respected adaptive learning rate algorithm today which have shown to be effective in a lot of different deep learning implementations is the *ADAM algorithm* (DeepLearningAI, 2017). ADAM in itself is a continuation and development of two other optimisation algorithms called *momentum* and *RMSprop* (Bushaev, 2018).

The basic idea behind momentum is to take into account not only the value of the gradient at the present location, but also let it be influenced by an exponentially decaying moving average of previous values of the gradient. In theory this means speeding up learning in especially steep or ravine-shaped regions in parameter space. A momentum implementation could also conquer a small hill to later finding its way to a new 'better' minimum, contrary to a non adaptive stepping schedule getting stuck. (Goodfellow et.al, 2016)

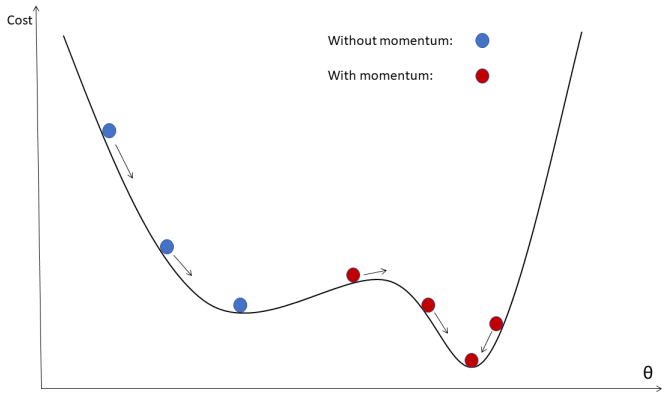


Figure 2: Shows difference in behaviour between momentum and non-momentum implementation.

The RMSprop algorithm calculates an appropriate step size in parameter space by scaling all the model parameters, weights and biases, 'inversely proportional to the squareroot of the sum of all the

historical squared values of the gradient [and] changing the gradient accumulation into an exponentially weighted moving average.'(Goodfellow et.al, 2016, p. 303). By doing this, RMSprop changes learning rate as the training proceeds but only as much as the nearest history of surroundings suggest. This is favourable as the shape of parameter space can change during the path towards a minimum, requiring an ability to local adaptation in the learning rate.(Goodfellow et.al, 2016)

ADAM is as mentioned a combination of the two algorithms explained. The first step of the algorithm is to calculate the influence of momentum and RMSprop on the step size.

$$\begin{aligned} V_{dw_t} &= \beta_1 V_{dw_{t-1}} + (1-\beta_1) dw_t \\ S_{dw_t} &= \beta_2 S_{dw_{t-1}} + (1-\beta_2) dw_t^2 \\ V_{db_t} &= \beta_1 V_{db_{t-1}} + (1-\beta_1) db_t \\ S_{db_t} &= \beta_2 S_{db_{t-1}} + (1-\beta_2) db_t^2 \end{aligned} \tag{16}$$

Equation (16) shows formulas to calculate the following: 'V_{dw_t}' and 'V_{db_t}' which are the momentum exponentially weighted average at step t for weights and biases. 'S_{dw_t}' and 'S_{db_t}' which are the RMSprop exponentially weighted average at step t for weights and biases. In equation (16) dw and db are the gradient of the cost function with respect to weights and biases calculated with back propagation at step t. 'β_1' and 'β_2' are two hyperparameters commonly set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. At the start of the training the values of V_{dw_t}, S_{dw_t}, V_{db_t} and S_{db_t} are set to be 0. This yields that at iteration $t = 1$, the values of $V_{dw_{t-1}}$, $S_{dw_{t-1}}$, $V_{db_{t-1}}$ and $S_{db_{t-1}}$ are all zero. (DeepLeraningAI, 2017)

It is necessary to correct the calculated parameters in equation (16) from a certain degree of bias. How this is done is shown in equation (17). (DeepLearningAI, 2017)

$$\begin{aligned} \hat{V}_{dw_t} &= \frac{V_{dw_t}}{1-\beta_1^t} \\ \hat{S}_{dw_t} &= \frac{S_{dw_t}}{1-\beta_2^t} \\ \hat{V}_{db_t} &= \frac{V_{db_t}}{1-\beta_1^t} \\ \hat{S}_{db_t} &= \frac{S_{db_t}}{1-\beta_2^t} \end{aligned} \tag{17}$$

Finally, the update of the weights and biases are as follows in equation (18).

$$\begin{aligned} w_t &= w_{t-1} - \alpha \frac{\hat{V}_{dw_t}}{\sqrt{\hat{S}_{dw_t}} + \lambda} \\ b_t &= b_{t-1} - \alpha \frac{\hat{V}_{db_t}}{\sqrt{\hat{S}_{db_t}} + \lambda} \end{aligned} \tag{18}$$

Here, 'α' is a parameter that has to be tuned for the specific neural network and problem. λ is a parameter to avoid division by zero, set to 10^{-8}.

1.1.3 TensorFlow, Keras and MNIST

TensorFlow is an end-to-end open source platform for machine learning by Google. Specifically, an interface within TensorFlow, Keras is used in this project. Keras is a deep learning API for Python, with a library of functions for building, training and testing NN:s. Within Keras there are multiple datasets built-in, which can be used for training the network. For this project, mainly the MNIST dataset was used, MNIST for 'Modified National Institute of Standards and Technology'. The MNIST dataset consists of 70000 images of hand-written digits, 28 by 28 pixels in size, all

labelled with what images these are supposed to be. Of the 70000 images, 60000 are reserved for training the network and 10000 for validation and testing. Examples from the dataset are shown below. (TensorFlow, 2021)

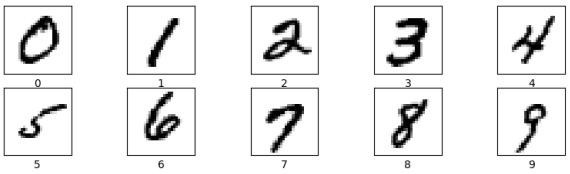


Figure 3: MNIST

1.2 Issue

Issues being dealt with in this report are the following:

- Does a stepping schedule improve the performance of the training of a neural network?
- Will a stepping schedule improve the classification of data made by the neural network?
- If a stepping schedule proves to be beneficial to use, does it hold for all circumstances?
- How do stepping schedules hold up to adaptive optimisation methods, such as ADAM?

1.3 Delimitations

When investigating the effect of using a stepping schedule while training a neural network, a few delimitations have been done. The delimitations are presented below:

- Only fully connected neural networks have been studied.
- Only three different sizes of neural networks have been analysed, with 2 layers maximum.
- Only five traditional stepping schedules and one adaptive optimisation algorithm have been tested.
- Only one dataset have been used.

2 Methodology

Here follows an explanation of the methodology used in this project.

2.1 Literature

The field of neural networks contain many concepts, as can be seen within the background section. A good piece of literature covering all of the basics is *Deep learning* by Goodfellow et. al. This book became the main reference for the first part of our literature study, offering insight into all basic and higher level aspects of a neural network. Following the first part of the study, which focused on theoretical aspects of neural networks and learning, a deeper understanding of practical forms was required. For this part, the report *Deep Learning: An Introduction for Applied Mathematicians* by Higham et. al, became a large part of the study. As this report repeats many of the conceps presented by Goodfellow et. al, but also implements them as examples in MATLAB, the gap between theory and practice was reduced.

2.2 Implementation

The goal of the implementation is to build, train and evaluate a neural network in Python. Bearing this in mind, the implementation began with a simple implementation of a NN in TensorFlow. This enabled us to adjust settings such as layers, nodes, activations, stepping schedules and such in order to obtain an understanding of how each setting affects the performance of a neural network, without actually understanding the code. This played a vital role in building an understanding of a NN, and preparing for writing a NN from scratch in Python.

The process of implementing the NN from the ground up was to begin with very straightforward. A NN needs layers, with weights and biases connecting each node in one layer to each node in the next layer. They need an activation function as well as a cost function. The NN needs parameters such as layer sizes and numbers, initialisations of weights and biases. All the weights of a layer are implemented as a matrix where rows and columns corresponds to the previous and the current layers respectively. The biases of the layers are implemented as an array with one value per node in the current layer. The weight-matrices and bias-arrays for all layers are kept in basic python-lists. Now, the activation of one layer can be calculated using equation (5). The activation function used is ReLu, seen in equation (4). With all this in place, the architecture of the NN is basically finished. The challenge becomes training the network.

The process of training the network relies heavily on the mathematical conclusions presented in the background section, namely the equations of back propagation (11). The implementation of back-propagation was assisted by the examples in MATLAB presented by Higham et. al, and finally yielded a function which calculated the gradient of the cost function with respect to all the weights and biases for a mini-batch. The algorithm then takes a step in the opposite direction of the gradient for all parameters, in accordance with the process of stochastic gradient descent. SGD is thoroughly presented in the background section 1.1.2.

Next, the training process was divided into sections, with each running a certain number of mini-batches. Using SGD means that not all the 60000 images must be treated, since images are selected randomly from the dataset. This sped up the training process significantly. The default choice was to evaluate 10% of the data (6000 images) in each section in mini-batches of 16, giving 375 mini-batches per section. Typically, 10 sections were ran, giving the total number of images treated as 60000, and the total number of batches as 3750. Further, implementing stepping schedules means that after each section, the learning rate is decreased according to the given schedule.

To compare schedules, two parameters were studied; training loss and testing accuracy. These

were evaluated after each batch, with a batch size of 16. Training loss was calculated using MSE in accordance to equation (6) for all images in the batch. Testing accuracy was calculated using 50 random images from the testing dataset. The fraction of these correctly predicted gave the training accuracy. Using all the 60000 training images in batches of 16 in training, this gives $60000/16 = 3750$ batches and as many data-points for training loss and testing accuracy.

2.3 Optimisation and visualisation

When investigating the stepping schedules' effect on the loss function and accuracy, a trial and error philosophy was used. The parameters, such as learning rate and decay rate, was tuned until satisfactory properties of the network was obtained. The optimal parameter for every stepping schedule was noted and used in the evaluation of the schedules. Also, worse than optimal parameters where chosen and noted for each stepping schedule to make it possible to present the impact of the parameters in plots.

When collecting data for visualisation, ten separate learning sessions were performed to make the results more trustworthy and less dependent on randomness. The data points collected were processed so that the mean of the ten training sessions was gained. To make the results more readable the number of data points was reduced from the number of steps taken, 3750, to 50. This was made by averaging every 75 steps to generate one single data point from that information. This reduced the noisy nature of the original data. The results were presented in plots. This approach to visualisation was used independently of the test performed and shown.

2.4 Parameter settings

The parameters used if nothing else is noted is seen below.

Table 1: The parameters used with the different schedules.

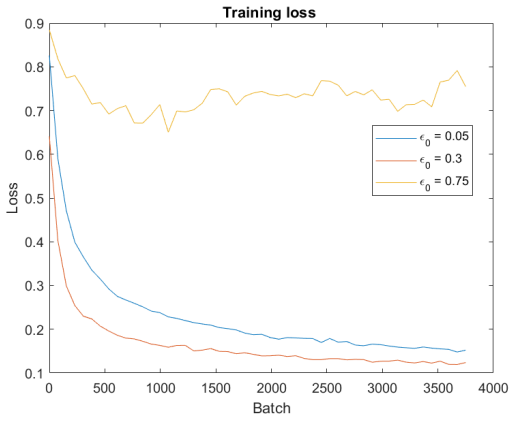
Stepping schedule	**Inital learning rate ϵ_0**	**Decay rate γ**	**End learning rate**
No schedule	0.3	-	-
Piece wise constant decay	0.3	linear	0.055
Exponential decay	0.5	0.75	-
Inverse time decay	0.5	0.5	-
Polynomial decay	0.3	2.5	0.01

When ADAM is used the parameter $\alpha = 0.003$ if nothing else is noted.

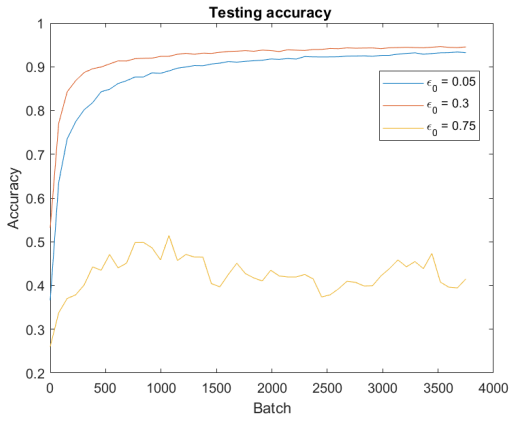
3 Result

3.1 Training with constant learning rate

Figure 4 showcases how learning rate affects the training process. Figure 4a shows the loss of each batch, and Figure 4b shows the accuracy when testing the network with test-images after each batch. No stepping schedule is used. The network has one hidden layer with 32 nodes.



(a) Training loss.

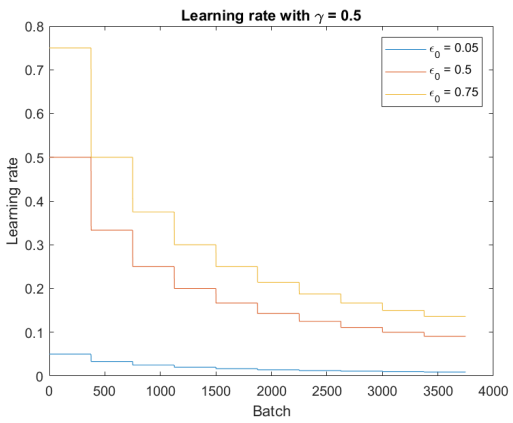


(b) Testing accuracy.

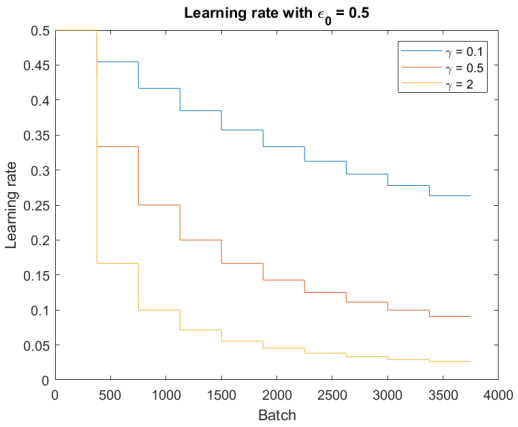
Figure 4: These figures show how loss and accuracy are affected by the learning rate ϵ in the case of no stepping schedule.

3.2 Training with stepping schedules

Next, the inverse time decay schedule is studied. Varying the parameters ϵ_0 and γ the following learning rates are used.



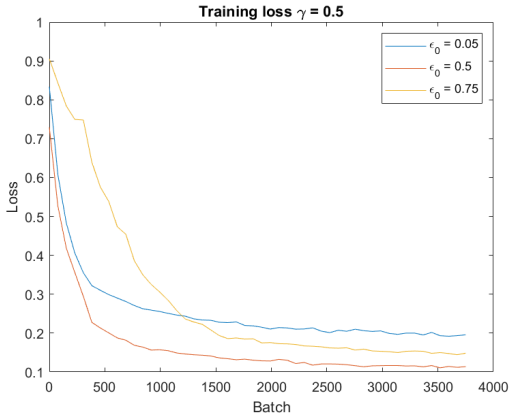
(a) Learning rate with varying initial learning rate ϵ_0.



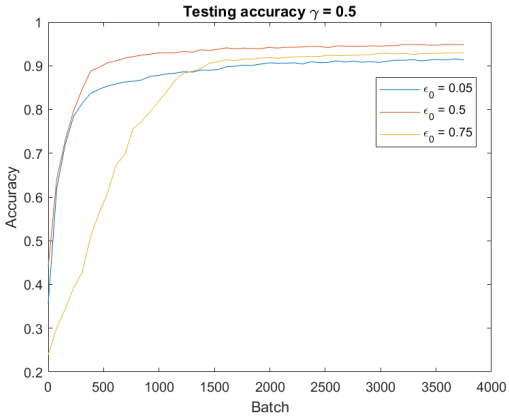
(b) Learning rate with varying decay rate γ.

Figure 5: These figures show how the learning rate is affected parameters ϵ_0 and γ for the inverse time decay schedule.

Now, these learning rates are implemented and gives Figure 6 and 7, which shows loss and accuracy corresponding to a change in initial learning rate with either a constant decay rate of $\gamma = 0.5$, or constant initial learning rate $\epsilon_0 = 0.5$.

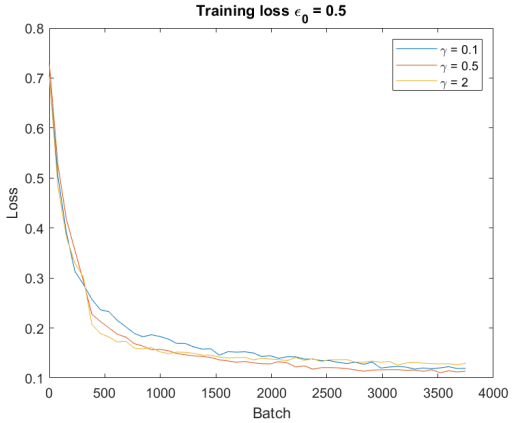


(a) Training loss.

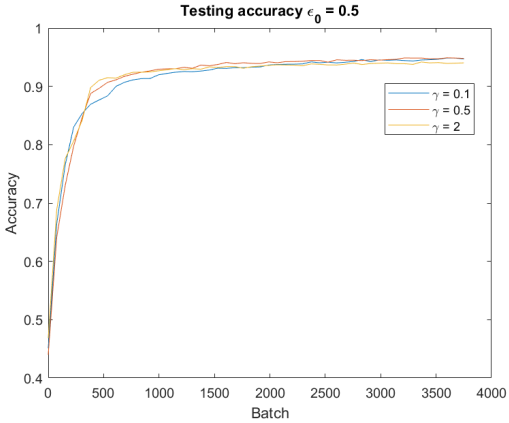


(b) Testing accuracy.

Figure 6: These figures show how loss and accuracy are affected by initial learning rate ϵ_0 for the inverse time decay schedule.



(a) Training loss.



(b) Testing accuracy.

Figure 7: These figures show how loss and accuracy are affected by decay rates γ for the inverse time decay schedule.

The other stepping schedules are now studied. Figure 8 shows how the learning rate varies during a training session for settings found in Table 1, whilst Figure 9 shows how loss and accuracy are affected.

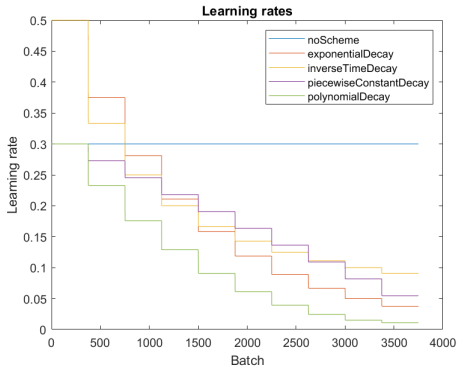
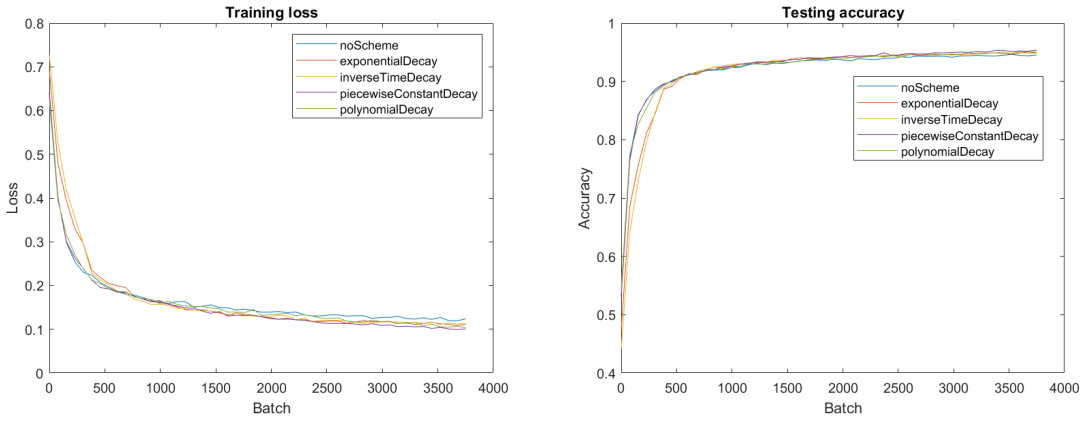


Figure 8: This plot shows how the learning rate varied in the different schedules.



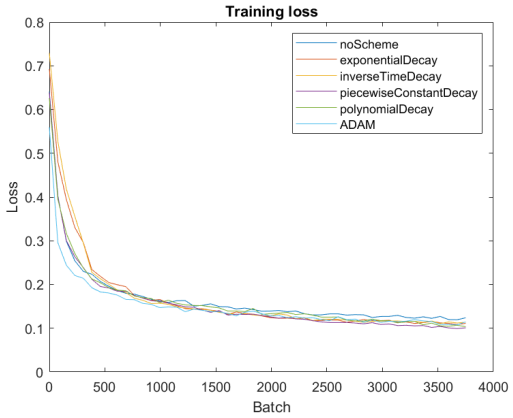
(a) Training loss.

(b) Testing accuracy.

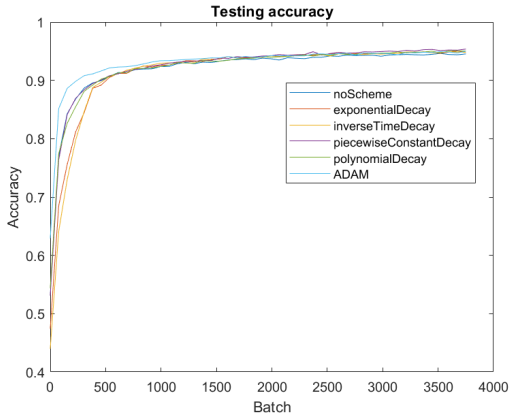
Figure 9: These figures show how loss and accuracy are affected by different stepping schedules.

3.3 Training with adaptive learning rate

ADAM is now compared to the regular stepping schedules. Figure 10 shows the performance.



(a) Training loss.

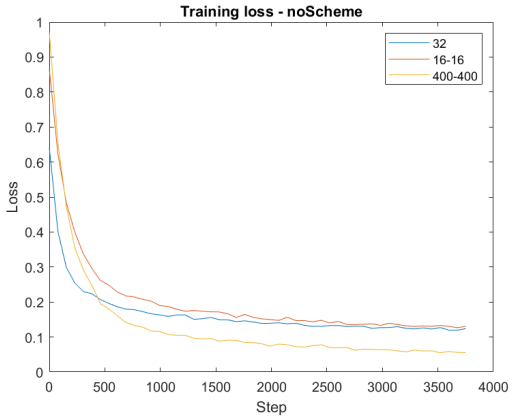


(b) Testing accuracy.

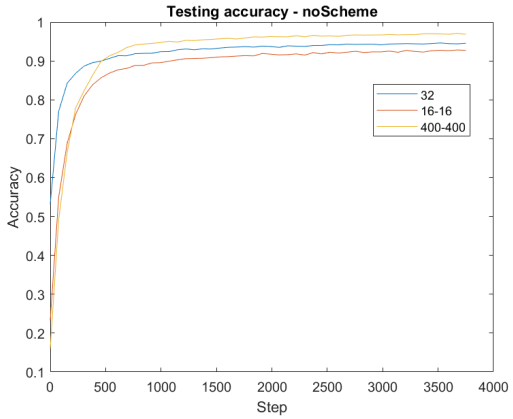
Figure 10: These figures show how loss and accuracy are affected by ADAM with respect to the other stepping schedules.

3.4 Network size

It is interesting to study how the results might be impacted by network structures and sizes. Figure 11 shows loss and accuracy when using a constant learning rate $\epsilon = 0.3$ for three different sizes of the hidden layers.



(a) Training loss.

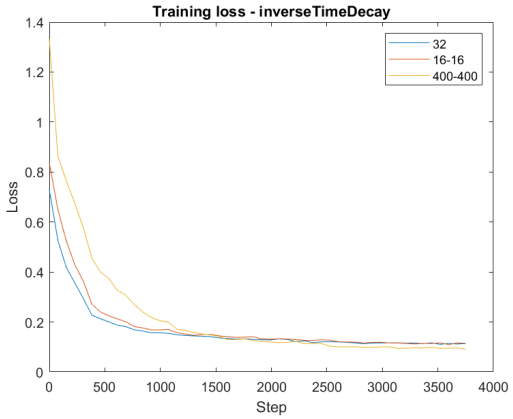


(b) Testing accuracy.

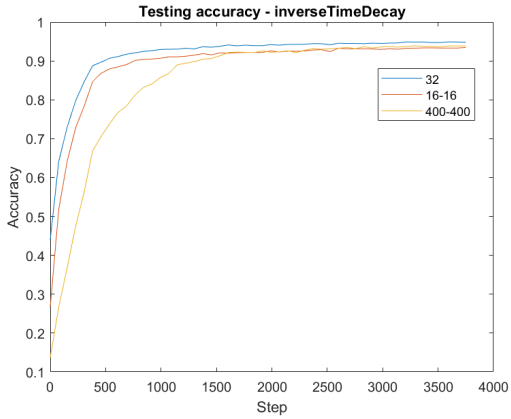
Figure 11: These figures show loss and accuracy with no stepping scheme and hidden layers of different sizes with $\epsilon = 0.3$.

As the slope of the yellow 400-400 curve is so steeply declining throughout training, a test with longer training was implemented. It ran for 187500 batches, 50 times the current amount. A result of testing accuracy over 98% was achieved, although severe overfitting was observed.

Implementing the inverse time decay schedule for the different sizes gives Figure 12, which shows loss and accuracy with the three network sizes. Parameters from Table 1 are used.



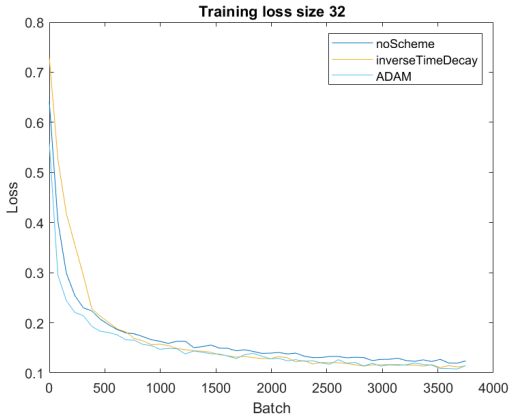
(a) Training loss



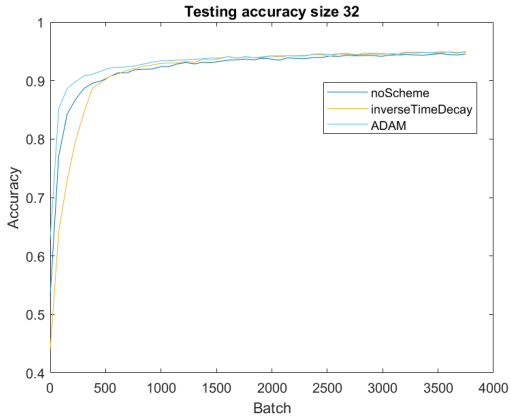
(b) Testing accuracy.

Figure 12: These figures show loss and accuracy with inverse time decay using $\epsilon = 0.5$, $\gamma = 0.5$ and hidden layers of different size.

In order to study how stepping schedules are affected by network sizes, no scheme, inverse time decay and ADAM are plotted together for the different network sizes. This gives Figures 13, 14 and 15. Parameters from Table 1 have been used.

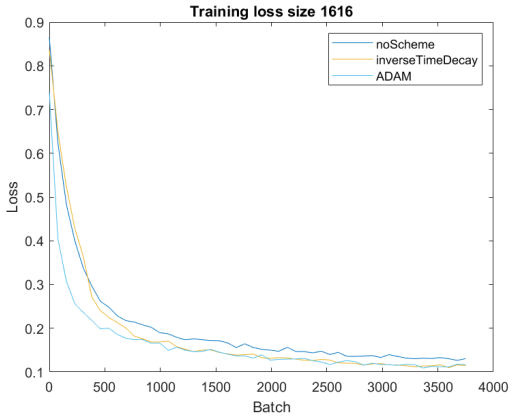


(a) Training loss

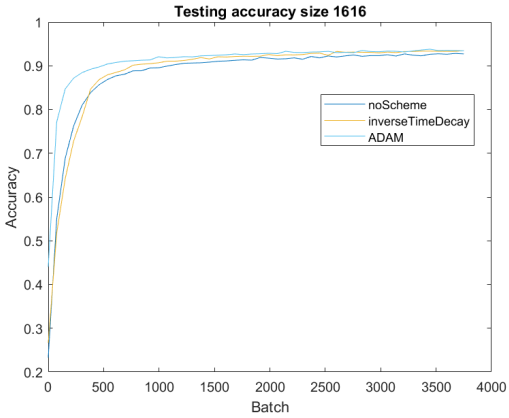


(b) Testing accuracy

Figure 13: Loss and accuracy for a hidden layer of size 32 with respect to different stepping schedules.

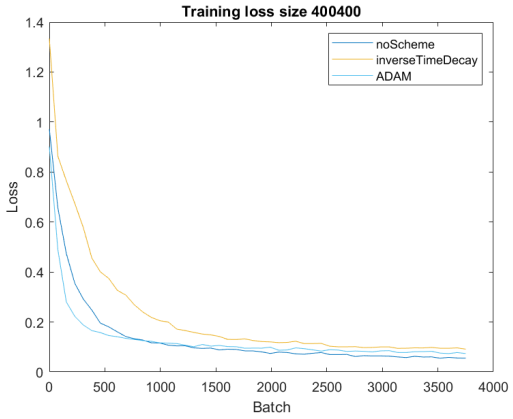


(a) Training loss

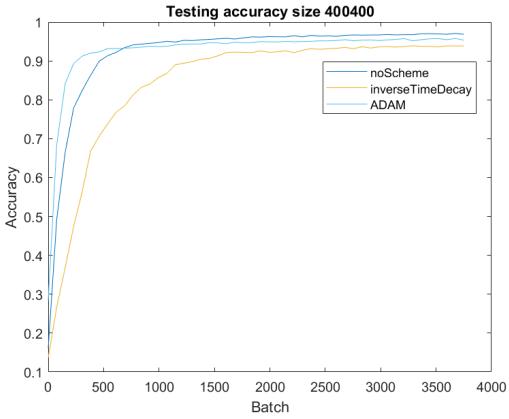


(b) Testing accuracy

Figure 14: Loss and accuracy for hidden layers of sizes 16-16 with respect to different stepping schedules.



(a) Training loss



(b) Testing accuracy

Figure 15: Loss and accuracy for hidden layers of sizes 400-400 with respect to different stepping schedules.

3.5 Code

The written code containing the full implementation is found in the following github repository. All generated data (loss and accuracy values) are also found here as .dat files, named with the particular parameter studied and setting used.
`https://github.com/axeboii/Neural-Network`

4 Analysis

4.1 How learning rate affects learning

Studying the results from Figure 4, it is clear how the training of a network is dependant on the learning rate. For constant learning rate throughout training, the setting $\epsilon = 0.3$ proved to be the best. A lower setting, $\epsilon = 0.05$, gives a slower learning and would probably reach the level of the higher setting after some more training. A higher setting such as $\epsilon = 0.75$ proved to not really give much learning at all, as the SGD converged to a large noise ball. These results are in line with what we would expect from theory, and a valuable lesson has been learnt as to which learning rate works best for this problem.

4.2 How stepping schedules affects learning

Optimising the settings for the inverse time decay schedule, we see how initial learning rate ϵ_0 and decay rate γ affects the learning process. Figure 6 show that a high initial learning rate, $\epsilon_0 = 0.75$, gives slow learning as it converges to a large noise ball when learning rate is high. Clear jumps can be seen when the learning rate is lowered according to schedule. A lower initial learning rate, $\epsilon_0 = 0.05$, gives fast learning in the beginning, but soon becomes slow and never reaches high accuracy. Figure 7 shows that the decay rate γ does not seem to have as large of an impact on learning as ϵ_0. This is surprising, as the variation in learning rate is quite drastic, which can be seen in Figure 5b. This may be explained by the fact that the learning rate is the same for the first 375 batches, before the first step in the schedule is taken. This means that the minima has been approached sufficiently for the higher decay rates to still yield effective learning.

Regarding the three other stepping schedules, it is from Figures 9a and 9b not obvious to see how they affect learning. The difference between them is very small, as can be seen from Figure 8. Unsurprisingly, this leads to very similar loss function values and testing accuracy between the four schedules. However, since they all are very different to a constant learning rate, it is not far fetched to expect the training process to also differ significantly. It turns out that all the schedules performs slightly better than no stepping scheme in the end and slightly worse in the beginning of training. For the default network, polynomial decay gave the best result in the end, although the margins are very small to other schedules. Furthermore, the optimal stepping schedule for convex problems, inverse time decay, was not optimal for this particular problem. This is not unexpected, as the training of a neural network is not a convex problem.

We can conclude that the largest contributing factor to an effective training algorithm is a well-chosen initial learning rate, rather that choosing the right stepping schedule.

4.3 How adaptive schedules affects learning

As described in section 1.1.2.3, the adaptive stepping schedule ADAM has a few unique properties that hopefully would improve the learning of a neural network. As can be seen in Figure 10 the ADAM algorithm seems to accomplish similar testing accuracies as the more traditional stepping schedules. However ADAM seems to be beneficial to the speed of the improvement of the network parameters, shown by the steeper slope in the beginning of the graph in Figure 10b. The speed of the way the accuracy improves with ADAM may be due to a very well adapted step size to the surrounding parameter space. The ADAM algorithm is probably able to adapt better than a simpler stepping schedule in this case. In theory ADAM should have the opportunity to perform better also further into the training session. Why this is not true for this case, as can be seen in Figure 10b, is still unclear.

Analysing Figures 13, 14 and 15 the same basic trends as above can be seen. The ADAM algorithm seems to have an advantage in the early stages of the training with the other stepping schedules managing to equal or better ADAM's result in the end.

4.4 How network size affects learning

Looking at the results presented in section 3.4, some interesting points can be made. At the larger network hidden layer size, 400-400 nodes, no stepping scheme performed significantly better than with the default hidden layer of 32 nodes. At the same time, the smaller network of 16-16 nodes performed worse. Seemingly, more parameters gives a better performing network. On the contrary, the behaviour is completely different when training with inverse time decay, which is the optimal stepping schedule for convex problems. Here, the larger network of 400-400 performs worse in the beginning, and basically on par in the end of training. The same is true for the smaller network of 16-16, although it is faster in the beginning of training. The network with one layer of 32 performs best. We do also see some overfitting tendencies for the 400-400 and 16-16 networks, as their training loss is lower than the 32-network whilst having a lower testing accuracy.

Comparing optimisation schedules for the three network sizes it can be seen that inverse time decay is the slowest for all sizes, but recovers and performs on par with ADAM at least for smaller networks. For the smaller networks, no stepping scheme performs averagely in the beginning and worst in the end. For the larger network however, no schedule clearly performs the best, although ADAM is faster in the beginning.

4.5 Sources of error

As the implementation of a neural network in Python is quite extensive, it is of course possible that small errors can occur in code. However, as testing accuracy over 98% was achieved it is reasonable to assume that no large errors are present in the network's implementation.

Another source of error concerns the drawn conclusion. It is possible that not enough different networks and stepping schedules were tested, which leads to the conclusion of whether or not stepping schedules helps optimise training possibly not being applicable for all networks. All networks implemented and tested in this project are relatively shallow, and consists of quite few nodes. It is possible that stepping schedules would make a more significant difference if deeper and wider networks were to be tested.

5 Conclusion

Summing up the results it is clear that a stepping schedule does not improve learning independently of other parameters. Generally speaking, the ADAM algorithm seems too speed up initial learning but as this speed trails off, it is unclear if that is significant in an implementation situation. Looking at the other stepping schedules, they seem to perform better than the constant scheme in certain situations, but worse in others. It is important to know the delimitations of this report as it is only concluded that for the tested circumstances, that a stepping schedule does not tend to improve learning significantly. Further testing is required to know whether stepping schedules are beneficial in the general case or not. If taking over from here it would be interesting to investigate the stepping schedules' impact when using other data sets together with different depths and widths of hidden layers. Also other parameters such as batch size and how often the learning rates are updated could be interesting parameters to tweak.

Lastly we could bring up some of the ethical aspects of the issues brought up in this report, or in general, to machine learning as a whole. Of course the fast development in machine learning makes for a few ethical questions needing an answer. As seen in China, governments can use machine learning and AI to control and monitor its citizens in a way that they deem suitable. This is a violation of the human right of freedom. On the other hand AI and machine learning could be very helpful in our everyday life when used in non surveillance and non war situations.

Acknowlegdements

We direct a great thank you to Anna Persson, our supervisor, who helped us with finding relevant literature and discussing ideas and results throughout this project. We would also like to thank Anton Sederlin who helped us setting up a Github repository for the code in this project.

6 References

Bushaev, Vitaly. 2018. Adam — latest trends in deep learning optimization.
https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c
(Retrieved 24.03.21)

Cornell CS. 2017.
https://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture2.pdf
(Retrieved 12.04.21)

DeepLearningAI. 2017. Adam Optimization Algorithm (C2W2L08). [video].
https://www.youtube.com/watch?v=JXQT_vxqwIs
(Retrieved 24.03.21)

Goodfellow, Ian; Bengio, Yoshua and Courville, Aron. 2016. Deep Learning. MIT Press.
http://www.deeplearningbook.org (Retrieved 19.03.21)

Higham, Catherine and Higham, Desmond. 2019. Deep Learning: An Introduction for Applied Mathematicians. Published by SIAM.
https://epubs.siam.org/doi/pdf/10.1137/18M1165748 (Retrieved 23.03.21)

Sharma, Sagar. 2017. Epoch vs Batch Size vs Iterations.
https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9
(Retrieved 24.03.21)

TensorFlow. 2021.
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules
(Retrieved 23.03.21)

7 Appendix

7.1 Word list

Activation function - The function σ used to control the activation of a node, see equation (5).

ADAM - An optimisation algorithm which uses an adaptive learning rate, see section 1.1.2.3.

Back propagation - The method used for updating weights and biases of a network in training, see section 1.1.2.1.

Bias - A node parameter b, used to calculate the activation of a node, see equation (5).

Cost function - A function to calculate the distance from the prediction and the optimimum value of an image. In this project, mean squared error is used, see equation (6)

Dataset - A cluster of data, in this case the MNIST dataset is used.

Decay rate - A parameter used in certain stepping schedules to determine how fast the learning rate is decreased.

Keras - A deep learning API for python within TensorFlow, with a library of functions for building, training and testing neural networks.

Layers - Layers consist of nodes and are what builds the network. Divided into an input layer, hidden layers and an output layer.

Learning rate - A parameter in SGD which determines how large steps are taken in the opposite direction of gradient.

Loss function - Same as cost function

Mini-batch - A group of randomly selected data points used to calculate a gradient.

MNIST - a dataset containing 70000 labelled images of handwritten digits

Noise ball - A convergence size for SGD.

Overfitting - Overfitting occurs when a network becomes too optimised for a training dataset, to the detriment of performance for new data.

ReLu - A common activation function, see equation (4)

Sigmoid function - A common activation function, see equation (3)

Stepping schedules - Predetermined schemes of how to decrease learning rate during training, see section 1.1.2.2

Stochastic gradient descent (SGD) - A common method in neural network training

TensorFlow - An end-to-end open source platform for machine learning by Google

Weight - A node parameter w, used to calculate the activation of a node, see equation (5).