How to train a neural network

Amrit Talwar
Department of Mathematics
University of York
Heslington, York, UK

December 14, 2021

Abstract

This dissertation explains what neural network training is and how it works on a mathematical and practical level. We first define mathematical abstractions for the components of neural networks using basic linear algebra, then introduce the Gradient Descent method for finding optimal values for the weights and biases of a neural network in order to train the network to better predict input-output pairs of data. After this, we discuss the backpropagation technique, a method allowing us to calculate the partial derivatives used in gradient descent. Here, we use linear algebra and vector calculus to derive the four backpropagation equations for neural networks. This leads us to our final section where we implement a neural network using industry standard programming libraries and compare different neural network settings to see which gives us the optimal neural network training performance.

Contents

1 Introduction

The aim of this dissertation is to give a high level mathematical overview of neural networks and the common network training techniques and to show practical examples of neural network training. This work is aimed at people with a basic understanding of linear algebra and vector calculus, as well as some basic statistical knowledge. By the end of this dissertation, the reader should have a broad understanding of the common neural network training methodologies, as well as how they relate to the practical implementations of neural networks.

Firstly, in section 2 we will discussing the theoretical concept of neural networks and creating mathematical abstractions to represent neural networks. This will entail first defining the general purpose of neural networks, and then breaking down each part of a neural networks (e.g. network layers, network nodes, network weights, etc.) and using basic linear algebra to represent these working parts.

After constructing mathematical abstractions for the parts of a neural network, we will then see how we can use these in section 3 where we cover how neural networks are trained. Here, we demonstrate how some basic vector calculus and linear algebra is applied in a set of methods categorised as "gradient descent", an technique which aims to train our neural network by using the gradients of a "cost function", a function measuring the performance of the neural network. After covering the gradient descent method, we will then be looking at some of it's main variants and commenting on the theoretical and practical advantages of them. We will also be going over some of the popular optimization methods for gradient descent which aim to further increase the performance and efficiency of the standard gradient descent techniques.

In section 4, we will be going over "backpropagation", a key technique that allows us to calculate the gradients used in gradient descent. This section contains a lot of mathematical detail, deriving the four "backpropagation equations" that we can implement to calculate our gradients to be used in gradient descent.

Finally in section 5, we will implement a neural network using TensorFlow and cuDNN to recognize handwritten digit drawings. With this network, we will change various settings such as learning rate and batch size in a numerical investigation to figure out which settings can give us the best neural network training performance.

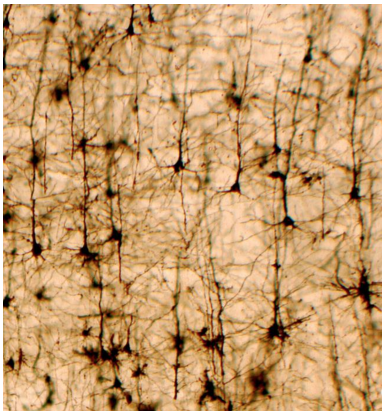
2 Neural Network Overview and Definitions

A neural network is a computational model inspired by the structure of neurons in the human brain. It consists of layers of nodes inspired by biological neurons, with the connections between neurons having an associated 'weight' value representing the strength of the connection. From a birdseye view, neural network can be thought of as a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ mapping input vectors X to prediction vectors Y , e.g. X is a vector containing the wind speed, temperature and humidity levels and Y is a prediction of if the weather will be sunny or cloudy the next day. This of course is quite a trivial example, but for an actual demonstration skip to section 5 where we implement a neural network to recognize handwritten digits.

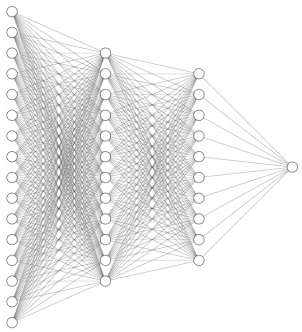
Neural networks seek to achieve accurate predictions and classifications of inputs, however they initially cannot do this right away. Much like when we are learning a new skill (e.g. a new language) we do not have a mastery of the skill right away. When we learn a skill, the strength of connections between certain neurons in our brain strengthen and weaken accordingly to create new neuron firing patterns that increase our proficiency at the skill. This is essentially what happens when neural networks learn, but in order to understand this on a mathematical level, we must first go over the mathematical representation of neural networks.

2.1 Network layers

Neural networks have a similar structure to neural networks found in the human brain. In the human brain, layers of neurons are formed for electrical activity to traverse over. Artificial neural networks follow this structure with layers artificial neurons where each neuron in one layer is connected to every other neuron in the next layer.



(a) Layered neurons in the brain [7]



(b) Typical layered structure of a neural network

Figure 1: A visual comparison of neuron layers in the brain and a typical neural network structure.

The reason we need multiple layers in a neural network is because it enables our neural network to learn and recognize more abstract and non-trivial features in our data. As our

neural networks learns, certain patterns are created between layers representing distinct features and patters in our input. If we have multiple layers, we can start identifying non-trivial patterns (e.g. patterns within patterns) that allow our neural network to better learn about relationships between input and output data.

Neural networks have at least three layers. The first layer is the **input layer**, where we pass in our input $X \epsilon \mathbb{R}^n$ where each entry x_i in the input vector X corresponds to the adjacent input node. The next layers are the **hidden layers** containing networks nodes (defined in the next subsection). The final layer is the **output layer**, giving us the entries y_i for our results or network prediction vector Y.

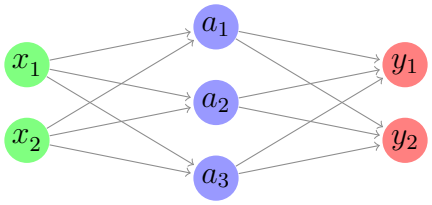


Figure 2: Simple network containing an input layer, hidden layer and output layer

For a network with L layers, we will index each layer with index $l = 0, 1, \ldots, L-1$

Definition 1 (Network layer) *A network layer at layer index l, denoted as $a^{(l)}$, is an n_l sized vector, where n_l is the number of nodes in the layer $a^{(l)}$.*

$$a^{(l)} = \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{pmatrix}$$

The entries of the vector, indexed by $i = 1, 2, \ldots, n_l$, correspond to the "activation values" of each node in layer $a^{(l)}$ (activation values will be defined in the next subsection, for now you can think of this as how strongly the neuron is firing with respect to neurons in the previous layer).

2.2 Weights (network connections)

Network layers are connected to each other as shown above in Figure 2. Each of these connections has an associated **weight** value. We can think of the weight on a connection as how much the current neuron should impact the activity in the next neuron it is connected to via that connection. Going back to our neural networks as a function analogy earlier, all the weights in our neural network can be thought of as coefficients in our massive neural network function.

The essence of training a neural network is ultimately down to figuring out what values our weights should be in order to get accurate predictions from the neural network. For now, we will not worry about how this calibration of weights is done, we will cover this in section 3 and 4.

Definition 2 (Network weights and weight matrix) *An individual network* ***weight*** *on a connection between node $a_i^{(l-1)}$ and $a_j^{(l)}$ is a real value denoted by $\omega_{j,i}^{(l)}$.*

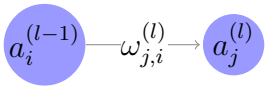


Figure 3: Weight on a connection between two network nodes

The ***weight matrix*** *between two layers $a^{(l-1)}$ and $a^{(l)}$, denoted by $W^{(l)}$, is an $(n_l \times n_{l-1})$ dimensional matrix where each entry in the matrix $\omega_{j,i}^{(l)}$ is a network weight.*

$$W^{(l)} = \begin{pmatrix} \omega_{1,1}^{(l)} & \omega_{1,2}^{(l)} & \cdots & \omega_{1,n_{l-1}}^{(l)} \\ \omega_{2,1}^{(l)} & \omega_{2,2}^{(l)} & \cdots & \omega_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n_l,1}^{(l)} & \omega_{n_l,2}^{(l)} & \cdots & \omega_{n_l,n_{l-1}}^{(l)} \end{pmatrix}$$

2.3 Network neurons

Neurons are the building blocks of neural networks. Neurons in the brain function by receiving electric activity from neurons in the previous layer, and if the activity is above a certain limit, the neuron activates and sends electrical activity to the next layer of neurons.

In neural networks nodes do something similar. The strength that the node fires at, called the **activation**, is a weighted sum of all activation values from nodes in the previous layer plus some **bias**, all passed through an **activation function**. Instead of the "fire or not" approach in brain neurons, it is common to allow the activation level a to be between a real valued number between 0 and 1 ($a \epsilon \mathbb{R}^{[0,1]}$). This is to allow our neural network to model any continuous real values functions who's values are not just binary (e.g. 0 or 1). Denote the activation value of node i in layer l as $a_i^{(l)}$.

As mentioned previously, nodes also have a **bias** b, which forces the activation to be at a minimum of b. This represents how much we want an individual node to be active irrespective of it's inputs (hence the name "bias"). Let the bias value of node i in layer l be denoted as $b_i^{(l)}$

Finally, we have the **activation function**, in which we pass in the weighted sum of the activation values in the previous layer plus the current node bias value. The purpose of this function is to model how "active" the node should be given the weighted activities

of the previous layer. It also serves to compress all inputs to outputs between 0 and 1 to make much of the following maths around neural networks easier.

There are many different activation functions, each with their unique pros and cons. Typical activation functions include **Sigmoid**, **ReLU** and **Softmax**. We will denote the activation function as function g from now on.

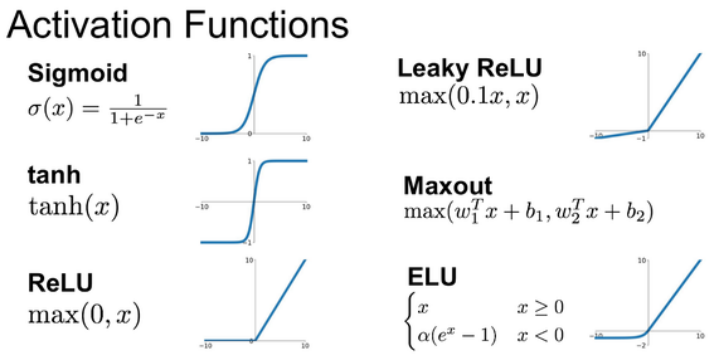


Figure 4: Common activation functions [15]

So, for node i in layer l, the activation value of this node is as follows:

$$z_i^{(l)} = \sum_{j=1}^{n_{l-1}} \omega_{j,i}^{(l)} a_i^{(l-1)} + b_i^{(l)}$$

$$a_i^{(l)} = g(z_i^{(l)})$$

or in matrix form:

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

$$a^{(l)} = g(z^{(l)})$$

As we see here, the activation of one neuron is effected by activations of neurons in the previous layers, which are effected be activations in the previous layers, etc. Going back to our motivation behind using multiple layers, stating that multiple layers allows us to model abstract features in our data, these equations now show why this is the case. Certain inputs will create certain activation patterns across the network due to the recursive nature of the network. If the network is deep enough, we can model many non-trivial features in the input.

So now we have mathematical representations for the parts of a neural network, but now we need a way to make our network learn patterns in the input data and adjust the weights and biases of the network accordingly to accurately model such patterns. To perform network learning, we use **gradient descent**.

3 Gradient descent

Gradient descent is the method which we use to adjust the weights and biases of our neural network to it to actually learn the patterns in the input data. There are three main types of gradient descent which we will be discussing, **batch gradient descent**, **stochastic gradient descent**, and **mini-batch gradient descent**. In practice, mini-batch gradient descent is preferred for big data and is most commonly used, but it is important to understand stochastic gradient descent and batch gradient descent to appreciate why this is the case.

At the end of this section, we will be going over some of the commonly used gradient descent optimization techniques used to circumvent some of the pitfalls of our three vanilla gradient descent techniques.

3.1 Measuring network performance: Cost functions

To measure how well the neural network performs on our training set (the data used to train our neural network), we use a **cost function** [11] which returns a real value representing how wrong the network predictions are from the actual/ expected predictions.

Definition 3 (Cost function) *Let $\Theta \, \epsilon \, \mathbb{R}^P$ be a vector containing all of our weights and biases (referred to as "parameters"):*

$$\Theta = \begin{pmatrix} \omega_{1,1}^{(1)} \\ \omega_{2,1}^{(1)} \\ \vdots \\ \omega_{n_L,n_{L-1}}^{(L)} \\ \vdots \\ b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_{n_L}^{(L)} \end{pmatrix}$$

The cost function of our neural network is a function $C : \mathbb{R}^P \longrightarrow \mathbb{R}$ which is smooth, differentiable and greater than 0 at every point.

Let X be our training data consisting of input/ output pairs ($\{(x_1, y_1), (x_2, y_2), \ldots, (x_T, y_T)\}$).

The cost function has the form:

$$C(X, \Theta) = \frac{1}{T} \sum_t C_t(x_t, y_t, \Theta) > 0$$

Where $C_t : \mathbb{R}^P \longrightarrow \mathbb{R}$ is the cost function incurred by an individual training example (which also satisfies the smooth, differentiable and greater than 0 constraints) .

Typical examples of C_t include mean squared error and categorical cross entropy. **For the rest of this dissertation, we will use the common abbreviations of $C(X, \Theta) = C(\Theta)$ and $C_t(x_t, y_t, \Theta)) = C_t(\Theta)$.**

Cost functions essentially serve as a way for neural networks to receive feedback on how bad they are doing. Much like when we learn, it is necessary to know how what mistakes we have made so we can adjust our actions to better minimize our mistakes the next time around.

When we are getting a neural network to learn, we are looking to minimize this cost function with respect to all the weights and biases to achieve a well performing network. The intuition is that by minimizing the average loss incurred across all training examples, we reach an optimal Θ which on average gives us a low cost per input. However, typical neural networks have upwards of billions of total weights and biases, and in addition with the compositional nature of neural networks (neuron activations are functions of previous activations) it means that finding the minimas of the cost function simply cannot be done analytically for typical a neural networks. We need to instead use an iterative approach for minimizing the cost function.

3.2 Gradient descent for minimizing the cost function

To minimize $C(\Theta)$, we use a **gradient descent** [9] method. The standard version of gradient descent is called **batch gradient descent** [22]. It works by updating Θ incrementally to minimize the cost function. The gradient descent iterations are as follows:

$$\Theta^{(i+1)} = \Theta^{(i)} - \gamma \nabla C(\Theta^{(i)})$$

$$\nabla C(\Theta^{(i)}) = \frac{1}{T} \sum_t \nabla C_t(\Theta^{(i)})$$

Where $\Theta^{(i)}$ is the vector Θ on the i^{th} iteration of gradient descent and $\gamma \, \epsilon \, \mathbb{R}$ is the **learning rate**.

The reason for the name "batch gradient descent" is due to the fact that it computes the gradient of the cost function with respect to all the parameters in Θ for the **entire** training set (or "batch") to complete one iteration (or step) in gradient descent. Here we are moving down toward a minima of the cost function in the direction of the average downhill gradient in all directions (thus lowering the cost function per iteration).

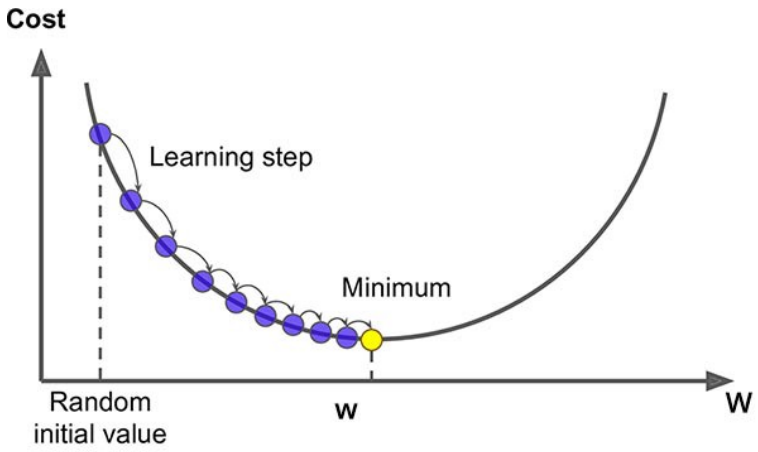


Figure 5: Gradient descent steps towards the minima of a cost function [24]

Although we are moving in the direction of steepest descent (w.r.t all parameters), we need to be careful as to how we choose our learning rate γ. Too high and we can overshoot the optimal values, too low and we may moving towards the minima too slowly. In practice, multiple test runs of gradient descent are ran to iteratively find the best value for γ.

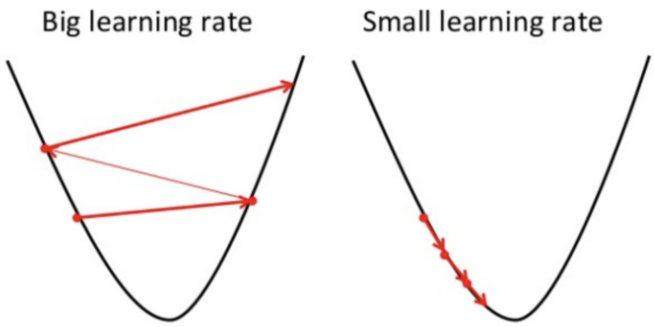


Figure 6: Example of how having a learning rate too big can cause gradient descent to diverge, where the red arrow is the path of gradient descent [2]

The **convergence theorem for gradient descent** [23] tells us that gradient descent will converge to a minimum when γ is sufficiently small:

Theorem 1 (Convergence theorem for gradient descent) *Suppose the function* $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ *is convex and differentiable, and that its gradient is Lipschitz continuous with constant* $L > 0$, *i.e. we have* $\|\nabla f(x) - \nabla f(y)\|_2 \leq L \|x - y\|_2$ *for any x,y. Then if we run gradient descent for k iterations with a fixed step size* $t \leq 1/L$, *it will yield a solution* $f^{(k)}$ *which satisfies:*

$$f(x^{(k)}) - f(x^{(*)}) \leq \frac{\left\|x^{(0))} - x^{(*)}\right\|_2^2}{2tk} \tag{1}$$

where $f(x^{(*)})$ is the optimal value.

Although this theorem is for convex functions, the actual surfaces of our cost functions are most of the time highly dimensional non convex surfaces with multiple minima. This is a problem as gradient descent can converge to any one of these local minima and get stuck, failing to reach a more optimal minima which better reduces the cost function.

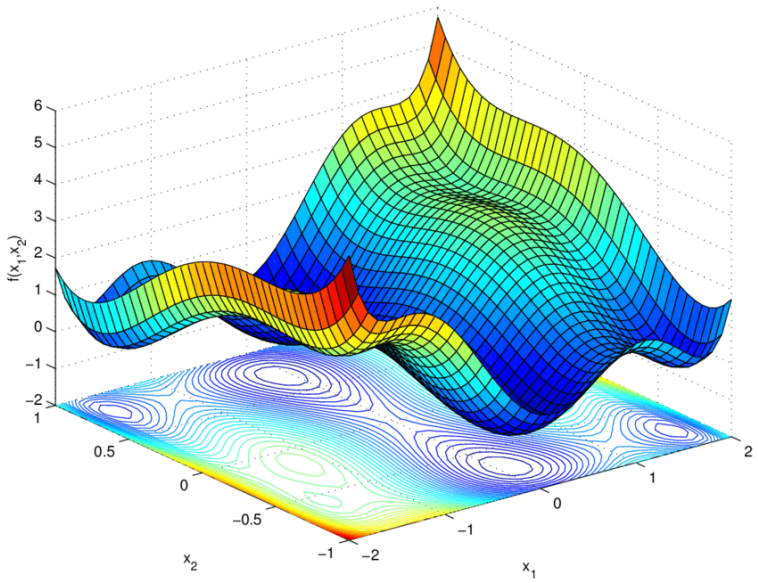


Figure 7: An example of a function with multiple minima which gradient descent can get stuck on [16]

This follows into one of our two issues with batch gradient descent:

1. Batch gradient descent can get stuck in a local minima which is still very sub-optimal for minimizing $C(\Theta)$

2. Batch gradient descent is extremely computationally expensive for big data

For typical neural networks, the cost function is actually non-convex with multiple minima, many of which are sub-optimal/ very far away from the global minimum. This obviously leads us to getting stuck in local minimas quite frequently. As for the second point of computational inefficiency, we need to scan through the **entire** training set **and** calculate all the gradients of the cost function w.r.t each weight and bias, just to perform one step of gradient descent. In practice, data sets can be terabytes large and simply cannot be stored in computer memory to be scanned over. This also means that whenever we want to learn on a new set of data, we must wait until we have gathered enough samples to create a batch from that new data set, which could mean we are having to wait days at a time between each execution of gradient descent.

We clearly need a new method which circumvents the memory issue and the multiple minima problem for large scale data. **Stochastic gradient descent** is the next evolution of gradient descent which deals with these problems.

3.3 Stochastic gradient descent

Stochastic gradient descent (**SGD**) [5] circumvents the above issues by modifying the gradient descent iteration. It instead uses a single training example to calculate the gradient per iteration rather than using all the training examples:

$$\Theta^{(i+1)} = \Theta^{(i)} - \gamma \nabla C_t(\Theta^{(i)})$$

Where t is a uniformly randomly chosen training example.

This introduces some variance to our gradient descent steps as not every step is guaranteed to decrease the cost function (as we are only moving in the direction that is optimizing for one of the training examples, not all). The reason this still works is because the gradient for a single training example is an unbiased estimator for the gradient of the whole cost function $\nabla C(\Theta_i)$ (i.e. $\mathbb{E}[\nabla C_t(\Theta_i)] = \nabla C(\Theta_i)$):

$$\begin{aligned} \mathbb{E}[\nabla C_t(\Theta)] &= \sum_{k=0}^{T-1} (\nabla C_k(\Theta) \times \mathbb{P}(t = k)) \\ &= \frac{1}{T} \sum_{k=0}^{T-1} \nabla C_k(\Theta) \\ &= \nabla C(\Theta) \end{aligned} \tag{2}$$

By the law of large numbers, this means that for a large amount of iterations of SGD, the gradient $\nabla C_t(\Theta^{(i)})$ will approximate the true gradient $\nabla C(\Theta)$ very well. In essence, we are getting a fast approximation of the true gradient which we can use per SGD iteration. This solves our multiple minima issue as SGD has the ability to escape local minima due to the noisy pathing.

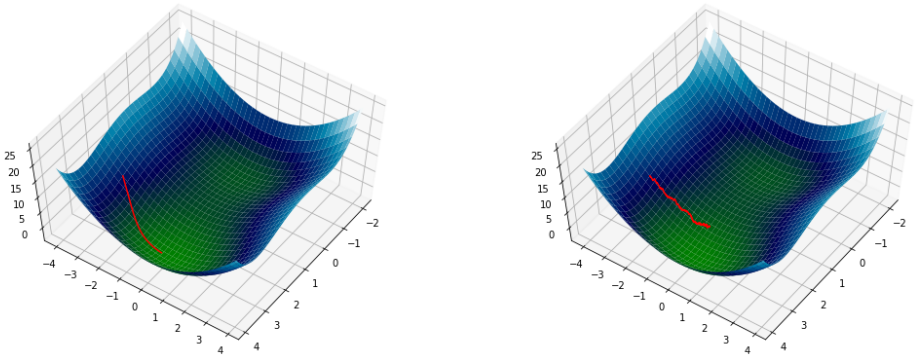


Figure 8: Linear batch gradient descent path VS stochastic SGD descent path [4]

SGD also solves the memory issue as now we only have to scan through one training example per iteration, essentially making this T times faster than batch gradient descent (as we are only computing one gradient per iteration rather than T gradients per iteration).

Although SGD is fast and can escape local minima, it has very high variance and is very sensitive to outliers in the training data. This can actually slow down our convergence to the global optimum as when close to the global optimum SGD will just dance around it, however in practice this can sometimes be desirable as we do not want to get an overly optimal Θ to avoid over-fitting our model to our training data.

In addition, modern machine learning is done with massive amounts of compute power that takes advantage of parallel computing [3], a technique where multiple parts of a calculation or computation can be done in parallel of each other with the results of each computation being combined at the end. SGD does not really take advantage of this capability as we only compute the gradient for one training example per step. We want a new technique that can take advantage of the technology we have available. An immediate thought is to just run batch gradient descent in parallel (calculate the gradients for each training example in parallel to each other and sum up the results), which makes sense, but is unfortunately still very expensive to do as you would need many many CPU cores to distribute terabytes worth of calculations over. This leads to a second intuition of instead partitioning the training data into smaller batches which are much easier to computer in parallel. These mini-batches will approximate our full batch and are small enough to be computed efficiently. This technique is unsurprisingly called "**mini-batch gradient descent**".

3.4 Mini-batch gradient descent

Mini-batch gradient descent [13] is similar to batch gradient descent, however for each iteration of gradient descent we divide our training data into batches of size n_B and update Θ according to the average gradient of examples in this batch:

$$\Theta^{(i+1)} = \Theta^{(i)} - \gamma \nabla C^{(B)}(\Theta^{(i)})$$

$$C^{(B)}(\Theta^{(i)}) = \frac{1}{B} \sum_{t=0}^{B-1} C_t(\Theta^{(i)})$$

Immediately we can see that this is far less vulnerable to outliers as we are calculating the average gradient across the whole batch, which is good if we expect our training data to contains many outliers. In addition, assuming that our training data is i.i.d. (independent and identically distributed), then we can clearly show how mini-batch gradient descent reduces the variance of the learning gradient. For a batch of size B we have:

$$
\begin{aligned}
Var(\nabla C^{(B)}(\Theta)) &= Var(\frac{1}{B}\sum_{t=0}^{B-1}\nabla C_t(\Theta)) \\
&= \frac{1}{B^2}Var(\sum_{t=0}^{B-1}\nabla C_t(\Theta)) \\
&= \frac{1}{B^2}\sum_{t=0}^{B-1}Var(\nabla C_t(\Theta)) \\
&= \frac{1}{B^2} \times BVar(\nabla C_t(\Theta)) \\
&= \frac{1}{B}Var(\nabla C_t(\Theta))
\end{aligned}
\tag{3}
$$

As we can see, as the batch size B increases, the variance of our learning gradient $\nabla C^{(B)}(\Theta)$ decreases. This means that we can decide how much variance we want our training steps to have by either increasing or decreasing the batch size. This gives us a nice balance between the stableness of batch gradient descent and the highly variant SGD.

As mentioned previously at the end of the SGD section, mini-batch gradient descent also allows us to take advantage of compute parallelization as the batches can be made small enough to parallelize. A typical batch size is around 100-1000, but you can increase your batch-sizes if you have more compute power to distribute your gradient calculations over.

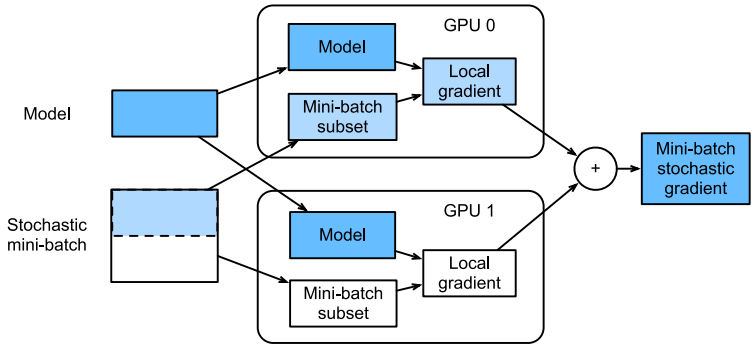


Figure 9: Basic mini-batch parallelization architecture showing the mini-batch gradient calculations being distributed across two GPU's [17]

As well as being able to take advantage of parallel computing, mini-batch gradient descent also means that we can conduct **online learning** [14]. Online learning is when we perform gradient descent based on an incoming stream of small batches of training examples rather then waiting to accumulate one massive batch to conduct batch gradient descent. This is very common practice with big tech companies. Facebook for instance have many data pipelines engineered to get real time data from users using their services.

This data is streamed to their machine learning services/ data centers in real time to conduct online learning.

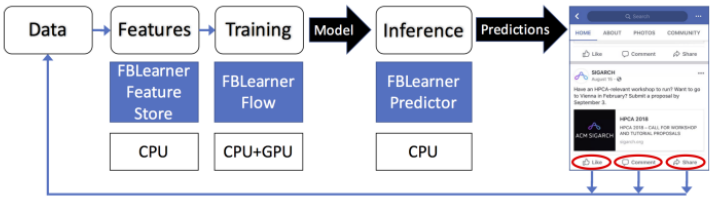


Figure 10: One of Facebook's data pipelines to stream real-time user data from the homepage into online machine learning algorithms [12]

3.5 Gradient descent optimization algorithms

Although SGD and mini-batch learning are widely used in practice, they are often supplemented with additional optimizations to further increase their efficiency and speed.

3.5.1 Momentum

Momentum [21] is a technique used to deal with the oscillations and fluctuations of SGD. It does so by adding a factor of the previous update gradient to the current update gradient:

$$v_i = \gamma \nabla C_t(\Theta^{(i)}) + m v_{i-1} \tag{4}$$

$$\Theta^{(i+1)} = \Theta^{(i)} - v_i \tag{5}$$

This works like momentum in classical physics, the momentum term adds the direction of the previous update vector to the direction of the current update vector. This amplifies movement along a similar or shared direction and softens oscillations in irrelevant directions (much like traditional momentum in real life), resulting in SGD converging quicker to an optimum. Momentum also helps SGD escape local minima even better as the gradient steps can have increased momentum behind them to escape.

Figure 11: SGD without momentum VS SGD with momentum [1]

Although momentum helps SGD converge faster to a minimum, this technique means that we often overshoot past the minimum. This is because once we have reached the minimum the momentum term is pretty high (as it contains a cumulative sum of all previous momentum terms). The algorithm has no idea that it should slow down when it approaches a minimum. This was solved by Yurii Nesterov in his 1983 paper "A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$" [19].

3.5.2 Nesterov accelerated gradient

The Nesterov accelerated gradient algorithm (or **NAG**) modifies the momentum algorithm. In the momentum algorithm, we are taking a step in the steepest direction amplified by the previous momentum term, but in NAG we essentially look ahead by stepping in the direction of the previous accumulated gradient and then measuring the gradient where it ends up and making a correction:

$$v_i = \gamma \nabla C_t(\Theta^{(i)} - m v_{i-1}) + m v_{i-1} \tag{6}$$

$$\Theta^{(i+1)} = \Theta^{(i)} - v_i \tag{7}$$

The key here is the "look-ahead" step $\nabla C_t(\Theta^{(i)} - m v_{i-1})$. Instead of taking a step in the direction of the steepest slope amplified by our previous momentum, we instead first look at the gradient at where we would end up if we follow our momentum vector, then make a step according to this gradient before adding our built up momentum $m v_{i-1}$. This gives NAG as sense of the gradients ahead of itself so it knows to slow down. Think of driving down a hill shaped like a parabola. When reaching the bottom, you know visually to slow down as you can see that the hill is flattening ahead of you. This is essentially what NAG is doing.

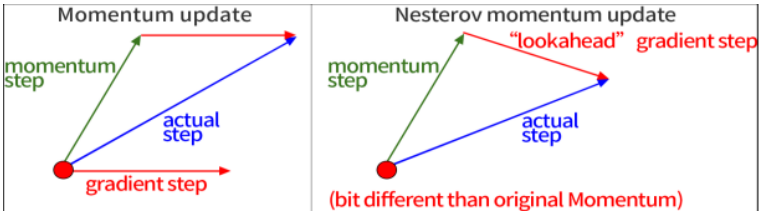


Figure 12: NAG step vs normal momentum step [13]

Although momentum and NAG are used in stochastic gradient descent, they can also be applied to mini-batch gradient descent by simply replacing C_t with $C^{(B)}$. This is makes momentum methods very beneficial for if we want to use compute parallelization, but don't have much compute power. Momentum methods help ease the oscillations and noise associated with smaller batch sizes, meaning we can get away with computing smaller batches with limited computer power and resources.

3.5.3 Adagrad

Another common problem with gradient descent is choosing an optimal learning rate γ. With all our previous gradient descent methods, we have to manually find the optimal learning rate ourselves. We don't want the learning rate to be too big otherwise we will overshoot when the gradient is steep, but we don't want the learning rate to be too small because it would massively slow down learning along shallow services and close to the minimum. It would be ideal if we could adapt and decay our learning rate to be slower on steeper gradients and faster on shallower gradients. **Adagrad** [6] (short for adaptive gradient) does just this.

Adagrad allows the learning rate to update based on the parameters. It performs smaller gradient descent steps for parameters with a historically steep direction in the loss function space and larger steps for parameters with a historically shallower/ more flat direction in the loss function space. It does so by using the sum of past squared gradients G_i in the following learning rule:

$$g_i = \nabla C_t(\Theta^{(i)}) \tag{8}$$

$$G_i = \sum_{j=0}^{i} g_j g_j^\top \tag{9}$$

$$\Theta^{(i+1)} = \Theta^{(i)} - \frac{\gamma}{\sqrt{\varepsilon I + diag(G_i)}} \cdot g_i \tag{10}$$

Where ε is some very small constant ≈ 0 that prevents us from dividing by 0.

A big downside with Adagrad is that the sum of squared gradients in the diagonal elements of $diag(G_i)$ is monotonically increasing, thus the denominator $\sqrt{\varepsilon I + diag(G_i)}$ is constantly increasing. This means that our gradient descent steps $\frac{\gamma}{\sqrt{\varepsilon I + diag(G_i)}} \cdot g_i$ will eventually become small and smaller to the point where they become so small that virtually stop converging to the minimum of the cost function. The following algorithm aims to resolve this issue.

3.5.4 RMSprop

RMSprop (short for **root mean squared propagation**) is an unpublished method proposed by Geoff Hinton in his course "Neural networks for machine learning" [13] .It seeks to combat the vanishing learning rate problem of Adagrad by instead using an exponentially weighted moving average of past gradients:

$$E[g^2]_i = \rho E[g^2]_{i-1} + (1-\rho) g_i^2 \tag{11}$$

$$\Theta^{(i+1)} = \Theta^{(i)} - \frac{\gamma}{\sqrt{\varepsilon I + diag(E[g^2]_i)}} \cdot g_i \tag{12}$$

Where ρ is a real value between 0 and 1.

RMSprop ensures that the denominator doesn't just monotonically as our denominator $\sqrt{\varepsilon I + diag(E[g^2]_i)}$ contains a moving average of gradients rather than the sum of past squared gradients, and the average can go up or down whilst the sum of squared gradients is monotonically increasing.

3.5.5 Adam

Adam, short for **Adaptive Moment Estimation** combines the ideas of momentum and RMSprop. It uses the exponential moving averages of the gradient (similar to momentum)

and the squared gradient (similar to RMSProp). The first moving average is an estimate of the mean of the gradients and the second moving average is an estimate of the uncentered variance:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{14}$$

Since the moving averages are initialized as vectors of 0's (causing early moment values to be biased towards 0) a bias correction is introduced to m_t and v_t to more accurately calculate the values of the moving averages:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{15}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{16}$$

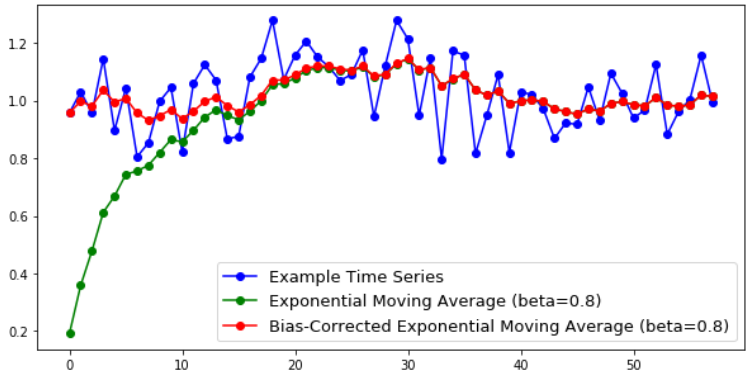


Figure 13: Effects of 0 bias and bias correction on an exponential moving average

This gives us our update rule:

$$\Theta^{(i+1)} = \Theta^{(i)} - \frac{\gamma}{\sqrt{\hat{v}_t} + \varepsilon I} \hat{m}_t \tag{17}$$

The intuition behind Adam is two fold. Firstly, there is our 1st moment term $\hat{m}_t$ which acts like momentum, giving us momentum like properties. Secondly, there is our term $\hat{v}_t$. As this increases, it essentially means that we are more uncertain whether our current gradients approximate the average gradient well, so hence we should reduce our step size to accommodate for this uncertainty. The authors of Adam had this to say about the ratio of to in their original paper "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION":

"With a slight abuse of terminology, we will call the ratio $\hat{m}_t / \sqrt{\hat{v}_t}$ the signal-to-noise ratio (SNR). With a smaller SNR the effective stepsize δt will be closer to zero. This is a desirable property, since a smaller SNR means that there is greater uncertainty about

whether the direction of $\hat{m}_t$ corresponds to the direction of the true gradient. For example, the SNR value typically becomes closer to 0 *towards an optimum, leading to smaller effective steps in parameter space: a form of automatic annealing."*

3.6 Gradient descent summary

In summary, gradient descent encompasses techniques for tuning the weights and biases of the neural network to values which optimize the neural network cost function. There are several techniques for this, each of which has its pros and cons (some allow us to take advantage of certain computational technology whilst increasing the randomness in gradient descent whilst others provide a highly accurate convergence at the cost of massive compute power). There are also further gradient descent optimization algorithms to speed up our basic gradient descent algorithms which we can use.

One key point that hasn't been mentioned in this section is the actual gradients used in gradient descent themselves. Obviously the entries of the vector $\nabla C(\Theta)$ are the partial derivatives of the cost function w.r.t. each weight and bias, but how do we calculate them? It turns out that this too is quite complicated due to the compositional nature of neural networks. However, in order to perform any version of gradient descent, we actually need to be able to compute these gradients to perform gradient descent steps. Fortunately, there exists an efficient way of computing these gradients. This method of computing the cost function gradients in called **Backpropagation**.

4 Backpropagation

Backpropagation is a technique for finding the derivatives of complex nested functions, which a neural network essentially is. The essence of backpropagation is to start from the output layer of the network and use the chain rule from calculus to propagate backwards through the network to compute all of the partial derivatives in $\nabla C(\Theta)$.

Here is a reminder for the notation we defined in chapter 1 and chapter 2 which will use again in this section:

- $\omega_{j,i}^{(k)}$ - Weight on connection of node i in layer $k-1$ to node j in layer k
- $z_i^{(k)}$ - Linear combination of weights of incoming connections and activations of previous layer (plus bias term) for node i in layer k
- $a_i^{(k)}$ - Activation of node i in layer k
- g - Activation function (e.g. ReLU, softmax)
- C - Cost function of neural network (e.g. MSE)

4.1 Derivative w.r.t weights

The derivation of the backpropagation equations starts with applying the chain rule to the cost function:

$$\frac{\partial C}{\partial \omega_{j,i}^{(k)}} = \frac{\partial C}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial \omega_{j,i}^{(k)}} \tag{18}$$

The first partial derivative is called the **error** of node j in layer k. The error is denoted by:

$$\delta_j^{(k)} \equiv \frac{\partial C}{\partial z_j^{(k)}} \tag{19}$$

The reason for the name "error" is because this partial derivative tells us how much the cost function is effected by the weighted sum in the node, i.e. how much error in the cost function is incurred by changes in this weighted sum in the node.

$\frac{\partial z_j^{(k)}}{\partial \omega_{j,i}^{(k)}}$ can be calculated from the equation for $z_j^{(k)}$ in section 2.3:

$$\frac{\partial z_j^{(k)}}{\partial \omega_{j,i}^{(k)}} = \frac{\partial}{\partial \omega_{j,i}^{(k)}} [(\sum_{n=1}^{n_{k-1}} \omega_{j,n}^{(k)} a_n^{(k-1)}) + b_j^{(k)}] = a_i^{(k-1)} \tag{20}$$

This means that the change in the weighted sum of a node w.r.t. a weight coming into the node equals the activation of the origin node (the node that the weight is coming from). This makes sense as when the activation of a node is high, it will heighten the weighted sum of the next neuron it is connected to.

So, now we can express our derivative $\frac{\partial C}{\partial \omega_{j,i}^{(k)}}$ as:

$$\frac{\partial C}{\partial \omega_{j,i}^{(k)}} = \delta_j^{(k)} a_i^{(k-1)} \tag{21}$$

This is great, we now have an equation for the derivative of the cost function w.r.t any weight in the network, but we still need to calculate what $\delta_j^{(k)}$ actually is. For the output layer L, we can apply the chain rule to give:

$$\begin{aligned} \delta_j^{(L)} &= \frac{\partial C}{\partial z_j^{(L)}} \\ &= \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \\ &= \frac{\partial C}{\partial a_j^{(L)}} g'(z_j^{(L)}) \end{aligned} \tag{22}$$

Since C is a function of $a_j^{(L)}$ and associated prediction Y_j (i.e. MSE, softmax), $\partial C/\partial a_j^{(L)}$ can be directly calculated. $g^{'}(z_j^{(L)})$ can also be directly calculated as it is just the derivative of the activation function g. Now, for the hidden layers, we can apply the chain rule for multivariate functions to get:

$$\begin{aligned}\delta_j^{(l)} &= \frac{\partial C}{\partial z_j^{(l)}} \\ &= \sum_{n=1}^{n_{l+1}} \frac{\partial C}{\partial z_n^{(l+1)}} \frac{\partial z_n^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_{n=1}^{n_{l+1}} \delta_n^{(l+1)} \frac{\partial z_n^{(l+1)}}{\partial z_j^{(l)}}\end{aligned} \tag{23}$$

For $\partial z_n^{(l+1)}/\partial z_j^{(l)}$, we have:

$$\begin{aligned}\frac{\partial z_n^{(l+1)}}{\partial z_j^{(l)}} &= \frac{\partial}{\partial z_j^{(l)}}[(\sum_{i=1}^{n_l} \omega_{n,i}^{(l+1)} a_i^{(l)}) + b_n^{(l+1)}] \\ &= \omega_{n,j}^{(l+1)} g^{'}(z_j^{(l)})\end{aligned} \tag{24}$$

So finally, we get:

$$\begin{aligned}\delta_j^{(l)} &= \sum_{n=1}^{n_{l+1}} \delta_n^{(l+1)} \omega_{n,j}^{(l+1)} g^{'}(z_j^{(l)}) \\ &= g^{'}(z_j^{(l)}) \sum_{n=1}^{n_{l+1}} \delta_n^{(l+1)} \omega_{n,j}^{(l+1)}\end{aligned} \tag{25}$$

4.2 Derivative w.r.t bias

For our partial derivatives w.r.t bias, we also apply the chain rule in a similar way as in (13):

$$\frac{\partial C}{\partial b_j^{(k)}} = \frac{\partial C}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial b_j^{(k)}} \tag{26}$$

For $\partial z_j^{(k)}/\partial b_j^{(k)}$, we have:

$$\begin{aligned}\frac{\partial z_j^{(k)}}{\partial b_j^{(k)}} &= \frac{\partial}{\partial b_j^{(k)}}[(\sum_{n=1}^{n_{k-1}} \omega_{j,n}^{(k)} a_n^{(k-1)}) + b_j^{(k)}] \\ &= 1\end{aligned} \tag{27}$$

So, as a result we have:

$$\frac{\partial C}{\partial b_j^{(k)}} = \frac{\partial C}{\partial z_j^{(k)}} = \delta_j^{(k)} \tag{28}$$

4.3 Backpropagation equations summary

In summary, we have the four following equations called the **backpropagation equations**:

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} g'(z_j^{(L)})$$

$$\delta_j^{(l)} = g'(z_j^{(l)}) \sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \omega_{i,j}^{(l+1)}$$

$$\frac{\partial C}{\partial \omega_{j,i}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

When looking at the backpropagation equations, its now clear as to where the technique gets it's name. We see that $\delta_j^{(l)}$ is dependent on $\delta_j^{(l+1)}$, i.e. we start with directly calculating $\delta_j^{(L)}$ and work backwards through the network to calculate the errors in the previous layers so we can calculate the derivatives for the weights and the biases.

5 Neural networks in practice

In this section, we will be first introducing the common software and programming libraries used to implement and train neural networks. We will be using these throughout this section and all subsequent results and graphs will be produced with these tools.

We will then be implementing a neural network in TensorFlow and training it to recognize images of handwritten digits using the MNIST dataset [18] for our training and test data. The plan is to first create our neural network, and then carry out the following investigations:

1. How does batch size effect accuracy and gradient descent convergence, and how much of a difference does batch size make in training time?

2. How does learning rate effect accuracy and gradient descent convergence. Particularly, how can a learning rate that is too high show divergence in gradient descent (and reductions in accuracy)?

3. How well do the different gradient descent optimizers covered in chapter 3 perform compared to each other?

For all of the investigations, the neural network will be trained over 30 epochs. An epoch refers to a full iteration through all the examples in the entire training set during gradient descent (note that if we are using minibatch, going through all of the batches still counts as an epoch).

5.1 Brief introduction to TensorFlow and cuDNN

TensorFlow (https://www.tensorflow.org/) is an open source programming library made by Google. It allows you to very simply define a neural network neural network structure and train the weights of the network using some simple code. TensorFlow automatically applies the common learning techniques of gradient descent and backpropagation without the user having to explicitly define these by hand. It also allows you to select and customize different parameters such as learning rate and gradient descent optimizer, which will be useful for us when it comes to showing the effects of these parameters. Nvidia cuDNN is a programming library also for creating and training neural networks

(https://developer.nvidia.com/cudnn). It is what technologies like TensorFlow are based upon. Nvidia cuDNN takes advantage of a special piece of hardware in computer called the GPU (graphical processing unit) to optimize many neural network implementation and training steps. cuDNN allows us to utilize the compute parallelization mentioned in Figure 9 to speed up neural network training.

5.2 MNIST dataset and our network architecture

The MNIST handrwitten digit data set consists of 60,000 training examples and 10,000 test examples of 28×28 255 grayscale (each pixel has an integer value from 0 to 255) hand written digit images.

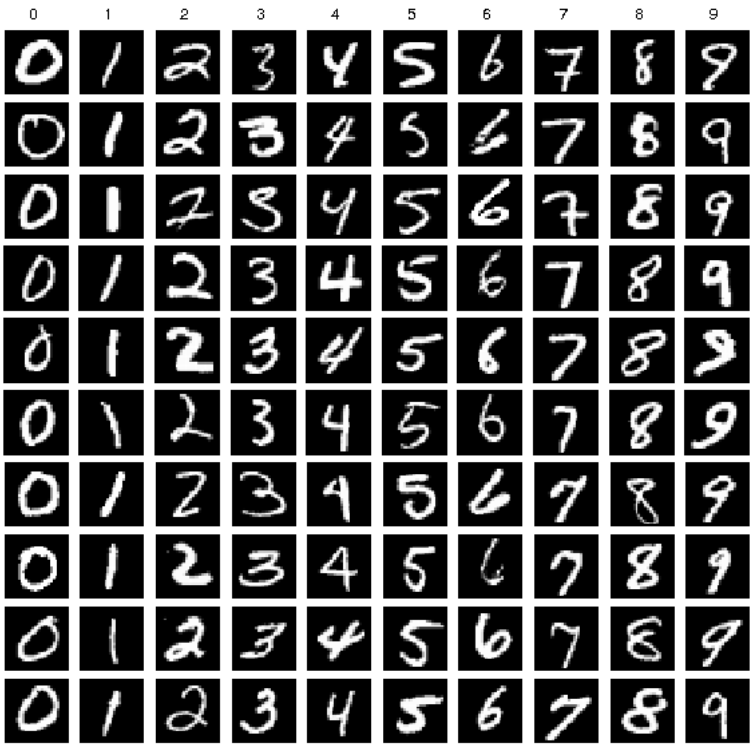


Figure 14: Images from the MNIST handwritten digit training data set [15]

Our neural network network architecture consists of the following layers and activation functions:

1. Input layer $\epsilon \mathbb{R}^{28^2}$
2. Hidden layer $\epsilon \mathbb{R}^{250}$ (ReLU)
3. Hidden layer $\epsilon \mathbb{R}^{250}$ (ReLU)
4. Output layer $\epsilon \mathbb{R}^{10}$ (Softmax)

5.3 Effects of batch size

In Chapter 3, we covered mini-batch gradient descent and talked about how it uses batches of data to perform gradient descent steps. Here, we will be comparing the effects of batch sizes on neural network performance.

Since we know that mini-batch gradient descent performs a step per batch, we can hypothesize that a lower batch size will give us lower network loss and higher accuracy as mini-batch gradient descent will perform more gradient descent steps due to having more batches (or fractions) of data. However, we can expect the higher batch sizes to give us a quicker training run time as we are making more use of compute parallelization with our higher batch sizes (as seen in Figure 11).

When using the batch sizes of $60, 600, 6000, 60000$, we get the following results:

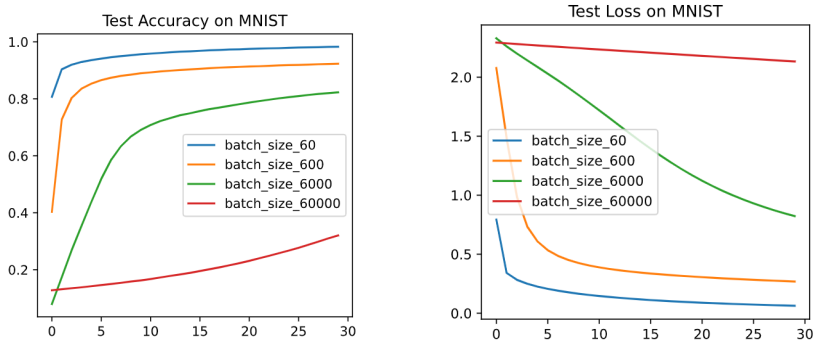


Figure 15: Test accuracy and loss for different batch sizes during training

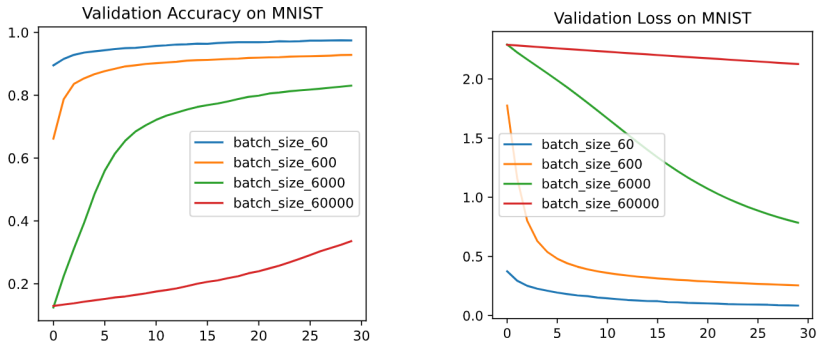


Figure 16: Validation accuracy and loss for different batch sizes during training

Note that the epochs are indexed 0 to 29, so the values at epoch 0 are the values after the first epoch, not the starting values.

Batch size	Accuracy after first epoch	Accuracy after last epoch	Loss after first epoch	Loss after last epoch
60	0.9	0.98	0.81	0.009
600	0.65	0.91	1.79	0.4
60000	0.052	0.82	2.43	0.97
600000	0.054	0.31	2.4	2.33

Batch size	Training time (seconds)
60	119
600	35
60000	5.12
600000	2.2

As we can see, our lower batch sizes result in higher accuracies and lower losses throughout all the training epochs. Initially after the first epoch, our smallest batch size gives a loss of just 0.81, whilst our largest batch size of 600000 (the full training data) gives us a much larger loss of 2.4 (almost 3 times the loss of batch size 60). As we observe in the table, this trend of increasing loss and decreasing accuracy occurs as we increase the batch size. These findings back up the claim that lower batch sizes give us

greater accuracies and lower losses due to more gradient descent steps being executed. On the flip side however, we see that the higher batch sizes seemingly give us extremely fast run times, with the maximal batch size giving us 2.2 seconds, and the lowest batch size giving us just under 2 minutes due the use/ non use of compute parallelization discussed in section 3.5.

These findings don't necessarily point to an ideal batch size. Ideally, you should keep the batch size low to increase accuracy and decrease loss more effectively. However, there may be some situations and applications where we don't mind having a slightly worse network with a way faster training time. In these cases, increasing batch size can get you what you want. In addition, organising and managing the compute hardware needed for higher batch size parralelization may also be an extra burden for whatever company/ organisation is running these models, which is a further argument against excessively high batch size.

In summary, a lower batch size seems more ideal for training a neural network due to better accuracies and losses and the lower logistical overhead of having to manage more compute hardware.

5.4 Effects of learning rate

As mentioned in the Convergence Theorem for Gradient Descent, we know that the learning rate must be below a certain value in order for gradient descent to converge. We also know that if the learning rate is too high, we can get divergent gradient descent. In this practical example, we are using the learning rates $1, 0.1, 0.01, 0.001$ to demonstrate the effects of learning rate.

The theory tells us that for the higher learning rate of 1, gradient descent should struggle to converge due to potential overshooting of the minima, whilst the smaller learning rates should converge to an optimal value. When training our neural network with these learning rates, we get the following results:

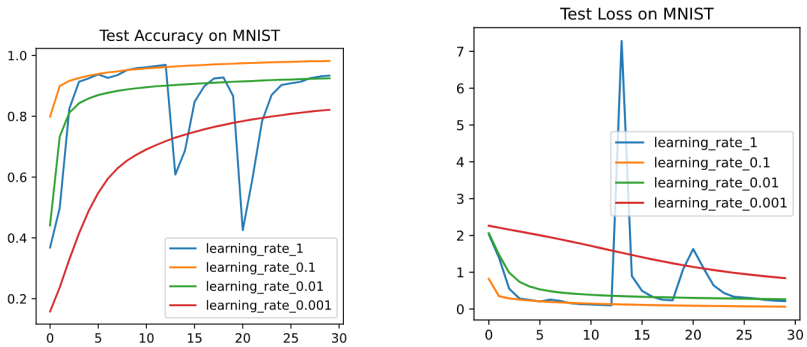


Figure 17: Test accuracy and loss for different learning rates during training

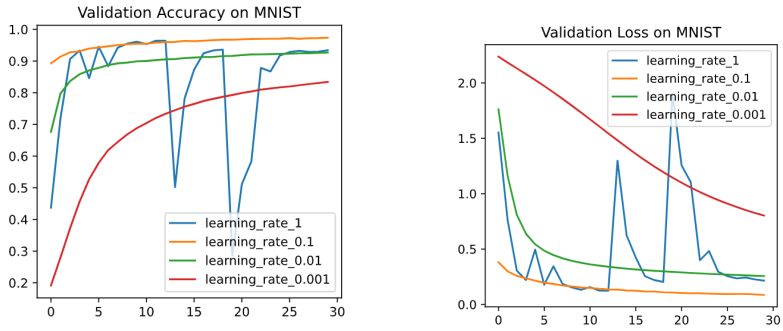


Figure 18: Validation accuracy and loss for different learning rates during training

Learning rate	Accuracy after first epoch	Accuracy after last epoch	Loss after first epoch	Loss after last epoch
1	0.44	0.9	1.53	0.26
0.1	0.89	0.97	0.44	0.13
0.01	0.68	0.9	1.79	0.25
0.001	0.2	0.82	2.44	0.8

For learning rate 1, we see that initially, we seem to converge to an optima very fast, reaching a test accuracy of over 0.9 and a test loss of under 0.25 by the 12th epoch, but then as we start to converge, we overshoot past the minima each time we get close to it as observed by the spikes in accuracies and losses around epoch 13 and epoch 18. This is due to our gradient descent steps being too big towards the minima and causing overshooting in gradient descent, as demonstrated in Figure 5 in section 3.3.

For the other learning rates, we don't observe any divergence. Learning rate 0.1 seems to converge that fastest, indicating that 0.1 is an ideal learning rate for gradient descent with our current neural network architecture and dataset. Although 0.1 seems to be the best learning rate with a final accuracy of 0.97, there is still risk of overshooting during later epochs as we converge closer to the optima. Although the lower learning rates converge slower, they greatly reduce the risk of overshooting during later epochs as we just saw with the learning rate of 1. Having a model overshoot after hours or maybe days of training is a big risk, so it is best to keep the learning rate on the lower side. A low learning rate can also be supplemented with a gradient descent optimizer to boost performance without increasing risk though, as we will see in the following section 'Comparing gradient descent optimizers'.

5.5 Comparing gradient descent optimizers

In section 3.6, we discussed gradient descent optimizers and how they improve upon basic gradient descent by using techniques such as gradient momentum and adaptive learning rate. Here, we will use these optimizers and standard gradient descent to train our neural network with a safe learning rate of 0.01 to prevent overshooting. When comparing the training and validation results for all our optimizers and standard gradient descent, we get the following results:

Already we can see how obvious the free performance boost is for all of our optimizers compared to standard gradient descent without an optimizer. All optimizers achieve

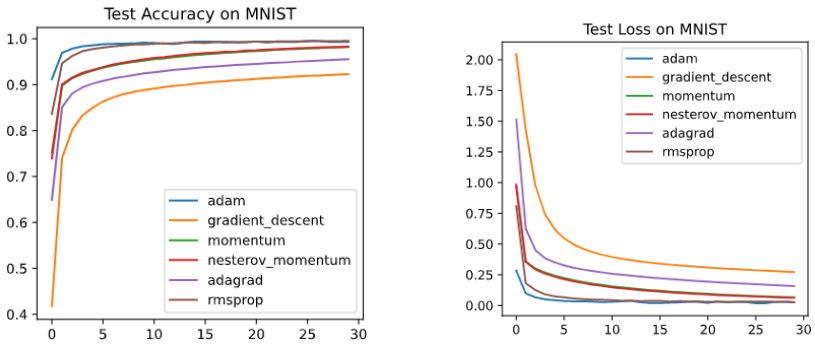


Figure 19: Test accuracy and loss for different optimizers during training

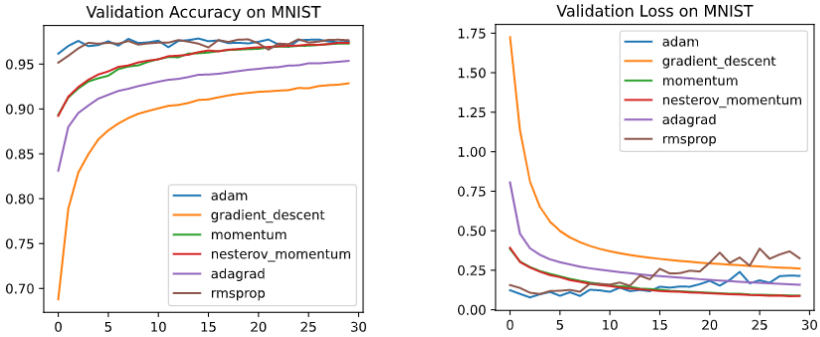


Figure 20: Validation accuracy and loss for different optimizers during training

an accuracy over 0.8 after the first epoch, whilst standard gradient descent achieves an accuracy of 0.69 after just the first epoch. This equates to an over 10 percent increase in performance after just the first epoch without having to do any extra work.

As far as the optimizers themselves go, Adam and RMSProp seem to be the best performing optimizers, with accuracies just above 0.95 after the first epoch, indicating extremely fast learning with these optimizers. Although they converge very fast, we observe fluctuations on the validation accuracy and loss for both of these optimizers. They also both result in a final accuracy of 0.97, practicaly the same as the starting accuracy. These two observations are probably due to these optimizers coming close to the minima initially but not quite converging due to the variance of gradients shown in section 3.5. We also observe slight divergence in the validation loss for both optimizers as the epochs progress but no in the training loss. This is probably due to these optimizers fitting extremely well for the training data and thus starting to overfit for the validation data.

Momentum and Nesterov Momentum seem to perform extremely similarly, almost tracing each other all of the above graphs. We can observe that the accuracy and loss for momentum is slightly worse than for Nesterov Momentum, but this difference is negligible. Although they come to a final validation accuracy of 0.96, the initial accuracies after the first epoch are 0.89, much lower than the initial accuracies for Adam and RMSProp. The reason for these optimizers eventually reaching similar performance to Adam and RMSProp only towards the later epochs most likely due to the build up of momentum

allowing these optimizers to converge faster and faster.

Adagrad appears to be our worse performing optimizer with an accuracy of 0.83 after the first epoch and a final accuracy of 0.92, lower than the final accuracies of all of the other optimizers. This is due to the vanishing learning rate problem mentioned in section 3.6.3 where the sum of past squared gradients is constantly increasing, thus constantly decreasing the size of our learning steps overtime, resulting in a worse performance.

In summary, these results show that you should aim to use a gradient descent optimizer when training a neural network, ideally Adam or RMSProp due to the evident advantages in accuracy and loss. However, when using highly performing optimizers that converge to loss function optima very fast, it may be worth considering early stoppage in training due to potential overfitting in later epochs as we observed in the later epochs for Adam and RMSProp.

6 Summary

In summary, a neural network is an abstract computational model of neural layers in the brain that can be represented by vectors for the network layers, matrices for the network connections, and activation functions for the neuron activity at each entry in the layer vectors.

We measure neural network performance with a cost function, a function parameterized by the weights and biases of the network that compares the output of network to the expected output to see how "wrong" or how much "cost" our neural network incurred when trying to predict an individual example.

To achieve a better performing network (i.e. to make a neural network "learn"), we seek to find weights and biases that minimize the cost function so we get a neural network that performs better prediction. To do this, we use gradient descent, an iterative method of slowly taking steps towards the minima of the cost function with out step sizes being proportional to some learning rate which we can choose. We also discussed how we can use several optimization algorithms to make gradient descent converge faster and more reliably to the optima.

The gradients used in gradient descent are calculated using the backpropogation technique, which involves starting from the output of our network and using four equations called the "backpropagation equations" which we derived in earlier sections.

Finally, using a neural network written and trained with TensorFlow and cuDNN on the MNIST handwritten digit dataset to detect handwritten digit images, we measured the empirical effects of batch size, learning rate and gradient descent optimizer on the learning performance of neural networks. In this section we came to the conclusion that lower batch sizes with low learning rates give the most reliable neural network learning without the risk of overshooting/ divergence in the cost function during gradient descent.

We also observed that the "Adam" optimize was a good choice for a gradient descent optimizer due to its superior performance during neural network learning.

6.1 Unanswered questions/ further reading

One big topic we have not covered is neural network architecture (how to structure the layers of the neural network). This is an important topic as using an optimal neural network architecture can massively improve the efficiency and performance of neural networks.

Going back to our MNIST handwritten digit dataset, what if we wanted to infer the location of a digit in the picture? What if there was background noise in the picture? What if we were using larger inputs rather than just 28 by 28? It turns out that the standard neural network architecture of flat layers just feeding into the next layer can be very inefficient for certain problems, particularly computer vision, which handwritten digit classification is. This is the motivation for using different neural network architectures.

Typical alternate architecture include convolutional neural networks (used for computer vision problems) and recurrent neural networks (used for sequential data and natural language processing problems). The work in this dissertation does not prove useless though as these networks still use gradient descent and backpropagation to optimize the weights and biases.

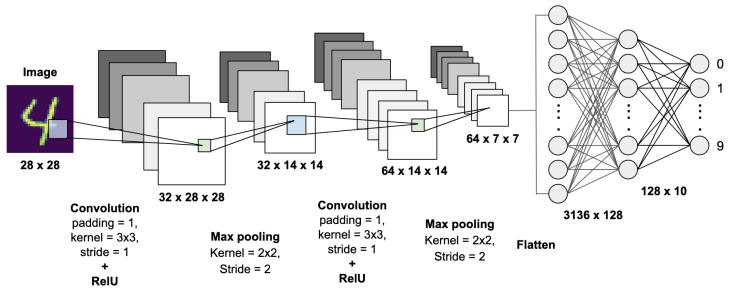


Figure 21: Architecture of a CNN designed for MNIST handwritten digits dataset [20]

For further reading on neural network architectures, see page 405 of "The elements of statistical learning" [8] and the article "Recent advances in convolutional neural networks" [10].

References

[1] *Momentum and learning rate adaptation*, 2020.

[2] Algorithmia, *Introduction to optimizers*.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al., *A view of the parallel computing landscape*, Communications of the ACM **52** (2009), no. 10, 56–67.

[4] David Li Band, *Saddle points and stochastic gradient descent*, 2018.

[5] Léon Bottou, *Large-scale machine learning with stochastic gradient descent*, Proceedings of compstat'2010, 2010, pp. 177–186.

[6] John Duchi, Elad Hazan, and Yoram Singer, *Adaptive subgradient methods for online learning and stochastic optimization.*, Journal of machine learning research **12** (2011), no. 7.

[7] Domagoj Džaja, Ana Hladnik, Ivana Bičanić, Marija Baković, and Zdravko Petanjek, *Neocortical calretinin neurons in primates: increase in proportion and microcircuitry structure*, Frontiers in neuroanatomy **8** (2014), 103.

[8] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al., *The elements of statistical learning*, Vol. 1, Springer series in statistics New York, 2001.

[9] AA Goldstein, *Cauchy's method of minimization*, Numerische Mathematik **4** (1962), no. 1, 146–150.

[10] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al., *Recent advances in convolutional neural networks*, Pattern Recognition **77** (2018), 354–377.

[11] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, Springer Series in Statistics, Springer New York, 2013. pg. 395.

[12] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al., *Applied machine learning at facebook: A datacenter infrastructure perspective*, 2018 ieee international symposium on high performance computer architecture (hpca), 2018, pp. 620–629.

[13] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky, *Neural networks for machine learning lecture 6a overview of mini-batch gradient descent*, Cited on **14** (2012), no. 8.

[14] Steven CH Hoi, Jialei Wang, and Peilin Zhao, *Libol: A library for online learning algorithms*, Journal of Machine Learning Research **15** (2014), no. 1, 495.

[15] Jean-Bernard Lasserre, *complete-guide-of-activation-functions.*

[16] ———, *Six-hump camel back function.*

[17] Dive Into Deep Learning, *Training on multiple gpus*, 2020.

[18] Yann Lecun, *The mnist database.*

[19] Yurii Nesterov, *A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2)*, Doklady an ussr, 1983, pp. 543–547.

[20] Krut Patel, *Mnist handwritten digits classification using a convolutional neural network (cnn).*

[21] Ning Qian, *On the momentum term in gradient descent learning algorithms*, Neural networks **12** (1999), no. 1, 145–151.

[22] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, arXiv preprint arXiv:1609.04747 (2016).

[23] Micol Marchetti-Bowick Ryan Tibshirani, *Gradient descent: Convergence analysis* (2013).

[24] Agnes Sauer, *Quick guide to gradient descent and it's variants.*

7 Neural Network Code Apendix

```
from tensorflow import keras
import tensorflow as tf
from tensorflow.keras import datasets
import matplotlib.pylab as plt
import numpy as np
from tensorflow.keras.callbacks import TensorBoard

(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

# pre-process labells so they can be used in our loss function
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# normalize inputs
x_train = x_train / 255.0
x_test = x_test / 255.0


plt.imshow(x_train[0], cmap=plt.get_cmap('gray'))
plt.show()

x_train.shape, y_train.shape, x_test.shape, y_test.shape

# Initializing default params

# NN architecture :
# 784 (28x28) node input layer
# 250 node hidden layer with ReLU activation function
# 250 node hidden layer with ReLU activation function
# 10 node output layer with Softmax activation function
def create_nn_model():
    return keras.Sequential([
        keras.layers.Flatten(input_shape=x_train[0].shape),
        keras.layers.Dense(250, activation='relu'),
        keras.layers.Dense(250, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ])

# Defining default network params
default_params = {
    'loss_fn': keras.losses.categorical_crossentropy,
    'batch_size': 600,
    'learning_rate': 0.01,
    'epochs': 30
}

# Defining optimizers
optimizers = {
    'rmsprop': tf.keras.optimizers.RMSprop(
            learning_rate=default_params['learning_rate'], rho=0.9,
                                                epsilon=1e-07
        ),
    'adam': tf.compat.v1.train.AdamOptimizer(
```

```
            learning_rate=default_params['learning_rate'], beta1=0.9,
                                                beta2=0.999, epsilon=1e-
                                                07
        ),
    'gradient_descent': tf.compat.v1.train.GradientDescentOptimizer(
                                            learning_rate=default_params['
                                            learning_rate']),
    'momentum': tf.compat.v1.train.MomentumOptimizer(
            learning_rate=default_params['learning_rate'], momentum=0.9
        ),
    'nesterov_momentum': tf.compat.v1.train.MomentumOptimizer(
            learning_rate=default_params['learning_rate'], momentum=0.9,
                                                use_nesterov=True
        ),
    'adagrad': tf.keras.optimizers.Adagrad(learning_rate=default_params[
                                            'learning_rate'])
}

def train_NN_with_optimizer(optimizer_name):
    model = create_nn_model()
    model.compile(
        optimizer=optimizers[optimizer_name],
        loss=default_params['loss_fn'],
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir="logs/{}".format(optimizer_name))

    history = model.fit(
        x_train,
        y_train,
        batch_size=default_params['batch_size'],
        epochs=default_params['epochs'],
        verbose=True,
        shuffle=True,
        callbacks=[tensorboard],
        validation_data=(x_test, y_test)
    )


    return history, model


def train_NN_with_batch_size(batch_size):
    model = create_nn_model()
    model.compile(
        optimizer=optimizers['gradient_descent'],
        loss=default_params['loss_fn'],
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir="logs/batch_size_{}".format(str(
                                            batch_size)))
    history = model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=default_params['epochs'],
```

```
                    verbose=True,
                    shuffle=True,
                    callbacks=[tensorboard],
                    validation_data=(x_test, y_test),

    )

    return history, model

def train_NN_with_learning_rate(learning_rate):
    model = create_nn_model()
    model.compile(
        optimizer=tf.compat.v1.train.GradientDescentOptimizer(
                                                learning_rate),
        loss=default_params['loss_fn'],
        metrics=['accuracy']
    )

    tensorboard = TensorBoard(log_dir="logs/learning_rate_{}".format(str
                                            (learning_rate)))
    history = model.fit(x_train, y_train,
        batch_size=default_params['batch_size'],
        epochs=default_params['epochs'],
        verbose=True,
        shuffle=True,
        callbacks=[tensorboard],
        validation_data=(x_test, y_test),
        )

    return history, model

mnist = {
  'adam': {'history': []},
  'gradient_descent': {'history': []},
  'momentum': {'history': []},
  'nesterov_momentum': {'history': []},
  'adagrad': {'history': []},
  'rmsprop': {'history': []},
  'batch_size_60': {'history': []},
  'batch_size_600': {'history': []},
  'batch_size_6000': {'history': []},
  'batch_size_60000': {'history': []},
  'learning_rate_1': {'history': []},
  'learning_rate_0.1': {'history': []},
  'learning_rate_0.01': {'history': []},
  'learning_rate_0.001': {'history': []},
}

for optimizer in optimizers.keys():
    print('--- {} ---'.format(optimizer))
    history, model = train_NN_with_optimizer(optimizer)
    mnist[optimizer]['history'].append(history)

for batch_size in [60, 600, 6000, 60000]:
    model_name = 'batch_size_{}'.format(str(batch_size))
    history, model = train_NN_with_batch_size(batch_size)
```

```
    mnist[model_name]['history'].append(history)

for learning_rate in [1, 0.1, 0.01, 0.001]:
    model_name = 'learning_rate_{}'.format(str(learning_rate))
    history, model = train_NN_with_learning_rate(learning_rate)
    mnist[model_name]['history'].append(history)
```