Eleftherios Toramanidis
Matriculation number S1958459
BEng Hons Project Report
Image classification using spiking neural networks and artificial neural networks
12 September 2023

Mission Statement

Project Definition

This project aims to develop or improve an already existing python code for image classification of hand written digits (human input) as an Artificial Neural Network (ANN), which will then be implemented in SpiNNaker, a neuromorphic platform, as a Spiking Neural Network (SNN). Tensor flow will be used for creating an Artificial Neural Network on the MNIST dataset.

Main Tasks

Developing a python code for an ANN approach and transferring it into an SNN using the SpiNNaker platform and then implement the SNN model onto the SpiNNaker board

Background Knowledge

Python Programming, Machine Learning, General computing, SNN, ANN.

Acceptance Criteria

Abstract

Two machine learning methods, which mimic the behaviour of the human brain, were trained and tested on an image classification task. namely Artifical Neural Networks (ANN) and Spikign Neural Networks (SNN). ANNs are emulating the concept of neurons and synapses and, using mathematical concepts, learn patterns. SNNs inherit the spike timing characteristic of biological neurons. While ANNs and SNNs have common characteristics, SNNs are more energy efficient. An Artificial Neural Network (ANN) was developed using the Tensorflow software to fit the hand written digits from the Modified National Institute of Standards and Technology (MNIST) database, and then converted into a Spiking Neural Network (SNN). The accuracies of the ANN and SNN on the MNIST testing dataset was found to be 97.65% and 97.5% respectively.

Declaration of Originality

I declare that this thesis is my
original work except where stated.

....

Statement of Achievement

I developed an Artificial Neural Network using my own approach which scored a relatively good final accuracy on a large scale data of handwritten digits and then converted it into a spiking neural network.

Contents

Chapter 1

Introduction

With machine learning and artificial intelligence blossoming in our generation, scientists are motivated to explore the capabilities of computers and exploit them for pattern recognition tasks. A newly introduced concept involves computers mimicking the behavioural and morphological features of the Central Nervous System, specifically those of the human brain.

Although computers are orders of magnitude faster and more effective in computational tasks (such a numeric calculations etc.), the human brain can perform specific functions not only in an easier and quicker manner, but also in a more energetically favourable fashion. An example of human superiority is face classification-recognition, as people can be instantly identified via an intrinsic cooperation between our eyes and neurons. Engineers have been, and still are, experimenting with implementing some functionalities of the human brain, in machine learning tasks, classification and others. However, how does this work? To answer this inquiry a few key parts of our nervous system and their functionallity have to be understood.

1.1 Neurons

Neurons are cells that are responsible for signal reception and transmission in the nervous tissue of most animals [1]. for the purpose of this study, a basic and comprehensive understanding of the fucntions of the human nervous system is necessary. The section below gives an extensive but not elaborate overview of the fucntions of the nerve cells, that will enable the explanation and reasoning behind artificial neurons. For instance, how neural membrane potential changes through ion flow will not be analysed, instead, merely the voltage difference will be considered.

1.1.1 Membrane potential, spikes, spike trains

A neuron's **membrane potential** is the difference in electric potential between the intracellular and the extracellular space, given by the former with respect to the latter. Resting potential, which is approximately -70mV, occurs when the neuron is in a non excited state (no stimuli-input), where the

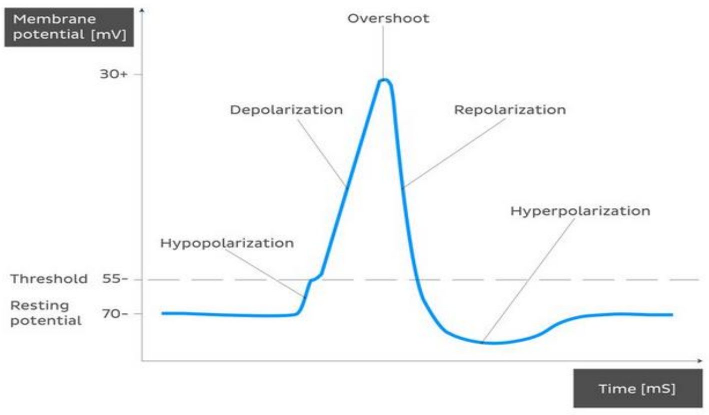

Figure 1.1: *The graph show the membrane potential of a neuron throughout time when the action potential is achieved i.e. when the membrane voltage becomes -55mV [7].*

intracellular voltage is more negative [2].

Action potential is a series of membrane voltages achieved when the membrane potential rises above the threshold voltage, which is approximately -55mV [3]. As we can see in figure 1.1, the voltage drastically increases to +30-40mV only to decrease to a point just below the resting potetnial and finally return to its resting potential [4]. Action potential typically occurs within a few milliseconds. The above behaviour is also described by the term neuron **spike**.

Additionally, on the same figure we can observe the **refractory** period, where the neuron cannot generate another spike [5]. In figure 1.1 the refractory period is where hyperpolarasation happens.

Spike trains detail information of the timings of the occurrence of spikes of a neuron. They do not conserve the action potential shape [6] (see figure 1.2).

1.1.2 Neuron Connections

Neurons are connected through dendrites and axons, the space between them is called synapse. Looking at figure 1.3 , the reception of a signal from other neurons happens through the dendrites while signal transmission to other neurons occurs through the axon [8]. The voltage (signal) passing through the axon and before the synapse, is called the presynaptic potential, while the voltage passing through the dendrite of the receiving neuron is called the postsynaptic potential [9]. There are two types of synapses : excitatory and inhibitory. In the excitatory synapse, the membrane voltage of the receiving neuron has a higher probability of increasing, while in the inhibitory synapse the probability of the receiving neuron's membrane potential increasing is lower [10]. When the membrane potential increases due to stimuli, it

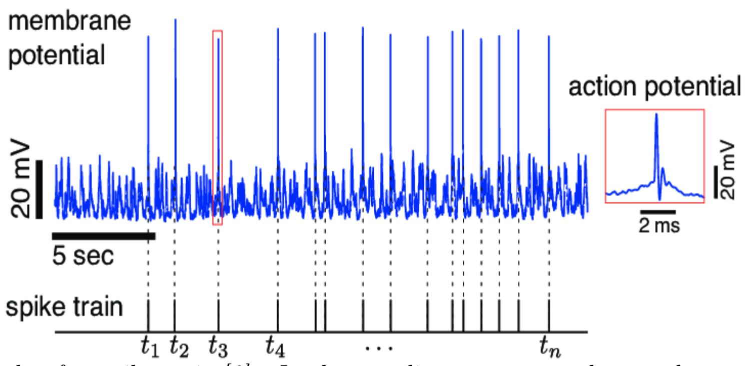

Figure 1.2: *Example of a spike train.[6]. In the top diagram we see the membrane potential of a neuron throughout time with spikes generated on different times. On the bottom graph, only the timings of the spikes are preserved.*

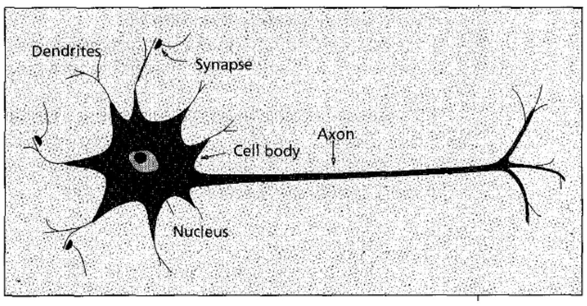

Figure 1.3: *Image of a neuron[8]. The axons serve the signal transmission functionality of the neuron and the dendrites the signal reception.*

instanteneously tends to go back to its resting potential in a logarithmic fashion. To reach the threshold voltage, multiple inputs must arrive in a small period of time, before the membrane potential resets to its resting potential. The **conductivity** of a neuron refers to its ability to transmit the signal to other neurons, hence is relative to the type of the synapse.

1.2 Artificial Neural Networks

Artificial Neural Network (ANN) is a system composed of interconnected artificial neurons, and are used for pattern recognition[11].

1.2.1 Artificial Neuron

Artificial neurons are the computational unit of ANNs. Their functionality is to collect input data and produce an output [12]. A simple artificial neuron model produces an output which depends on the input

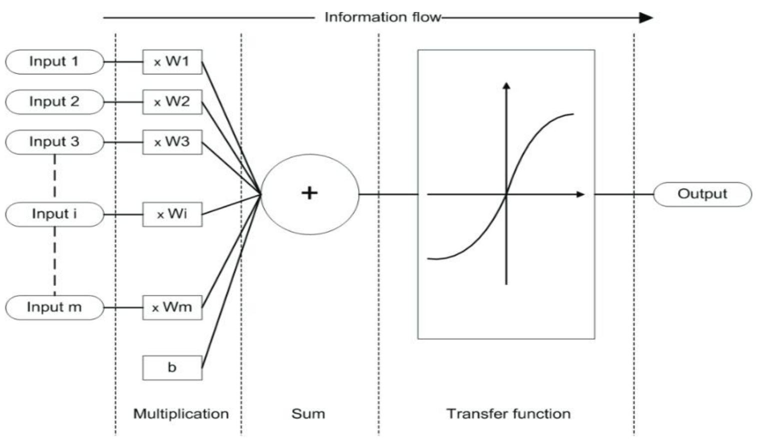

Figure 1.4: *Image of an artifical neuron[11]. All the inputs are multiplied by a weight before reaching the neuron. Then the neuron sums the multiplied inputs and that will be the input to the transfer (also knows as activation) function (see 1.2.2 for activation functions)*

in the following way:

$$\sum_{i=1}^{N} W_N X_N \tag{1.1}$$

where N is the number of input connections with other neurons, X_N is the input supplied by those neurons, which can be interpreted as the presynaptic membrane potential and W_N is the weight by which the input is multiplied by to give the postsynaptic potential, which can be interpreted as the conductivity of the synapses i.e. if it is a excitatory or inhibitory1.4 . The model provided is called the McCulloh-Pitts model [13], and can be adjusted with a bias to increase performance. If the weighed input sum crosses a specific threshold voltage, the output produced by the neuron, will be forwarded to other neurons that are connected to the original neuron, with the value of 1 (which represents the presynaptic voltage), otherwise the output is 0.

1.2.2 Activation Functions

Activation functions refer to the processing done within the neuron after the weighed inputs are summed. They are used to solve the issue of linearity yielded by the equation 1.1, which can be inefficient when applied to real world non-linear data. Adding a bias and an activation function f the final output of the neuron is:

$$f(\sum_{i=1}^{N} W_N X_N + bias) \tag{1.2}$$

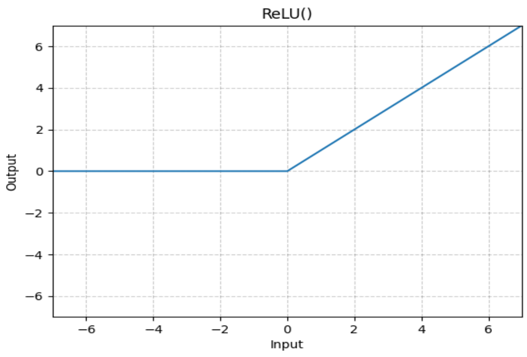

Figure 1.5: *A plot of the ReLU activation function.*

Unlike the McCulloh-Pitts model, the range of values of the output in 1.2 depends on the activation function. While there are a few activations functions, it is worth mentioning only 3 that were considered in this project, ReLU, sigmoid and softmax.

ReLU (Rectified Linear Unit) returns 0 for a negative input or it returns the input itself, when it is positive.

$$f(x) = max(0,x) \tag{1.3}$$

It is widely used due to simplicity and networks implemented with ReLU allow easier optimization and require less computation([14], 1.5).

Sigmoid is a nonlinear activation function which allows neurons to understand more complicated data patterns. The equation of the sigmoid function [15] :

$$f(x) = \frac{1}{1+e^{-x}} \tag{1.4}$$

For large positive and negative values of x the value tends towards 1 and 0 respectively. A form of linearity is observed when the input is close to 0 1.6. While solving the linearity issue, the sigmoid function requires more computation power compared to the ReLU and can run into the vanishing gradient descend problem [1].

Softmax differs from the other two activation functions, since it converts real values into probabilities. Its equation is given by :

$$f_j(z) = \frac{e^{z_j}}{\sum\limits_{k=1}^{n} e^{z_k}} \tag{1.5}$$

[17].The input z refers to a vector with the values of the inputs to all of the neurons of interest, for instance the neurons in a layer [2], j refers to the neuron of interest, z_j refers to the input of the neuron of interest and n is the number of neurons in the group examined. The output of the softmax function

[1]As we will see later, learning algorithms utilize the gradient of the activation function and on extreme values the gradient tends to zero [16].

[2]Explanation of layers will be provided in the structure and organization part

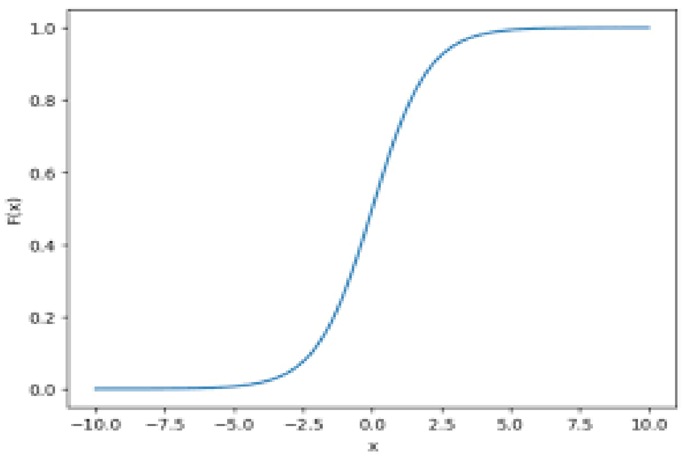

Figure 1.6: *A plot of the sigmoid function.*

yields the probability of the input data being classified as the object that jth neuron represents [3].

1.2.3 Structure and data movement

The structure of all ANNs has the same main components as described below, however the data movement can change depending on the type of model. The two main types of ANNs with respect to their types of connections are feed-forward [18]and recurrent [19]. In the feed-forward neural networks, data is transferred from the input towards the output, while in the recurrent neural networks, there is some kind of feedback from the outputs of the neurons to the inputs of neurons [20]. For the purposes of this study only the feed-forward data movement will be thoroughly explained.

Structure

ANNs are organized in multiple layers, where each layer is comprised of multiple neurons and has a chosen activation function to be applied to its neurons. An ANN must have an **input layer** where the raw data [4] to be classified is entered. The number of neurons in the input layer must match the dimensions of the data [21], for instance if we are trying to classify a pixel's colour, which is composed of 3 subpixels-dimension (red, green, blue), where each subpixel can get a value between 0 and 255, we would have 3 neurons in the input layer, so that each neuron can "capture" one subpixel value from the pixel data.

In the ANN there must also be an **output layer**, the number of whose neurons must be equal to the number of objects,that the data can be classified as, with each neuron representing one object. In our previous example, if we were trying to classify pixels in only red, green or blue colors we would have 3

[3]In the output layer, each neuron represents an object, that the data given can be classified as (see section 1.2.3).
[4]The ANN will not change the data that is given to it, although preprocessing can be done (such as data normalisation or shifting) before feeding in the data to it.

nodes in the output layer, and the data would be classified as the color that is represented by the node with the highest value.

Layers between the input and output layer can exist and are called **hidden layers**. Plenty of researches have been done on the number of hidden layers a network should have, whatsoever there is still very little intuition built. A rule of thumb suggests that the number of hidden layers depends on the complexity of the problem they are applied on [22]. An example of an ANN can be found in figure in figure 1.7.

Data movement

In a feed-forward ANN, data is entered in the input layer and travels across layers, until the output layer (left to right convention [18] in figure 1.7). Data with n dimensions (or attributes) will require an ANN with n nodes in its input layer. Each input layer neuron will have a separate weight for each connection it has with the next layer neurons [23]. Usually, activation functions on the input layer neurons, are not applied. Conversely, each hidden layer neuron has a connection with every neuron from the previous layer. A hidden layer neuron, sums all the received signals and produces an output according to figure 1.4. Once all of the hidden layer neurons have computed their output, these outputs have to be forwarded to the next layer in the same manner. The data will finally arrive to the output layer, where the output of each neuron withing the layer can be calculated and represents the confidence, the system has, that the data presented to it, is of the class that the neuron refers to. For instance, if we present some attributes of a car to an ANN with the possible car brands being TOYOTA, HONDA and BMW (leading to 3 output nodes), and the ANN calculates an output of 3.14, 2.71 and 10 respectively for those brands, then the system will classify the data as BMW.

1.3 ANN learning

The prediction of an ANN model on a type of data depends on the weights of the model. Finding the best combination of weights can be challenging. The process where the ANN tries to find the optimal weights is the learning process. During this process, examples of data are presented to the model, one at a time, and after the model gives a prediction on the specific data point, it tries to adjust its weights to give the correct output. Learning is split into two categories : supervised [24] and unsupervised [25], see section 1.3.1. Supervised learning makes use of an error function, which essentially denotes how distant the prediction from the actual value is, and is utilized by rectification (also called learning) algorithms. Those algorithms aim to minimize the error function, with the most used one being the back-propagation algorithm, for supervised feed-forward networks [23].

1.3.1 Supervised and Unsupervised learning

In supervised learning, the label of the data, during training, is given to the model, and compared to the prediction of the model on that data. The model, adjusts its parameters after the comparison, in order to come closer to predicting the true label of the data [24] in the future. Some of the learning processes

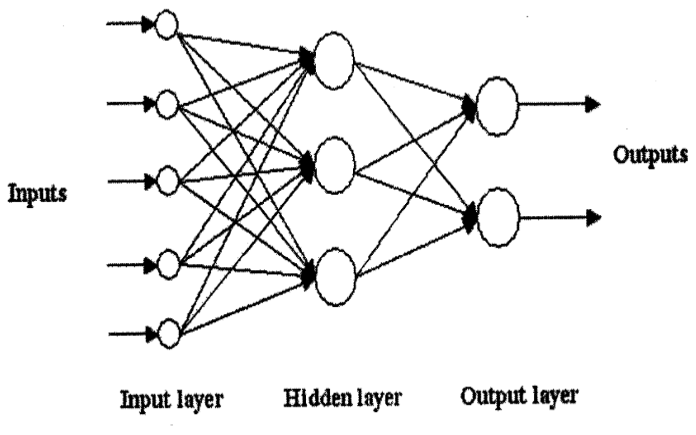

Figure 1.7: *This is an ANN with one hidden layer[20]. Neurons that are vertically alligned belong to the same layer. The edges represent connection between neurons (all to all connections).*

that use supervised elarning are : Neural Networks [26], Naive Bayes [27], Linear Regression [28] and others [29]. In unsupervised learning, the learning model tries to cluster together data and essentially understand patters, instead of naming those patterns [25]. Unsupervised learning is used in : K-means clustering [30], Probabilistic clustering [31] and others [25]

1.3.2 Error Functions

As mentioned previously, error functions tell, how wrong a prediction of an ANN was, compared to the label of the data. Most of the times, an error function includes the computation of the difference between the prediction and the desired value, and is considers all the errors, not just one data point. Examples of such functions are the Mean Absolute Error (MAE) function, the Mean Squared Error (MSE) function, the Categorical Crossentropy and others.

Mean Absolute Error

The absolute error is first calculated, for all the data points, and then the mean is computed. The final formula of MAE is :

$$MAE = \frac{\sum\limits_{i=1}^{n} Y_i - X_i}{m} \tag{1.6}$$

where n is the number of data points fed into the model , Y_i refers to the actual label of the data ith data point, X_i refers to the prediction of the model on the ith data point and m is the number of errors [32].

Mean Squared Error

The MSE is calculated similarly to the MAE, with the exception that the difference between the target value and the predicted values, is squared. The equation is as follows:

$$MSE = \frac{\sum_{i=1}^{n}(Y_i - X_i)^2}{m} \tag{1.7}$$

where the same symbol convention as in the MAE explanation, is used [33].

Categorical Crossentropy

Categorical crossentropy is a loss function that takes into consideration the probability of the class the model model predicted (i.e. the confidence of the model that the given data is of a specific class) and the truth label of that class. The output of this loss function is calculated as :

$$L_{CE} = -\sum_{i=1}^{n} t_i \cdot log(p_i) \tag{1.8}$$

where n is the number of classes, t_i is the true value of class i (which should be equal to 0 except for when the i class refers to the actual label of the data) and p_i is the predicted output (or probability) of class i [34]. The output of the loss function in equation 1.8 is related to the parameters of the model (weights and biases) through the term p_i.

1.3.3 Gradient descent

Gradient descent is an algorithm that aims to minimize a function, which in the case of ANNs, could be the error function. Minimizing the error will lead to better performance of a model and involves adjusting the parameters of the model, such as the weights and biases. Firstly, we should assume some values for the parameters of the model, and this will be the starting model we are trying to optimize. For instance, it is common to set all the weights in an ANN equal to 1 and the biases equal to 0, although, there are occasions where there is some intuition about the problem and the data, that could lead to some instinctive choices about the initial parameters. The derivative of the error function is taken, with respect to one parameter to be optimized (partial derivatives). An error function usually yields a positive value, depending on the chosen equation (see equation 1.7 for an example), and the actual value of the parameter is substituted in the derivative equation. By intuition, we can conclude that if the derivative of the error function with respect to a parameter is positive, the value of the parameter should be decreased in order for the error to get closer to 0, while if the derivative is negative, the value of the parameter should be increased. The amount of change of the parameter, depends on the magnitude of the gradient, as well as a chosen learning rate (a scaling factor) [35]. The magnitude is determined by the average of the gradient of the error of all training examples. Particularly, first the partial derivative of the error function is computed, for all weights and biases, for each training example. Each training example will contribute its own desired twist to each parameter, so that the parameters can come closer to predicting the correct value for that example. The average of those twists, is computed for each weight and bias, to yield the magnitude of the desired change. The step by which the variable should change is thus defined as :

$$Stepsize = -\eta \frac{dError(p_{current})}{dp} \tag{1.9}$$

and the updated parameter has the equation :

$$p_{new} = p_{current} - \eta \frac{dError(p_{current})}{dp} \tag{1.10}$$

where, p is the parameter to be optimized, $p_{current}$ is the current value of the parameter, η is the learning rate and p_{new} is the value to replace p_{old}. In general, when the the current value is way off the target value that yields the minimum error, large steps are preferred in order for the parameter to reach the desired value faster, and when the current value is close to the target value, smaller steps should be taken, ensuring that there is no overshoot and oscillation [36]. Integrating for all parameters, we get the equation :

$$\mathbf{P}_{new} = \mathbf{P}_{current} - \eta \cdot \nabla_{\mathbf{P}} Error(\mathbf{P}_{current}) \tag{1.11}$$

where $\mathbf{P}$ refers to all the weights and biases as a vector [37].

While the algorithm tries to minimize the error function, it does so by computing the gradient, which can be problematic or not optimal. A local minimum will deceive the algorithm, if it is not the global minimum, into being satisfied with the error calculated, which is observed in non-convex [5] functions, see figure 1.8. In addition, the algorithm will not be able to make any progress by using the derivative of the error function since the latter will be close to 0, leading to very small changes in the weights and biases of the model.

This is the original Gradient Descent algorithm and is also referred to as batch Gradient Descent, because the derivative of the error computed accounts all training examples.

1.3.4 Gradient Descent alternatives

The Gradient Descent algorithm, takes all of the training examples to change the weights and biases of the system once. However, the system will have to go through the training data multiple times before having an appropriate accuracy, which could be an expensive process if the training data is large.

Stochastic Gradient Descent (SGD) [38] on the other hand, updates the weights and biases after computing the gradient of the error of each training example. SGD appears to be computationaly faster,

[5] A function is convex if, when two points points of the function are joined by a straight line, there is no instersection between the line and the original function. In a non-convex function, the line and the original function intersects

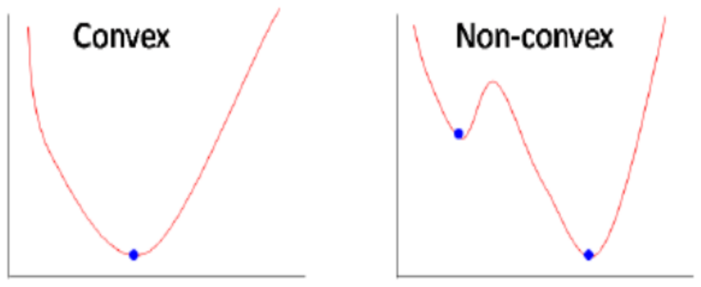

Figure 1.8: *In both figures, the error functions is plotted against a parameter (weight or bias). In the left figure, the local minima is also the global minima (convex function). In the right figure, the local minima is not always the global minima, which can make the computation of the optimal value for the examined parameter, inconvenient. Since the Gradients Descent algorithm changes the variables depending on the derivative of the error function, in the local minima of the non-convex error function, the variable would experience very little change.*

since one sample is processed at a time. It also updates the weights and biases more frequently, which gives a higher probability to the model to "escape" local minima, and work its way towards a global minima (or a worse local minima!). Noisy examples can have a greater impact on the training, which in the case of batch Gradient Descent would fade away (since the average of the errors is computed) [37].

Mini-batch Gradient Descent [39] is a combination of the SGD and the main Gradient Descent ideas. A number of training examples is taken at a time to calculate the average of the gradient of the error. A change in the variables of the model is made accordingly and then another set is considered, until all training examples are used. This algorithm offers a frequent update of variables, while forbidding noisy data from having a significant effect on them.

1.3.5 Back-Propagation algorithm

Back-propagation is an algorithm used to improve the performance of machine learning models, such as ANNs. It is the most famous algorithm used for feed-forward Neural Networks and it utilizes the gradient descent algorithm.

Back-propagation is a method for optimizing the weights and biases based which depends on the gradient of the error function with respect to each of those parameters. The way this is accomplished, is by starting from the weights and biases of the output layer connections and working through all the neurons in that layer before moving to the previous layer's connections, as described by the name of the method. Let us look over a simple example of an ANN composed of 1 input layer, with only 1 neuron, 1 hidden layer, with 2 neurons and an output layer of 1 neuron as depicted in figure 1.10. Assuming that the starting weights w_1, w_2, w_3, w_4 are all equal to 1 and bias b is equal to 0 and using the MSE (equation 1.7) function on 2 different inputs x_1 and x_2 with labels y_1 and y_2 respectively, with the corresponding prediction of the model being y_1' and y_2', and multiplying the equation 1.7 by m for simplification, the error is described by :

$$Error = \sum_{i=1}^{n}(y_i - y_i')^2 \tag{1.12}$$

since we have 2 data points, substitution of n by 2 can be made ,and expanding the sum we get:

$$Error = (y_1 - y_1')^2 + (y_2 - y_2')^2 \tag{1.13}$$

The prediction of the model is the output of the output neuron as per figure 1.10 and equation 1.13 becomes :

$$Error = (y_1 - (\sum x') - b)^2 + (y_2 - (\sum x') - b)^2 \tag{1.14}$$

At this point, we choose to optimize the bias b of the model, so according to the gradient descent we first need to calculate the derivative of the error function in equation1.14 with respect to b :

$$\frac{dError}{db} = \frac{d(y_1 - (\sum x') - b)^2}{db} + \frac{d(y_2 - (\sum x') - b)^2}{db} \tag{1.15}$$

and by applying the chain rule :

$$\frac{dError}{db} = -1 \cdot 2(y_1 - (\sum x') - b) - 1 \cdot 2(y_2 - (\sum x') - b) \tag{1.16}$$

$$= -2(y_1 - (\sum x') - b) - 2(y_2 - (\sum x') - b) \tag{1.17}$$

All of the values in equation 1.17 are known and the equation itself can be used to calculate the updated value of b following the equation 1.10. After a better value for the bias is calculated, the same process can be followed for the same or a different parameter, until the performance improves to an acceptable level, depending on the application.

In the above example, the chosen parameter is directly related to the prediction of the model. That is not the case for the optimization of parameter in previous layers, such as w_1 and w_2. The difference is that the $\sum x'$ term in equation 1.14 needs to be unravelled before the differentiation of the error function. Differentiation of this term can become challenging, since activation functions are included, and that is for the unravelling of a neuron belonging to a layer, precedent to the output layer. With multiple hidden layers, can become difficult for a human to solve.

Another key factor in the learning process of an ANN is the η scaling factor, which will be discussed in the next subsection. During learning, the model is allowed to go over the training data as many times as specified by the designer, updating its weights and biases each time. The number of times it uses the same training data for learning is termed as **epochs**.

1.3.6 Learning rate η and Optimizers

The learning rate is the other key factor in weights and biases optimization. Finding a proper learning rate, can lead to faster convergence (reaching the minima) of the error functions. Another common issue in the minimisation of the error function is the oscillations. To understand the oscillation probelm, let us look at the convex function in figure 1.8. Supposing that the parameter of interest yields and error that is before the minima of the error function by a small amount. If the learning rate is very high, the parameter will lead to an error that is after the minima point. When the parameter keeps oscillating back and forth from the minima point without getting close to it, no progress is made. Be that as it may, if the learning rate is very small, then the progress made towards the minima point can be very little and the training process becomes longer. A few methods have been developed to overcome the issues above and enhance the learning of the model [37].

Momentum

The Momentum optimization method, aims to smooth out unnecessary changes in variable values, while embracing more important changes. It makes use of previous updates to calculate the new updates to the weights and biases. In figure 1.9 (a) , the vertical oscillation is irrelevant towards minimising the error function (centre of the graph), instead only the parameter in the horizontal direction should be changed. In the same figure, but looking at graph (b) the momentum algorithm is used, where more gravity has been given towards the parameter on the horizontal axis [40]. From equation 1.11 we can calculate the changes to be applied to all variables at time t :

$$\Delta \mathbf{P}_t = -\eta \cdot \nabla_{\mathbf{P}} Error(\mathbf{P}_t) \tag{1.18}$$

where $\mathbf{P}_t$ is the values of all the variables at time t. With the inclusion of momentum, the changes to be applied to the variables become :

$$\Delta \mathbf{P}_t = -\eta \cdot \nabla_{\mathbf{P}} Error(\mathbf{P}_t) + \beta \cdot \Delta \mathbf{P}_{t-1} \tag{1.19}$$

where β [6] denotes the dependency on the previous change $\Delta \mathbf{P}_{t-1}$. If the sign of $\Delta \mathbf{P}_{t-1}$ agrees with the sign of $-\eta \cdot \nabla_{\mathbf{P}} Error(\mathbf{P}_t)$, then there is an accumulation of change towards one direction. while if the signs disagree, one term will start balancing out the other term[41].

Adagrad

The Adagrad optimization algorithm stands for adaptive gradient. It aims to increase η for parameters that are updated infrequently and decrease it for parameters that are updated frequently, which means that a form of memory is utilized as in the Momentum algorihm [42]. Adagrad performs well for sparse data. The gradient of the error function with respect to a parameter (or set of parameters) P_i, at time step t, is denoted as $g_{t,i}$ and given by the equation :

$$g_{t,i} = \nabla_{P_t} \cdot Error(P_{t,i}) \tag{1.20}$$

[6] β should be between 0 and 1. The most common value is 0.9

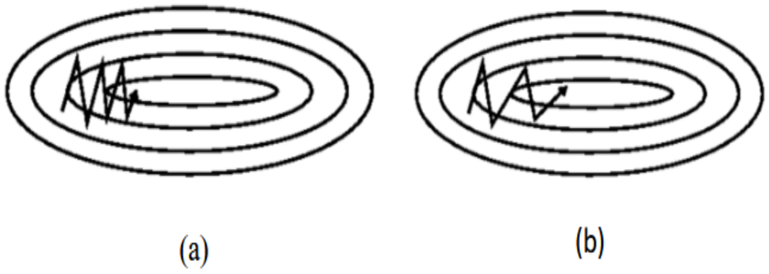

Figure 1.9: *The error computed with respect to two different parameters, projected on those parameters (contour plot), is shown in both figures. In (a) no momentum is used while in (a) momentum is used. The minima of the error function is located in the middle of both graphs. [40]*

At the next time step $t + 1$ the parameter(s) $P_{t+1,i}$ are updated as follows :

$$P_{t+1,i} = P_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} \cdot g_{t,i} \tag{1.21}$$

where $G_{t,i}$ is a diagonal matrix with each element positioned on the diagonal representing the sum of the squared gradients with respect to $P_{t,i}$, and ϵ is a small number added to ensure there is no division by 0 [43]. For infrequently updated parameters the term $G_{t,i}$ in the denominator is small, which leads to an increase in the step size of the parameters $P_{t,i}$.

Adadelta

In the Adagrad algorithm (equation 1.21), as time passes, the $G_{t,i}$ term gets larger, leading to a decrease in the learning rate, therefore making very little steps towards the optimal values. Adadelta deals with this issue by averaging the squared errors of previous steps and depending the current change of variable in the previous changes of the variables ΔP_{t-1} in a Root Mean Squared (RMS) fashion [44]. The Adadelta still considers previous gradients but instead of accumulating all of them, it only accumulates a small window from them. Computing the average of the previous squared gradients, also includes appointing a gravity to them ρ. The current gradient squared has a gravity of $1 - \rho$. The equation of the updates of the parameters is derived from the Adagrad equation. The term $E[g^2]_t$ is the factor denoting the dependency on previous gradients and the current gradient g_{t-1}^2 is computed as :

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2 \tag{1.22}$$

where ρ usually has a value of around 0.9 [44]. The $E[g^2]_t$ needs to be initialized to 0 and the window of dependencies to previous gradients is determined by the value ρ, which has a similar functionality to the Momentum algorithm's β value. Defining the RMS of any value x to be :

$$RMS[x] = \sqrt{E[x^2]_t + \epsilon} \tag{1.23}$$

the final equation of of a parameter or a set of parameters P at time $t+1$ using the Adadelta optimization is :

$$P_{t+1} = P_t - \frac{RMS[\Delta P]_t}{RMS[g]_t} \cdot g_t \tag{1.24}$$

[44]. In this optimization algorithm, there is not learning rate η.

RMSprop

RMSprop was developed with the same motivation as the Adadelta, to restrict the growth of the $G_{t,i}$ term in the Adagrad equation. The only difference between the RMSprop and the Adagrad method, is that RMSprop replaces the $G_{t,i}$ term with $E[g^2]_t$, where the latter is calculated following equation 1.22 (which has a typical value of $\rho = 0.9$). The final equation of RMSprop is :

$$P_{t+1} = P_t - \frac{\eta}{RMS[g]_t} \cdot g_t \tag{1.25}$$

where a common value for η is 0.001 [37].

Adam

The Adam optimizer, inherits characteristics from the from the Adadelta (using the average of previous squared gradients) and the Momentum (average of previous gradients) algorithms. The average of the previous gradients squared is calculated following equation 1.22 and the average of previous gradient follows the exact same equation, replacing the term g^2 with g where ρ can have a different value from the ρ used in $E[g^2]_t$:

$$E[g]_t = \rho_2 E[g]_{t-1} + (1 - \rho_2) g_t \tag{1.26}$$

[37]. Since $E[g]$ and $E[g^2]$ are initialised as 0 vectors, they will be biased towards 0 and to overcome this issue they can be re-written as:

$$E'[g]_t = \frac{E[g]_t}{1 - \rho_2^t}, \tag{1.27}$$

$$E'[g^2]_t = \frac{E[g^2]_t}{1 - \rho^t} \tag{1.28}$$

[45]. The updated parameters P follow are computed as :

$$P_{t+1} = P_t - \frac{\eta}{\sqrt{E'[g^2]} + \epsilon} \cdot E'[g] \tag{1.29}$$

[45]. From the Adam optimizers, other optimization methods are derived such as the AdaMax [46] and Nadam [47].

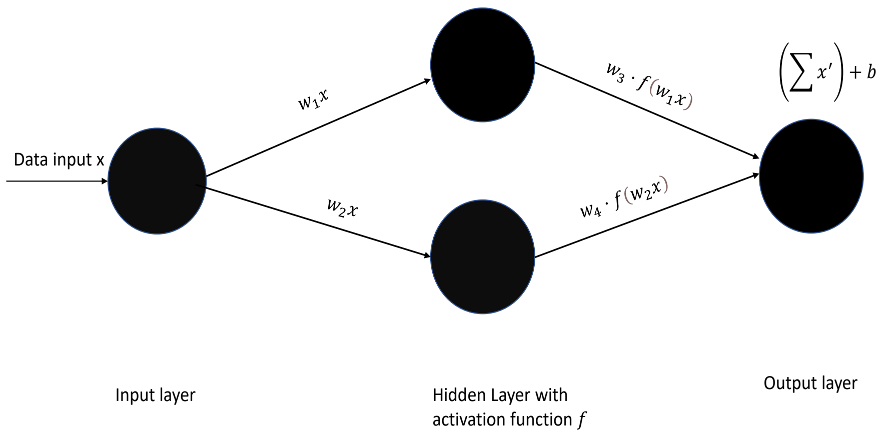

Figure 1.10: *A simple feed-forward ANN with 1 input node, 1 hidden layer with 2 nodes and 1 output node. Usually each hidden layer node would need to sum the incoming data from all the previous layer's nodes, however in this occasion, there is only 1 node in the preceding (input) layer, because the data is of one dimension. The hidden and output layers have activation functions represented by the symbols f and f', which could be referring to the same function. In the equation of the output neuron x' represents the outputs of the previous layer's nodes , which in our case are $w_3 \cdot f(w_1 x)$ and $w_4 \cdot f(w_2 x)$. The output node has a bias b which is added to the input of its activation function f' while the hidden layers do not have a bias. Additionally, the output neuron does not have a weight in this model.*

1.4 Spiking Neural Netwroks

Spiking Neural Networks (SNN) have the same structure as ANNs, they are comprised of layers and neurons. Their major difference is the behaviour of their neurons. A spiking neuron inherits more functionalities from the biological neuron, unlike the artificial neuron. There is a number of spiking neuron models available, however only the Leaky Integrate and fire (LIF) model and a simplification of that model, the Integrate and fire (IF) model, will be explained. The hardware implementation of those models (on an FPGA board for example, see [48] and [49]) is not taken into consideration for the purposes of this study.

Training methods of SNNs can be classified in 3 main categories, namely, Shadow training, Back-propagation using spikes and local learning rules. In this paper, only Shadow training is examined and used for developing an SNN and involves building an ANN and then converting it into an SNN [50].

1.4.1 Spiking Neuron

The **leaky integrate** and **fire** (LIF) model is the dominant model to represent the behaviour of a human neuron, and its behaviour is shown in figure 1.12. A spike (that looks like figure 1.1) in one of the presynaptic neurons in 1.12 will lead to a change in potential in the postsynaptic neuron. As described in a previous section, after the change in the membrane potential of a neuron, the voltage tends to go back to its resting potential, thus the term Leaky in the LIF model. The postsynaptic neuron itself, builds a membrane voltage, which if increased at the the threshold voltage, will generate a spike to be forwarded to the next layer neurons (through its axons' connections) [51]. The integration of the incoming signals, by the postsynaptic neuron is described by the "integrate" part in the LIF terminology, and "fire" stands for the ability of the neuron to forward the signal to other neurons, when it reaches the threshold voltage. For such a neuron model, the equation of the membrane potential u of any postsynaptic neuron i at time t is :

$$u_i(t) = \sum_j \sum_f e_{ij}(t - t_j^{(f)}) + u_{rest} \tag{1.30}$$

where e_{ij} refers to the voltage profile imposed on postsynaptic neuron i from the presynaptic neuron j as described in figure 1.11. Looking at equation 1.30, the change in voltage of the neuron of interest, is not related to the voltage itself in the presynaptic neuron, but only the spikes and the spike timings.

Another neuron model is the **Integrate** and **fire** model, which has a similar functionality as the LIF model, with the only difference, that when the voltage membrane is increased due to a presynaptic spike, the membrane voltage of the postsynaptic neuron, will not start leaking towards its resting potential i.e. in equation 1.30, ϵ_{ij} will be different.

1.4.2 Rate and Latency coding

In spiking neurons, as we can see in equation 1.30, the membrane voltage of a neuron depends on the aggregation of the inputs. However we have previously mentioned that if those inputs signlas, from

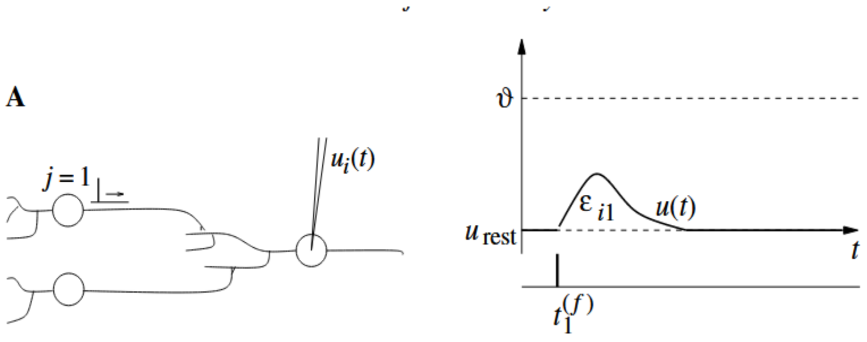

Figure 1.11: *On the left figure, there are 2 presynaptic neurons (aligned vertically) whose spikes can lead to a change of the membrane potential of the 3rd neuron. The voltage at any given time of the postsynaptic neuron is shown on the figure on the right, where ϵ_{i1} is the profile of the voltage change of the postsynaptic neuron i due to a spike in the presynaptic neuron $j = 1$ over time. ϵ_{i1} has an equation itself, which justifies its shape, but it will be abstracted as it is irrelevant to the aims of this paper. It does not take into consideration multiple spikes of other presynaptic neurons or the same neuron.*

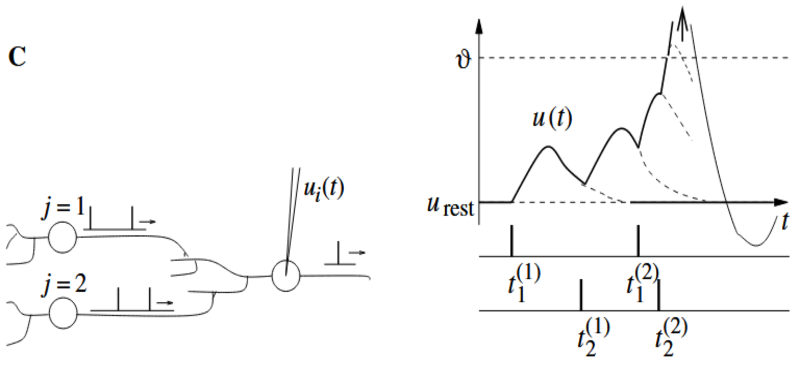

Figure 1.12: *Picture on left shows a postsynaptic neuron, with 2 presynaptic neurons evoking potential on the postsynaptic neuron. Neurons 1 and 2 ($j = 1$, $j = 2$) can be considered to belong to the same layer. Picture on the right shows the change in potential over time of the postsynaptic neuron, where $t_x^{(y)}$ refers to the time when the y th spike of the the x th ($j = x$) neuron leads to an increase in potential of the postsynaptic neuron. The dotted lines, describe what would happen if there was not another spike in the presynaptic neurons, to increase the voltage again, as shown in figure 1.11, where the voltage will gradually would go back to its resting potential [51]. Backpropagation for weight update cannot be used, since spikes can be difficult to differentiate. Additionaly, on the right figure, at the bottom we can see the, previously discussed, spike trains of the presynaptic neurons.*

other presynaptic spikes, are far apart in time, the membrane potential of the neuron can reset to the resting potential before another signal arrives at its dendrites. Therefore, for a neuron to generate a spike, the received signals must at relatively small time intervals, in order for the its membrane potential to gradually increase and finally reach the threshold potential. We can thus conclude that in a Spiking Neural Network, only the spiking trains are relevant for the computation of the prediction of the model.

Rate coding

In rate coding the input intensity is converted into spiking frequency/rate. The highest possible intensity on the input is replaced by the most frequent spikes in time, depending on the designer's choice for maximum frequency and the device's specifications. All input values are converted to frequency following the formula :

$$f = \frac{x}{x_{max}} \cdot f_{max} \tag{1.31}$$

where f is the frequency by which the neuron with value x will transmit signals to the next layer. Determining the output, includes monitoring all output nodes, and selecting the one with the highest firing frequency [50].

Latency coding

In latency coding the frequency by which a neuron fires is no longer of interest, instead the timing of the spike is relevant. The values of the input data are now ordered depending on their magnitude, so that the the timing that the corresponding neurons fire gets longer, the smaller the value is. A model that uses latency coding, classifies an input as the cluster represented by the neuron in the output layer,that fired first [50].

1.5 Neuromorphic hardware

Neuromorphic hardware is hardware that has been designed for the sole purpose of implementation of spiking neural networks. Its purpose is to increase computation speed while being more energy efficient than other devices [52]. The dominant hardware type that SNNs are implemented on is the Field-Programmable Gate Array (FPGA) [53, 49], while a remarkable project is the SpiNNaker project by the University of Manchester which has released hardware with up to 2500 processors [54, 55].

Chapter 2

Impact of ANNs and SNNs

ANNs and SNNs are two approaches to machine learning and patter recognition problems. They are being used for image and speech recognition, medicine, food safety [56] and other fields [57]. The main motivation for using SNNs specifically is its power consumption which is considerably lower than other machine learning algorithms [58].

2.1 Impact of ANNs

ANNs are currently being used in a variety of industries. Clinical practitioners are utilizing ANNs for disease diagnosis such as cancer. By simply creating an ANN, training it with data and choosing the correct parameters to simulate, a prediction can be made [59]. Regarding financial decisions, stock predictions can be made using artificial neural networks [60]. In a research conducted by IEEE [61], where they compared different number of inputs on the ANN and concluded that with more inputs the accuracy increases. Their error varied from less than 1% to around 20%. Althoughv ANNs seems to perform well in some situations, there are cases where careful decisions have to be made and complete understanding of how ANNs is necessary [62]. The crucial information is that even if ANNs may have a high accuracy and be helpful, the false negatives and false positives may have a large cost. In the instance of cancer detection, a false negative (i.e. the subject is predicted to not have cancer) can lead to ignorance to an unrecoverable state.

2.2 Impact of SNNs

The applications of SNNs are similar to the ANNs' and evolved around image recognition, object classification, decision making and others [63]. An advantage of the SNN over the ANN approach to problems is the power consumption, which in the case of SNN is lower. In image recognition tasks SNN has reportedly shown that it can perform at the same level as an ANN and its ability to run on neuromorphic hardware, reducing its power consumption [64]. Other studies have shown promising results in energy efficient and fast computation within robotics [65] and robot navigation [66].

Chapter 3

Experiment

This experiment's purpose is to compare the performances between an ANN and an SNN approach to the classification of hand written digits. The data that was used is taken from the Modified National Institute of Standards and Technology (MNIST) database [1]. Firstly, an ANN is built, and through shadow training, it is converted into an SNN. Both of those tasks are done with python programming.

3.1 Building ANN

For this task, Tensorflow [67], a python library, will be used. The main goal is to find the best parameters that have the highest accuracy on the MNIST dataset. Those parameters are :

- Type of model
- Number of hidden layers
- Number of neurons within the hidden layers
- Activation functions applied to each layer
- Loss function
- Optimizer
- Batch training size
- Number of epochs

Those parameters will be referred to as hyper-parameters.

[1] The MNIST data can be downloaded from `http://yann.lecun.com/exdb/mnist/`

3.1.1 Building SNN

The ANN developed will be converted into an SNN using shadow training. The method of conversion that will be followed is inspired by the method used in [68] and described by the following steps :

1. Use ReLUs at all layers (if the ANN model does not use only ReLUs, change the model to satisfy this rule)
2. Save the weights and load them on the SNN model
3. Use weight normalisation

Snntoolbox [69] is the software that will be used to convert the ANN to SNN, and it does support ANN models created by Tensorflow. The SNN that will be developed will utilize the LIF neuron model and information propagating through the network will be of the frequency code nature.

3.1.2 Software and data

As mentioned previously, the main software component that was used to create an ANN is the **Tensorflow** [67] python library, which essentially is specific to machine learning models. On top of Tensorflow the API called **Keras** [70] is run, enabling quick experimentation. The reason it is commonly used is that it has very high level API functions, making it easy even for a beginner to build their own model. Passing arguments to those functions is convenient and does not require deep understanding of how ANNs work. For the conversion to SNN a software that is still being tested will be used, Snntoolbox [69]. Not only does Snntoolbox support Tensorflow models, but it also can do other operations before running the SNN (such as weight normalisation) and has a list of results it can print, such as activations of specific neurons, spike trains and others.

The MNIST data includes hand handwritten digits that have been pixelated and centered in 28x28 pixel grey scale images, see figure 3.1. It provides 60000 training examples and 10000 testing examples. Tensorflow has its own function for loading the MNIST data as a multidimensional array. Each image's pixel is encoded as an 8-bit number, with a range of values from 0 up to $2^8 - 1$, and therefore can be normalized by dividing the array by 255.

3.1.3 Hyper-Parameters

In this section, the hyper-parameters will be discussed, specifically their possible values that will be tested and their meaning.

The type of model can be Sequential, Functional API or Model subclassing. The Sequential model is the simplest model and allows exactly one input and one output layer connected in a feed-forward fashion. The number of hidden layers tested, will be either 1 or 2 since the complexity of the problem is not high, and the number of neurons will go up to 110 for the same reason. The activation functions considered for this experiment are the ReLU, Softmax and Sigmoid which are the most widely used activation functions. The loss function examined is the Mean Squared Error. The optimizers taken into

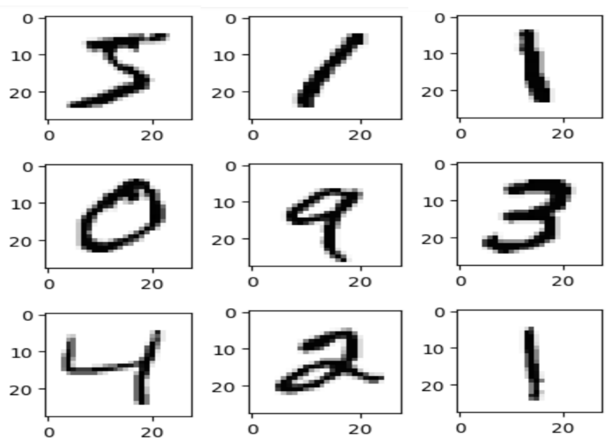

Figure 3.1: *9 examples of what the MNIST data looks like . The x and y axis represent the coordinates of each pixel and can be used to index each of the 784 pixels. The darker the pixel is, the higher its intensity is and the higher its value is in a python array.*

consideration are all of the optimizers mentioned in the introduction. The batch size can be specified during fitting the data and will range from 10 to 150 in steps of 10. The number of epochs considered ranged between 5 and 100 in steps of 5, and is specified during training as well.

As mentioned previously, there are no rules that will tell the optimal values for such parameters. Decision on the range of values had to be made, based on the time provided for the project. More values can be tested if needed for another project, to yield the actual best hyper-parameters.

3.1.4 Main Algorithm

The main algorithm used for finding well performing parameters (but not the best parameters!) reminds of the exhaustive search [2], since there is no intuition yet on choosing the parameters, except for the fact that more neurons allow the model to capture a higher complexity pattern [3]. In this algorithm, an ANN model is initially built with randomly chosen parameters. Then, each parameter gets optimized at a time, by trying all possible values for that parameter and recording the accuracy for each of those values. The value of the parameter that scores the highest accuracy is the one chosen for that parameter. The ANN model updates its parameter. Another parameter is then chosen and optimized in the same manner, using the updated ANN.

1. Initialize array x which stores the possible values of the hyper-parameter

2. for i in x

[2]A technique where all possible combinations are tried.

[3]A lot of neurons can lead to better performance but can also become redundant or decrease the accuracy. There is still no formula to give the optimal number of neurons or layers

   (a) build a model using i, substituting any random values of hyper-parameters with previously calculated best values
   (b) train the model using the training data
   (c) compute accuracy on testing data
   (d) print i with the best testing accuracy

3. Repeat steps 1 and 2 for another hyper-parameter

The order by which the hyper-parameters where tested is

1. Number of hidden layers and their activation functions of the hidden layers and the output layer
2. Number of neurons in hidden layer(s)
3. Optimizer
4. Batch size
5. Number of epochs
6. Loss function

The structural hyper-parameters, such as number of hidden layers and their activation functions will first be tested in a proper exhaustive search. This choice was made based on general engineering problem solving, where usually engineers first realise what a design should include and then the technical decisions come, for instance, first there is the realisation of the addition of a resistor to a specific point in the circuit and then the type of resistor and its value is considered. The number of neurons can also be considered a structural hyper-parameter and therefore is tested immediately after. The reason a proper exhaustive search was not implemented for all hyper-parameters is that this process would be time consuming. Additionally, carrying out the experiment by testing one parameter at a time, can lead towards building intuition on what hyper-parameter values to choose in the future for similar data.

Changing the structure of the ANN (i.e. the number of layers, their activation functions and their neurons) requires building the model again instead of re-training it with those parameters. Changing technical characteristics of the model that are declared during the training of the model, such as the loss function and the optimizer, demands building a new model as well (step 2.a). This is necessary, because if the same model that was trained earlier with different technical parameters, is now trained with new ones, it will continue updating the weights and biases from where it left off, making the last value of such parameters to always have the best performance.

There is no additional work that needs to be done for converting this ANN to an SNN, as the Snntoolbox does that automatically, by reading a model that has been saved as a ".h5" file.

3.1.5 Results

The initial model uses the Adam optimizer, with the Mean Squared Error loss function, 1 epoch, and a default batch size of 32. The accuracy of a 2 hidden layers model and a one hidden layer model was compared, with the two models utilising the same number of neurons (i.e. allowing very close to equal computation power [4]). The one hidden layer model had 100 neurons and the two hidden layer model had 50 neurons in each layer. The number of neurons in the input and output layer is fixed at 784 and 10 neurons respectively, as stated in the introduction. Specifically the accuracy on the test data of the best combination of activation functions in the hidden layers and output layer was compared, with the 2 hidden layer model scoring a 90.1% accuracy while using ReLU in the first hidden layer, and Sigmoid in the 2nd hidden layer and the output layer, and the one hidden layer model using ReLU in both the hidden and output layer scoring 95.4% accuracy.

On the one hidden layer model, the number of neurons were tested, with the range of values being between 20 and 110 in steps of 10, and the results can be found in figure 3.2. The model with 90 hidden layer neurons scored the highest accuracy of 96.5%.

Different optimizers were then tried upon the new model. The recorded accuracy of each optimizer can be found in figure 3.3. The best performing optimizer was the Nadam optimizer with 97.2% accuracy.

Then the different batch sizes were examined and the results are shown in figure 3.4. The highest accuracy was 97.6% and was observed at a batch size of 150. The epochs tested ranged between 5 and 100 and the best observed result was at 20 epochs with an accuracy slightly better than without using epochs at all, at 97.65%.

The final model has one input layer with 784 neurons, one hidden layer with 90 neurons and an output layer of 10 neurons, with the ReLU activation function applied on the input and output neurons. The model utilizes the Nadam optimizer and goes over the data 20 times, updating the weights after every 150 training examples. Due to some unexpected behaviour observed in figure 3.2, another simplistic experiment was performed, but not taken into consideration for the final design. The number of hidden layer neuron were tested between 110 and 210 and the results observed are shown in figure 3.6.

For the SNN, the accuracy over the 10000 testing examples was recorded to be 97.5% which is very close to the corresponding ANN accuracy. An example of how the the network classifies a given image is shown in figure 3.7 with spike trains or in figure 3.8

[4]From figure 1.10 if the 2 hidden neurons were to occupy 2 hidden layers of 1 neuron each, the number of weights and biases would decrease, although the number of activations (which is the biggest part of computation) computed is the same.

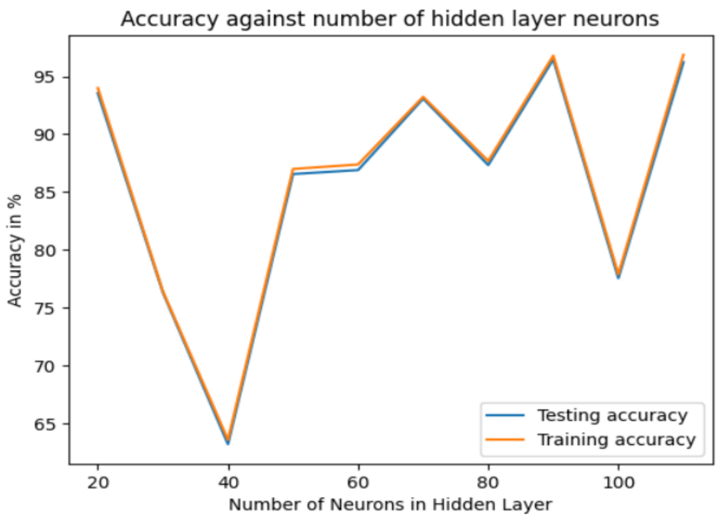

Figure 3.2: *A graph of the accuracy on the training and testing data against models with different number of hidden layer neurons.*

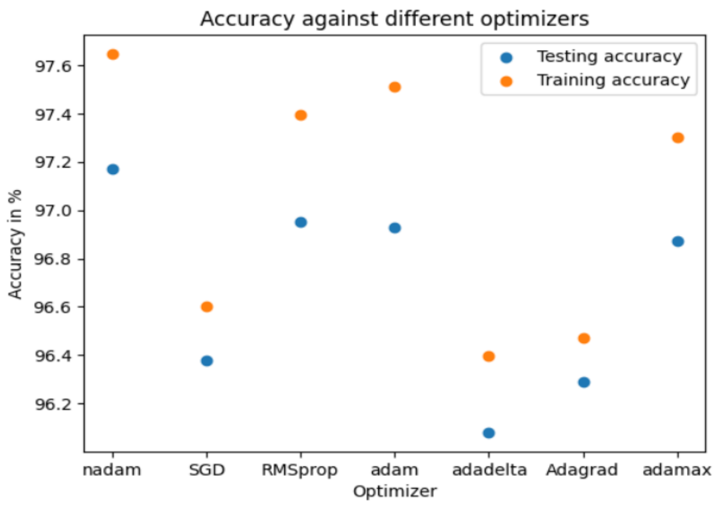

Figure 3.3: *A graph of the accuracy on the training and testing data of different optimizers.*

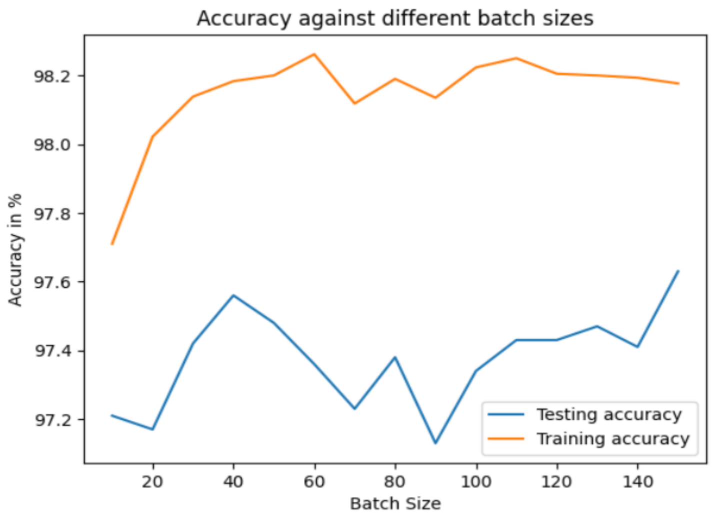

Figure 3.4: *The accuracy on the training and testing data against different batch sizes.*

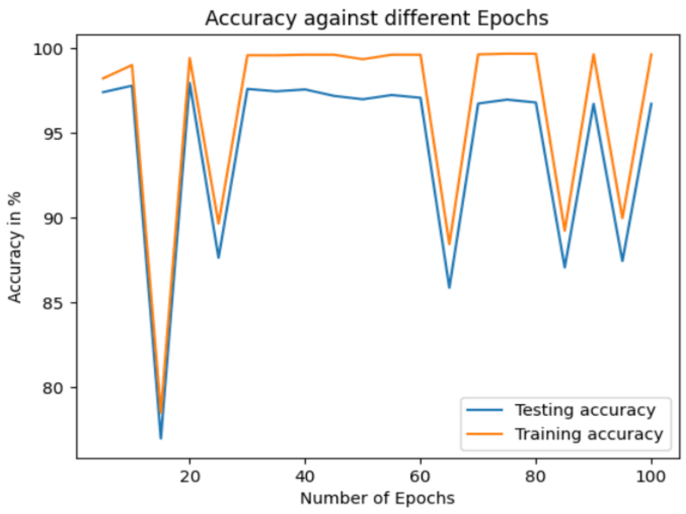

Figure 3.5: *The accuracy on the training and testing data against different number of epochs.*

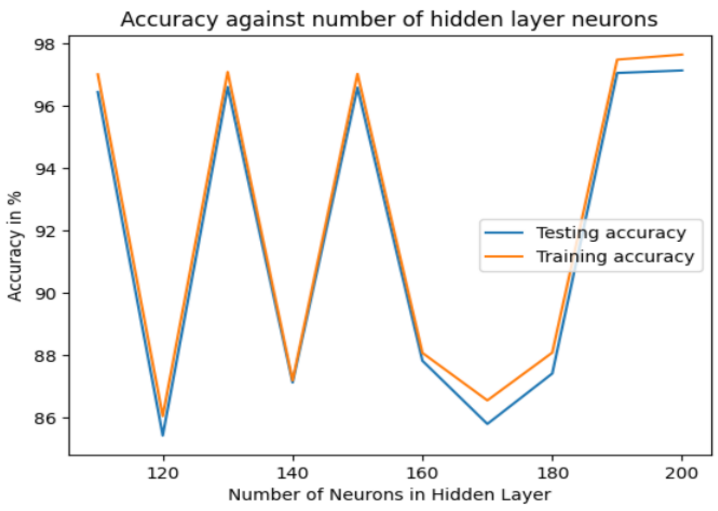

Figure 3.6: *Accuracy on training and testing data against varying number of neurons in the hidden layer.*

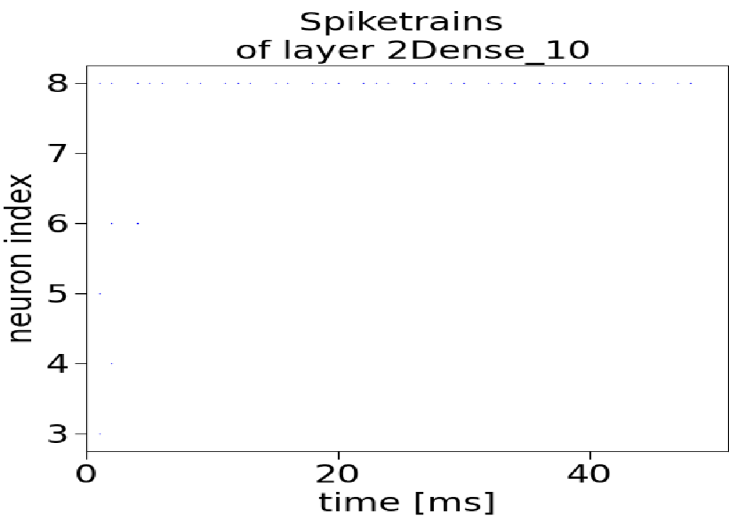

Figure 3.7: *A graph of spikes observed on each of the output layer neurons (y-axis) throughout time (x-axis). Each dot represents a spike observed on a neuron at a specific time. This graph show that for the specific image shown to the network, the class represented 8th neuron (where count starts from 0) has the most spikes and therefore the system will classify the image as an 8.*

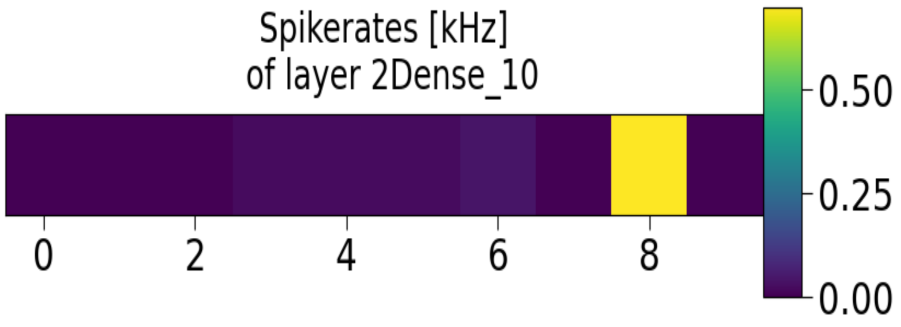

Figure 3.8: *A graph of the frequency of the spikes observed at each output neuron.*

Chapter 4

Discussion and Future Improvements

Discussion

The results obtained result in some doubts about the initial idea of how ANNs work. Firstly, looking at figure 3.2 the 20 hidden neurons model seems to perform relatively well compared to the 90 hidden neurons model and the 110 hidden neurons model. Additionally, the accuracy significantly drops for 40 neurons, meaning that there is probably no continuity between the performance and the number of neurons, as the performance varies a lot for small changes in the number of hidden layer neurons. No conclusions can be made on the performance of the ANN as the number of hidden neurons increase. In figure 3.6 a similar behaviour is observed, where it seems that the performance depends on the number of neurons in a periodic fashion , i.e. every 20 or so neurons from a base number of neurons the performance is high while from another base the performance is low. A similar research on the optimal number of hidden layers and their neurons was conducted, and the main conclusion is that it is challenging to find the optimal values for such topological parameters [71]. A decision on the number of neurons will have to be made depending on the application and its requirements. For instance, a design may be developed to be energy efficient [72], in which case t one may prefer the 20 hidden layer neurons, since there are less activations to be calculated and less weights and biases to be optimized. In this study, power consumption was not of interest, therefore the best accuracy was chosen.

The different optimizers seem to make little difference on the accuracy of the model. The model with the lowest accuracy is the one utilising the Adadelta optimizer with a 96.1% score and the highest accuracy is observed at 97.2% by the Nadam optimizer.

The batch size is one of the factors that determine the training time. Larger batches for training, will lead to less updates of the weights and biases of the models, since more training examples are considered before an update. Additionally, batch training will not have a significant effect on the accuracy of the model, if it is not an extremely large number compared to the number of training examples [73].

One of the common issues in machine learning is overfitting the training data. The number of epochs is a determining factor in overfitting. If the training set accuracy is higher than the testing set accuracy, the model has fit the training data more than desired and cannot perform in "real world" application i.e. testing data. As the improvement of the model using epochs is only 0.05% better than without using epochs, one can argue that such a computation is worth doing at all, for an application that takes into consideration the training time. The small difference observed could also be a result of there not being lot of room for improvement, since the accuracy of the model is already high. Supposing a different order of fixing the hyper-parameters where the epochs is fixed at an earlier stage, the impact of epochs could be larger and the same conclusion can be drawn for the impact of other parameters. Looking at the graph in 3.5 a similar behaviour is observed, to the accuracy of the model as the number of hidden neurons changes, which was discussed earlier. The accuracy does not seem to vary monotonically with the number of epochs.

Future Improvements

While the model's final accuracy can be satisfactory for some applications, there are still improvements that can be made depending on the application. As it was mentioned previously, the method developed to build the ANN will probably not achieve the best performance possible by an ANN. A proper exhaustive search over a larger range of possible values is what can lead to best accuracy. However, the latter may lead to a significant increase in training time. For instance, iterating over the epochs between 5 and 100 in steps of 5, took an average computer a bit more than 51 minutes. If all epochs were to be tested for all the possible combinations of the other hyper-parameters, the training would take several days. An additional improvement in this project is to implement the SNN on a neuromorphic platform and measure other performance factors such as energy consumption and classification time, which due to time constraints was not possible.

Chapter 5

Conclusion

The low energy consumption and high efficiency of the human brain in recognition and classification tasks has motivated scientists to develop machine learning methods that mimic its behaviour. Artificial Neural Networks and Spiking Neural Networks are the main instances of such machine learning methods, where both have been proven to achieve promising results in tasks including image or speech recognition, classification and decision making in fields such as medicine finance and others.

The main computational unit of an ANN is the artificial neuron whose input is calculated in a summative way of other neurons' outputs that have been multiplied by a weight, and its output is determined by the chosen activation function and a bias which serves a fixing purpose of the calculated input. The learning method of an ANN is the back-propagation method which involves calculating the gradient of a chosen error function with respect to the different weights and biases throughout all neuron connections and updating the weights and biases following this gradient, to minimize the error function.

A Spiking Neural Network can be trained from an ANN (shadow training) or individually using back-propagation. SNN is compose of spiking neurons instead of the artificial neurons used in ANNs. Spiking neurons introduce more concepts of biological neurons, such as the decaying membrane potential and the timing of the received signal and their computation involves the build up of that membrane potential to a point where the spiking neuron crosses a threshold voltage to go through and action potential and send a signal to other neurons. The strength of that signal depends on the weight of the connection between the transmitting and receiving spiking neuron.

In this study, an ANN was developed using Tensorflow, testing the performance of different values for some parameters in an exhaustive search manner, to fit handwritten digits of the MNIST dataset. Those parameters were the number of hidden layers, the optimization method, the number of epochs, the batch size and different activation functions applied to each layer. The results confirmed other papers' finding such as the fact that there is correlation observed between the number of hidden neurons and the complexity of the problem or the dimensionality of the input data and that ANNs have a high performance in classification tasks. The final accuracy of the ANN model developed was 97.65%. The weights and biases of the model were then transferred into an SNN using the Snntoolbox software and the SNN model achieved an accuracy on the testing data of 97.5% which relatively high to the ANN performance.

Acknowledgements

This project would not have reached its extend without the assistance of Shouyu Xie, who was attentive and helpful in understanding the concept of ANNs and SNNs.

References

[1] Kristan, W. B.: 'Early evolution of neurons', in: *Current Biology* 26.20 (2016), R949–R954.

[2] Chrysafides SM Bordes SJ, S.: 'Physiology , Resting Potential', NCBI Bookshelf, 2022, `https://www.ncbi.nlm.nih.gov/books/NBK538338/`, accessed 8th Apr. 2023, website owner StatPearls Publishing LLC.

[3] Bean, B. P.: 'The action potential in mammalian central neurons', in: *Nature Reviews Neuroscience* 8.6 (2007), pp. 451–465.

[4] Stuart, G., Spruston, N., Sakmann, B. and Häusser, M.: 'Action potential initiation and backpropagation in neurons of the mammalian CNS', in: *Trends in neurosciences* 20.3 (1997), pp. 125–131.

[5] Platkiewicz, J. and Brette, R.: 'A threshold equation for action potential initiation', in: *PLoS computational biology* 6.7 (2010), e1000850.

[6] Memming P. Sohan S., A. L.: 'Kernel Methods on Spike Train Space for Neuroscience: A Tutorial', IEEE Signal Processing Magazine 30(4), Feb. 2013, `https://www.researchgate.net/publication/235702794_Kernel_Methods_on_Spike_Train_Space_for_Neuroscience_A_Tutorial`, accessed 9th Apr. 2023.

[7] MD, J. V.: 'Action Potential', 28th Sept. 2022, `https://www.kenhub.com/en/library/anatomy/action-potential`, accessed 9th Apr. 2023.

[8] Jain, A., Mao, J. and Mohiuddin, K.: 'Artificial neural networks: a tutorial', in: *Computer* 29.3 (1996), pp. 31–44, DOI: `10.1109/2.485891`.

[9] Larsen, R. S. and Sjöström, P. J.: 'Synapse-type-specific plasticity in local circuits', in: *Current opinion in neurobiology* 35 (2015), pp. 127–135.

[10] Kitamura, A., Marszalec, W., Yeh, J. Z. and Narahashi, T.: 'Effects of halothane and propofol on excitatory and inhibitory synaptic transmission in rat cortical neurons', in: *Journal of Pharmacology and Experimental Therapeutics* 304.1 (2003), pp. 162–171.

[11] Krenker, A., Bešter, J. and Kos, A.: 'Introduction to the artificial neural networks', in: *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech* (2011), pp. 1–18.

[12] Harmon, L. D.: 'Artificial neuron', in: *Science* 129.3354 (1959), pp. 962–963.

[13] Krogh, A.: 'What are artificial neural networks?', in: *Nature biotechnology* 26.2 (2008), pp. 195–197.

[14] Ramachandran, P., Zoph, B. and Le, Q. V.: 'Searching for activation functions', in: *arXiv preprint arXiv:1710.05941* (2017).

[15] Narayan, S.: 'The generalized sigmoid activation function: Competitive supervised learning', in: *Information sciences* 99.1-2 (1997), pp. 69–82.

[16] W., C.-F.: 'The Vanishing Gradient Descend Problem. The Problem, Its Causes, Its Significance, and its solutions', 8th Jan. 2019, `https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484`, accessed 10th Apr. 2023.

[17] Dunne, R. A. and Campbell, N. A.: 'On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function', in: *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, vol. 181, Citeseer, p. 185.

[18] Svozil, D., Kvasnicka, V. and Pospichal, J.: 'Introduction to multi-layer feed-forward neural networks', in: *Chemometrics and intelligent laboratory systems* 39.1 (1997), pp. 43–62.

[19] Medsker, L. and Jain, L. C.: 'Recurrent neural networks: design and applications', (CRC press, 1999).

[20] Sazlı, M. H.: 'A brief review of feed-forward neural networks', in: *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering* (2006), pp. 0 - 0, DOI: `10.1501/commua1-2\_0000000026`.

[21] Zupan, J.: 'Introduction to artificial neural network (ANN) methods: what they are and how to use them', in: *Acta Chimica Slovenica* 41 (1994), pp. 327–327.

[22] Uzair, M. and Jamil, N.: 'Effects of hidden layers on the efficiency of neural networks', in: *2020 IEEE 23rd international multitopic conference (INMIC)*, IEEE, pp. 1–6.

[23] Shanmuganathan, S.: 'Artificial neural network modelling: An introduction', (Springer, 2016).

[24] 'Supervised learning', IBM, `https://www.ibm.com/uk-en/topics/supervised-learning`, accessed 12th Apr. 2023.

[25] 'Unsupervised learning', IBM, `https://www.ibm.com/uk-en/topics/unsupervised-learning`, accessed 12th Apr. 2023.

[26] Wasserman Philip D., T.: 'Neural networks. II. What are they and why is everybody so interested in them now?.', in: (1988).

[27] Murphy, K.: 'Naive bayes classifiers.', in: (2006), 18(60), pp1-8.

[28] Isobe, T., Feigelson, E. D., Akritas, M. G. and Babu, G. J.: 'Linear regression in astronomy.', in: *Astrophysical Journal, Part 1 (ISSN 0004-637X), vol. 364, Nov. 20, 1990, p. 104-113. Research supported by NASA.* 364 (1990), pp. 104–113.

[29] Nasteski, V.: 'An overview of the supervised machine learning methods', in: *Horizons. b* 4 (2017), pp. 51–62.

[30] Hartigan, J. A. and Wong, M. A.: ‘Algorithm AS 136: A k-means clustering algorithm’, in: *Journal of the royal statistical society. series c (applied statistics)* 28.1 (1979), pp. 100–108.

[31] Ben-Israel, A. and Iyigun, C.: ‘Probabilistic d-clustering’, in: *Journal of Classification* 25 (2008), pp. 5–26.

[32] ‘Supervised learning’, IBM, `https://dataplatform.cloud.ibm.com/docs/content/wsj/model/wos-quality-mean-abs-error.html`, accessed 12th Apr. 2023.

[33] ‘Supervised learning’, IBM, `https://www.ibm.com/docs/en/cloud-paks/cp-data/3.5.0?topic=overview-mean-squared-error`, accessed 12th Apr. 2023.

[34] Rusiecki, A.: ‘Trimmed categorical cross-entropy for deep learning with label noise’, in: *Electronics Letters* 55.6 (2019), pp. 319–320.

[35] ‘Supervised learning’, IBM, `https://www.ibm.com/uk-en/topics/gradient-descentr`, accessed 13th Apr. 2023.

[36] Xiaosong, D., Popovic, D. and Schulz-Ekloff, G.: ‘Oscillation-resisting in the learning of backpropagation neural networks’, in: *IFAC Proceedings Volumes* 28.5 (1995), pp. 21–25.

[37] Ruder, S.: ‘An overview of gradient descent optimization algorithms’, in: *arXiv preprint arXiv:1609.04747* (2016).

[38] Amari, S.-i.: ‘Backpropagation and stochastic gradient descent method’, in: *Neurocomputing* 5.4-5 (1993), pp. 185–196.

[39] Hinton, G., Srivastava, N. and Swersky, K.: ‘Neural networks for machine learning lecture 6a overview of mini-batch gradient descent’, in: *Cited on* 14.8 (2012), p. 2.

[40] Haji, S. H. and Abdulazeez, A. M.: ‘Comparison of optimization techniques based on gradient descent algorithm: A review’, in: *PalArch’s Journal of Archaeology of Egypt/Egyptology* 18.4 (2021), pp. 2715–2743.

[41] Qian, N.: ‘On the momentum term in gradient descent learning algorithms’, in: *Neural Networks* 12.1 (1999), pp. 145–151, ISSN: 0893-6080, DOI: `https://doi.org/10.1016/S0893-6080(98)00116-6`, URL: `https://www.sciencedirect.com/science/article/pii/S0893608098001166`.

[42] Défossez, A., Bottou, L., Bach, F. and Usunier, N.: ‘A simple convergence proof of adam and adagrad’, in: *arXiv preprint arXiv:2003.02395* (2020).

[43] Lydia, A. and Francis, S.: ‘Adagrad—an optimizer for stochastic gradient descent’, in: *Int. J. Inf. Comput. Sci* 6.5 (2019), pp. 566–568.

[44] Zeiler, M. D.: ‘Adadelta: an adaptive learning rate method’, in: *arXiv preprint arXiv:1212.5701* (2012).

[45] Bae, K., Ryu, H. and Shin, H.: ‘Does Adam optimizer keep close to the optimal point?’, in: *arXiv preprint arXiv:1911.00289* (2019).

[46] Zeng, X., Zhang, Z. and Wang, D.: ‘AdaMax Online Training for Speech Recognition’, in: *2016* (2016).

[47] Halgamuge, M. N., Daminda, E. and Nirmalathas, A.: ‘Best optimizer selection for predicting bushfire occurrences using deep learning’, in: *Natural Hazards* 103.1 (2020), pp. 845–860.

[48] Rice, K. L., Bhuiyan, M. A., Taha, T. M., Vutsinas, C. N. and Smith, M. C.: ‘FPGA implementation of Izhikevich spiking neural networks for character recognition’, in: *2009 International Conference on Reconfigurable Computing and FPGAs*, IEEE, pp. 451–456.

[49] Pani, D., Meloni, P., Tuveri, G., Palumbo, F., Massobrio, P. and Raffo, L.: ‘An FPGA platform for real-time simulation of spiking neuronal networks’, in: *Frontiers in neuroscience* 11 (2017), p. 90.

[50] Eshraghian, J. K., Ward, M., Neftci, E. et al.: ‘Training spiking neural networks using lessons from deep learning’, in: *arXiv preprint arXiv:2109.12894* (2021).

[51] Gerstner, W. and Kistler, W. M.: ‘Spiking Neuron Models: Single Neurons, Populations, Plasticity’, (Cambridge University Press, 2002), DOI: `10.1017/CBO9780511815706`.

[52] Javanshir, A., Nguyen, T. T., Mahmud, M. P. and Kouzani, A. Z.: ‘Advancements in Algorithms and Neuromorphic Hardware for Spiking Neural Networks’, in: *Neural Computation* 34.6 (2022), pp. 1289–1328.

[53] Han, J., Li, Z., Zheng, W. and Zhang, Y.: ‘Hardware implementation of spiking neural networks on FPGA’, in: *Tsinghua Science and Technology* 25.4 (2020), pp. 479–486.

[54] Furber, S. B., Galluppi, F., Temple, S. and Plana, L. A.: ‘The spinnaker project’, in: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665.

[55] University of Manchester, `http://apt.cs.manchester.ac.uk/projects/SpiNNaker/hardware/`, accessed 2nd May 2023.

[56] Huang, Y., Kangas, L. J. and Rasco, B. A.: ‘Applications of artificial neural networks (ANNs) in food science’, in: *Critical reviews in food science and nutrition* 47.2 (2007), pp. 113–126.

[57] Hinton, G., LeCun, Y. and Bengio, Y.: ‘Deep learning’, in: *Nature* 521.7553 (2015), pp. 436–444.

[58] Aimone, J. B., Ho, Y., Parekh, O. et al.: ‘Provable advantages for graph algorithms in spiking neural networks’, in: *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 35–47.

[59] Baxt, W. G.: ‘Application of artificial neural networks to clinical medicine’, in: *The lancet* 346.8983 (1995), pp. 1135–1138.

[60] Dase, R. and Pawar, D.: ‘Application of Artificial Neural Network for stock market predictions: A review of literature’, in: *International Journal of Machine Intelligence* 2.2 (2010), pp. 14–17.

[61] Vui, C. S., Soon, G. K., On, C. K., Alfred, R. and Anthony, P.: ‘A review of stock market prediction with Artificial neural network (ANN)’, in: *2013 IEEE international conference on control system, computing and engineering*, IEEE, pp. 477–482.

[62] Anagnostou, T., Remzi, M., Lykourinas, M. and Djavan, B.: ‘Artificial neural networks for decision-making in urologic oncology’, in: *European urology* 43.6 (2003), pp. 596–603.

[63] Ponulak, F. and Kasinski, A.: ‘Introduction to spiking neural networks: Information processing, learning and applications’, in: *Acta neurobiologiae experimentalis* 71.4 (2011), pp. 409–433.

[64] Sengupta, A., Ye, Y., Wang, R., Liu, C. and Roy, K.: ‘Going deeper in spiking neural networks: VGG and residual architectures’, in: *Frontiers in neuroscience* 13 (2019), p. 95.

[65] Bing, Z., Meschede, C., Röhrbein, F., Huang, K. and Knoll, A. C.: ‘A survey of robotics control based on learning-inspired spiking neural networks’, in: *Frontiers in neurorobotics* 12 (2018), p. 35.

[66] Nichols, E., McDaid, L. and Siddique, N.: ‘Case study on a self-organizing spiking neural network for robot navigation’, in: *International Journal of Neural Systems* 20.06 (2010), pp. 501–508.

[67] `https://www.tensorflow.org/`, accessed 3rd May 2023.

[68] Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C. and Pfeiffer, M.: ‘Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing’, in: *2015 International joint conference on neural networks (IJCNN)*, ieee, pp. 1–8.

[69] `https://snntoolbox.readthedocs.io/en/latest/`, accessed 3rd May 2023.

[70] `https://keras.io/api/`, accessed 2nd May 2023.

[71] Karsoliya, S.: ‘Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture’, in: *International Journal of Engineering Trends and Technology* 3.6 (2012), pp. 714–717.

[72] Tso, G. K. and Yau, K. K.: ‘Predicting electricity energy consumption: A comparison of regression analysis, decision tree and neural networks’, in: *Energy* 32.9 (2007), pp. 1761–1768.

[73] Radiuk, P. M.: ‘Impact of training set batch size on the performance of convolutional neural networks for diverse datasets’, in: *Information Technology and Management Science* 20.1 (2017), pp. 20–24.