COMP5329 - Deep Learning Project 1

# Multi-class Classification

# through Multi-layer Neural Network

**---A comparative study of different models using Python with codes from scratch**

**Quang Trung NGUYEN 470518197**

**Youyou CHEN 308012798**

**Zhehao ZHANG 470490653**

## Table of Contents

**1. Introduction** ..... **2**
**2. An Overview of Neural Network Architecture** ..... **3**
2.1. Multi-layer perceptron (MLP) ..... 3
2.2. Activation Functions ..... 4
2.3. Loss Function ..... 5
2.4. Back Propagation ..... 6
**3. Exploratory Data Analysis and Pre-processing** ..... **6**
**4. Model and Experiment Design** ..... **7**
4.1. Initialization ..... 7
4.2. Gradient Descent Optimization ..... 8
4.2.1. Stochastic ..... 8
4.2.2. Momentum ..... 9
4.2.3. Adam ..... 10
4.2.4. Dropout ..... 11
**5. Proposed Classification Method & Extensive Analysis** ..... **11**
5.1. Stochastic (SGD) Only ..... 12
5.2. Momentum ..... 13
5.3. Adam ..... 13
5.4. Dropout ..... 14
5.5. Summary of Experiment Results ..... 14
5.6. Proposed Classification Model ..... 17
**6. Conclusion and Future Works** ..... **17**
**References** ..... **18**
**APPENDIX A – How to run the codes** ..... **20**
**APPENDIX B – Predicted Outputs of Test Data** ..... **20**
**APPENDIX A – Hardware and Software Specifications** ..... **20**

## 1. Introduction

The purpose of this study is to complete a multi-class classification task on the provided dataset using multi-layer neural network system. The training dataset provided for this task is a labelled multivariate dataset consisting of 128 features and 60,000 samples. There are a total of 10 unique classes in the dataset with a balanced distribution of 6,000 samples in each class. The testing dataset (unlabeled) consists of 10,000 samples and same number of attributes as the training dataset. The goal is to use the labelled dataset to construct a neural network classifier to produce the most accurate and efficient prediction on the labels of the testing dataset.

Multiclass neural network classification usually involves building neural networks that map the input feature space to the output space containing more than two classes (Ou et al., 2004). Essentially, two general neural network structures could be used to categorize multiple classes. The first approach builds a model consisting of several independent binary classifiers (one binary classifier for each possible outcome). During training, the model runs through the sequence of binary classifiers, each of which completes a separate binary classification task. One advantage of using this technique is that different features can be applied to train different classifiers (Murphey et al., 2002). However, the model becomes progressively inefficient as the number of possible classes increases. To overcome this, a model in which each output node represents a different class could be built. Decision boundaries are usually sharper than the first approach with the use of Softmax activation function (which will be introduced later in this study) (Ou et al., 2004). In order to avoid uncertainty in classification boundary, the second approach is adopted in this study.

In this report, we will firstly explain the brief principle of building multiclass neural network models. Then we will continue to demonstrate how we preprocess the underlying dataset accordingly and how a few different classifier structures were tested and applied to complete this task. Different methods for parameter initialization and optimization were also experimented to seek for best possible classifier performance. It noted that the instructions on how to run the final codes are in the appendix.

We constructed all of the neural network systems in this study using Back Propagation (BP) as the learning algorithm for parameter tuning. We will show that different architectures, modelling approaches and implementations with the same neural learning algorithm can give us different performances in terms of model complexity, running time, testing accuracy, and degree of generalization. A comprehensive comparison of these different methods with respects of criteria mentioned above was also produced. Eventually, we choose Stochastic Gradient Descent with Momentum as our proposed classification model as it gives us the optimal accuracy on training data and also has a feasible runtime.

## 2. An Overview of Neural Network Architecture

The basic computation unit in a neural network is the neuron, sometimes referred to as a node or unit. It receives information from some other units or sources and computes an output. The connection pathways provide the means of transferring the information between any two consecutive layers (Fumo, 2017). For example, the connection between the input neuron i and an output neuron j can be partly represented by the assigned weight $w_{ji}$. The node then applies a function (also known as the activation function) to the weighted sum of its inputs and a bias (b) term to increase the flexibility of the model to fit the data. The intuitive purpose of imposing the activation function to the node will be introduced later in section 2.2 of this report. Several forms of different activation function will also be presented. The feedforward neural network is considered the first and simplest type of artificial neural network which contains multiple neurons (nodes) arranged in layers (Sharma et al., 2013). The information flows successively only in the forward direction, i.e. from input layer to hidden layer, from hidden layer to output layer. This feed-forward model is also the type of network structure we adopted for the purpose of this study. A flowchart of how typical network architecture works is plotted with 'draw.io' for demonstration in Figure 1 below.

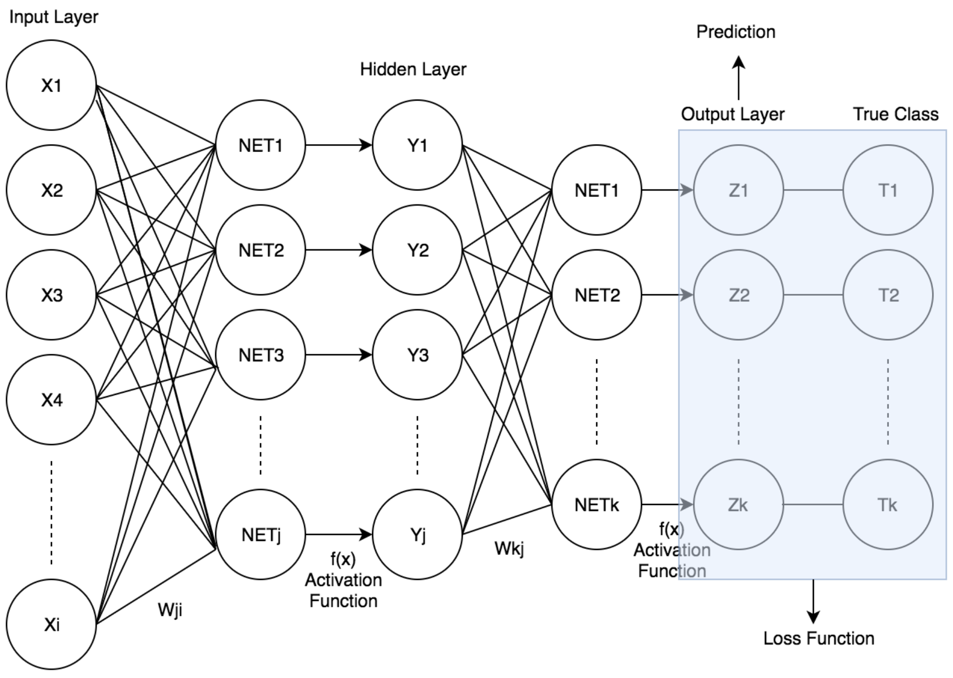

**Figure 1. Neural Network Architecture**

### 2.1. Multi-layer perceptron (MLP)

Multi-layer perceptron consists of multiple layers of computational units. A layer consists of a block of nodes. In most real-life classification cases, the data presented is not linearly separable and one of the most important advantages of MLP is that it is capable of learning non-linear representations and complex relationships (Sharma et al., 2013). In addition, MLP does not impose any restrictions on the input variables or input (e.g. assumptions on the distribution of data) (Sharma et al., 2013). In a complete MLP system, there are 3 types of layers:

- Input Nodes (input layer): Input layer receives all the external inputs to the network. No calculation is completed within this layer whereas they just pass the information to the next layer;

- Hidden nodes (hidden layer): Hidden layers are where intermediate processing or computation is completed, they perform computations and then transfer the weights (signals or information) from the input layer to the following layer (another hidden layer or to the output layer). It is possible to have a neural network without more than one hidden layer. Generally, the more hidden layers a neural network has, the more complicated the model usually is (Shamseldin et al, 2002);

- Output Nodes (output layer): Computation of output layer is similar to the one of hidden layers except that it usually uses an activation function that maps to the desired output format (e.g. Softmax for multi-class classification). The output values are then used to calculate the ultimate loss function for parameter and model optimization. Some generally used loss function will also be introduced in section 2.3.

### 2.2. Activation Functions

As introduced earlier, linear transformation on the input is completed through weights and biases (see equation 1 below). On the other hand, the non-linear transformation is achieved through activation functions (see equation 2 below) so that more complex tasks could be solved. The information moves from the previous layer and the subsequent layer is then complete. Activation function is an extremely important element of the artificial neural network systems because the output value produced by the activation function could help decide whether the output node should be considered as “activated” or not:

$$Net = \sum \quad (Input * Weight) + Bias \qquad (1)$$

$$Output = Activation\,function\,(Net) \qquad (2)$$

The transformed output neurons then become the input neurons to the next layer. A neural network without an activation function is essentially just a linear regression model no matter how many hidden layers there are. In addition, activation functions also make the back-propagation possible since the gradients are supplied along with the loss function to update the weights and biases in learning algorithm. Without the continuously differentiable nature of the activation functions, this would be impossible (Xu, 2018).

Some of the popular activation functions used in neural network systems are: binary step function, Sigmoid function, Tanh function, rectified linear unit (ReLU) function, leaky ReLU function and Softmax function. ReLU function is a preferred activation function and is used in most cases comparing with Sigmoid and Tanh functions (Xu, 2018). Also, the case of dead neurons in our networks could be avoided by using the leaky ReLU function. To produce for relative probabilistic values in the output layer in multi-class classification tasks, Softmax activation function should be used in the final layer. This study will test 3 different kinds of activation functions in the interim layers with Softmax function in the final layer. A summary of the 4 functions adopted in this study could be found in table 1 below. It should be noted that Softmax function is not a function of a single node but depends upon the relative values of other nodes from the same layer. The derivative of each function is also calculated and presented for the purpose of backpropagation in the learning phase.

| Name | Equation | Derivative | Output Range |
|---|---|---|---|
| *Tanh* | $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | $f'(x) = 1 - f(x)^2$ | $(-1,1)$ |
| *ReLU* | $f(x) = 0 \;\; for\, x < 0$ <br> $= x \;\; for\, x \geq 0$ | $f'(x) = 0 \;\; for\, x < 0$ <br> $= 1 \;\; for\, x \geq 0$ | $[0, \infty)$ |
| *Leaky ReLU* | $f(\alpha, x) = \alpha x \;\; for\, x < 0$ <br> $= x \;\; for\, x \geq 0$ | $f'(\alpha, x) = \alpha \;\; for\, x < 0$ <br> $= 1 \;\; for\, x \geq 0$ | $(-\infty, \infty)$ |
| *Softmax* | $f(x_i) = \frac{e^{x_i} - \max(e^{x_i})}{\sum_{j=1}^{J}(e^{x_j} - \max(e^{x_j}))}$ <br> $for\; i = 1,2,\ldots,J$ | $f'(x_i) = f(x_i) \times (1 - f(x_i)$ | $(0,1)$ |

**Table 1. Summary of Different Activation Functions**

## 2.3. Loss Function

We could deduce from above that applying Softmax function normalizes the values in the array of final layer ($z_k$) to the scale between 0 and 1. Vectorising the true class label (numeric) to a true class array ($t_k$) should be performed so that the two arrays are of the same dimension. The true class array ($t_k$) should be constructed in such way that $t_k$[i] = 1 if i is the class index/value of the true class and $t_k$[i] = 0 elsewhere. To quantify the differences/distance between these two arrays into a simple number, the cross-entropy function could be applied (equation 3). By imposing the loss function on the final output layer is also a necessary step for backpropagation so that the derivative of the cross-entropy function in respect to $z_k$ is also presented in equation 4.

$$L(t,z) = -\sum_{k=1}^{J} t_k log(z_k) \qquad (3)$$

$$L'(t,z) = -\frac{t_k}{z_k} \qquad (4)$$

Also, the sum of predicted outputs always equals to 1 when Softmax is applied. The final predicted class is usually the index of output nodes that has the maximum probability value.

### 2.4. Back Propagation

The learning process (Back Propagation in this study) is the algorithm which modifies or updates the parameters of the neural network for a given input so that the loss function (introduced in section 2.3) is minimized. Finding the minimal value in the loss function is completed through gradient descent is neural network system (Xu, 2018). Details of different gradient descent approaches will be discussed in details in section 4.2.

## 3. Exploratory Data Analysis and Pre-processing

As introduced, the training dataset provided for this task is a labelled dataset consisting of 128 attributes and 60,000 tuples. 10 unique classes are presented with a balanced distribution in each class. Size of the testing dataset is of 10,000 samples and of the same feature dimension as the training dataset:

| Dataset Name | No. of Features | No. of Samples | No. of Class | Labelled/ Unlabeled |
|---|---|---|---|---|
| Training set | 128 | 60000 | 10 | Labelled |
| Testing set | 128 | 10000 | 10 | Unlabeled |

**Table 2. Data Summary**

Considering the different scales for different features, it is generally required to perform features scaling or standardizing before constructing the neural network model to reduce computational cost. Another major advantage of normalizing the data is that different scales could place a certain degree of limitation on the learning ability of the network (Scharth, 2017). Hence, we standardized the training data such that input data are with mean of 0 and standard deviation of 1 with respect to each feature space. It is noted that standardizing target label is unnecessary for the purpose of calculating cross entropy loss as introduced in section 2.2.

To evaluate the performance, the labelled dataset is split into a training set containing 90% of the training samples and a validation set containing the other 10%. Because of the balanced distribution in each class, we split the dataset into 2 subsets randomly. The prediction accuracy on the validation set could be used as a proxy indicator of the ultimate classifier performance.

## 4. Model and Experiment Design

The specific structure of neural network and some basic parameters initialization techniques for this task will be described firstly in this section, followed by discussions on activation functions and the loss function used. Next, regularization method will be introduced, and optimization techniques will be described at the end. The flowchart below can briefly describe how the brief implementation cycle follows:

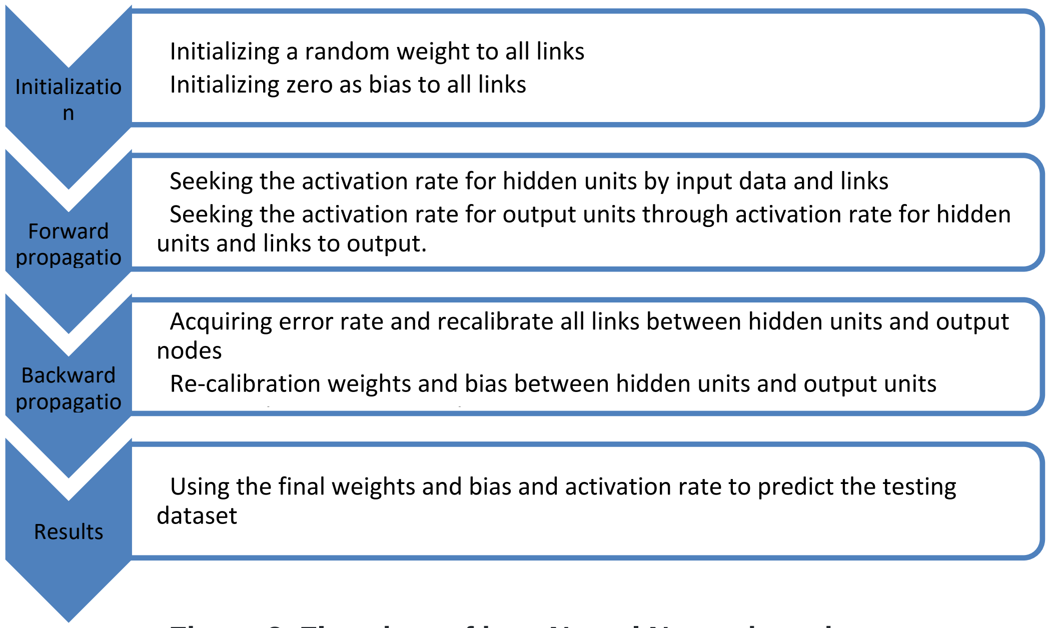
Initializatio n
Initializing a random weight to all links
Initializing zero as bias to all links
Forward propagatio
Seeking the activation rate for hidden units by input data and links
Seeking the activation rate for output units through activation rate for hidden units and links to output.
Backward propagatio
Acquiring error rate and recalibrate all links between hidden units and output nodes
Re-calibration weights and bias between hidden units and output units
Results
Using the final weights and bias and activation rate to predict the testing dataset

**Figure 2. Flowchart of how Neural Network works**

### 4.1. Initialization

As introduced in section 2 of this study, there are two main components in multilayers neural network architectures; one is processing elements (neurons). The other is links, which used for connecting neurons which locate in different layers, including the parameters such as weight and bias associated with each link.

There are two vital parameters need to be initialized first as described above, parameters weight and bias respectively. The appropriate initialization is essential for a neural network to function properly, which can contribute to acquiring an acceptable time for the model to converge and avoids a situation where the value of the loss function becomes unreasonable after thousands of iterations (Joshi, 2018). Hence, the team developed 2 kinds of weight initialization methods for different case according to literature researches. The first one is suitable for all activation functions expect ReLU function. The methodology is to initialize small random number that follows a uniform distribution for the sake of activation variances and back-propagated gradients variance when one moves up or down in network (Glorot & Bengio, 2010).

The distribution can be expressed by following formula, known as normalized initialization (Glorot & Bengio, 2010):

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\right] \text{ (Glorot \& Bengio, 2010)}$$

Where $n_j$ and $n_{j+1}$ represent input and output units respectively.

On the other hand, another uniform distribution was used for ReLU function only, because the Xavier initialization (He, et al, 2015) method does not work well on ReLU activation function. Thus, He and colleagues (2015) have created a replacement method that can be expressed as follow:

$$W \sim U\left[-\frac{\sqrt{2}}{\sqrt{n_j}}, \frac{\sqrt{2}}{\sqrt{n_j}}\right] \text{ (He, et al, 2015)}$$

With regard to initialization of bias, zero can be an acceptable start point (Xu, 2018).

### 4.2. Gradient Descent Optimization

Gradient descent, known as batch gradient descent, is one of most common algorithms to acquire weight update in the direction of negative gradient for each step of (5).

$$W^{(\tau+1)} = W^{(\tau)} - \eta \nabla E(W^{(\tau)}) \quad (5)$$

Where $\eta > 0$ is learning rate, the weight is updated through formula (5) after each iteration until the convergence criteria meet. E() is overall error function.

However, this traditional method has been considered as a poor method such as it can be very slow on a huge dataset, although it can guarantee the local minima (Nasrabadi, 2007). Thus, three improved versions of gradient descent optimization means, stochastic gradient descent, stochastic gradient descent with momentum and stochastic gradient descent with Adam will be discussed and applied to this project.

#### 4.2.1. Stochastic

It might be unnecessary to run a gradient-based algorithm thousands of times in order to find the local minima (Nasrabadi, 2007), especially for a large dataset in this case. The methodology for Stochastic gradient descent (SGD) is updating weights via each training example through equation (2) at a time, which is the main different step compared to batch gradient descent. The starting off data point is different for each time, and comparing the results on another validation dataset (Ruder, 2016). Hence, SGD has an advantage over gradient descent with regard to time of convergence although it may not be guaranteed to find even the local minima in a reasonable time (Goodfellow et al, 2016).

$$W(\tau+1) = W(\tau) - \eta \nabla E\, n(W(\tau)) \qquad (6)$$

Where $\eta > 0$ is learning rate, and En () is error produced by the model when each training example is fed.

Put it simply, the major strength for SGD over the batch gradient descent is computational cost. Batch gradient descent may need double the time than SGD (Nasrabadi, 2007), which is also the reason that SGD has been chosen for this case. Additionally, SGD leads to more stable convergence since it decreases the variance when updating the parameters.

Nevertheless, choosing an appropriate learning rate for SGD is a challenge. The learning rate for SGD is usually smaller than batch gradient descent because of more variance used in updating parameters. Thus, starting off with a small enough value with enough epochs is the method suggested. Then we could slowly vary the rate through trials and error and eventually pick an appropriate value. The experiment results will be presented in the next section.

### 4.2.2. Momentum

Although SGD is a very popular optimization method, learning can be sometimes still slow (Goodfellow et al, 2016). To make learning faster, the method of momentum (Polyak,1964) has been proposed. The idea of momentum is through accumulating the past gradient by exponential moving average and moving to the corresponding direction (Goodfellow et al, 2016). The momentum introduced a new parameter - velocity, which measures the speed and direction that parameter move to. Mathematically, adding one fraction γ into the past update weight to the current update weight (as in equation 7):

$$W^{(\tau+1)} = \gamma W^{(\tau)} - \eta \nabla E_n(W^{(\tau)}) \qquad (7)$$

Where $\eta > 0$ is learning rate, and En () is error produced by the model when each training example is fed. γ is velocity.

The following pseudo-code shows how SGD with momentum works, and the detail implementation can be viewed in codes file.

```
SGD with Momentum
Initialisation: velocity v, parameter $\theta$
Require: learning rate $\eta$, momentum $\alpha$
**While** stop convergence criterion not met **do**:
    Compute gradient estimate:  gradient (g)
    Compute velocity update: $v \leftarrow \alpha\theta - \eta g$
    Compute parameter update: $\theta \leftarrow \theta + v$
```

(Adapted from 'Deep Learning' (Goodfellow, 2016, p. 311))

It has been proven that momentum is able to accelerate convergence to a local minimum by Polyak (1964), which is the main benefit to employ momentum techniques to the neural network model.

### 4.2.3. Adam

Adaptive Moment Estimation (Adam) (Kingma & Ba, 2014) is another adaptive learning rate method to update parameters. It takes both advantages of Adgrad and RMSprop which are two prevalent methods (Kingma & Ba, 2014). That is, it is not only can store the exponentially decaying average of past squared gradient, but also keep exponentially decaying average of past gradient (Xu, 2018). In addition to this, it is able to apply adaptive learning rate for different parameters respectively (Xu, 2018). Therefore, some strengths of Adam including it is able to achieve a good result faster than Adgrad, RMSprop and AdaDelta etc (Ruder, 2016). It is suggested (Ruder, 2016) that this method achieves best overall results amongst the adaptive learning algorithms mentioned above. Additionally, Adam introduces the bias-correct concept (Ruder, 2016). Thus, the team made an attempt to implement Adam to acquire better accuracy. At the cost of better performance, the algorithm becomes more complex as 3 more new parameters were introduced. The pseudo code explains how these parameters work with SGD, and implementation details can be seen in codes file.

```
SGD with Adam
Initialization: parameter $\theta$ , time step t = 0 , two moment variables: m =0, r=0
Require : learning rate $\eta$, small value for stabilization $\delta$ (default : $10^{-8}$)
        beta 1 & beta 2 for representing the exponential decaying for moment $\beta1, \beta2$
While stop convergence criterion not met do:
        Compute gradient estimate:  gradient (g)
        t ← t+1
    Compute biased first-moment update: $s \leftarrow \beta1 s + (1-\beta1)g$
        Compute biased second-moment update: $r \leftarrow \beta2 r + (1-\beta2) g \cdot g$
        Compute correct bias for first & second moment: $s' \leftarrow \frac{s}{1-\beta1^{(t)}}$ $r' \leftarrow \frac{r}{1-\beta2^{(t)}}$
        Compute the update: $\Delta\theta \leftarrow -\eta \frac{s'}{\sqrt{r'}+\delta}$
                $\theta \leftarrow \theta + \Delta\theta$
```

(Adapted from ‘Deep Learning’ (Goodfellow, 2016, p. 311))

The actual performances in terms of SGD with momentum and Adam will be presented and discussed in section 5.

#### 4.2.4. Dropout

The ‘dropout’ terminology in neural network refers to randomly drop some units and its connections from hidden layers during the training stage (Srivastava et al, 2014). Ove-fitting could be a problem for neural network models because of its flexibility (Srivastava et al, 2014). In machine learning field, the over-fitting issue can be reduced by applying a penalty to the loss function, which known as regularization techniques. In a neural network, one of the regularisation techniques, dropout, is another effective means to address this issue, which can minimise interdependent learning amongst the neurons. The computational cost is cheaper than other regularization methods such as weight decay and L1 & L2 regularization, only ***O(n)*** complexity in training stage (Goodfellow et al, 2016). Besides, the dropout is not limited by the type of model used (Goodfellow et al, 2016), which is another reason we chose to implement this approach.

Technically, there are two types of dropout, normal dropout and inverted dropout. In this project, the team developed normal dropout in forward-propagation according to Li & Yang’s study (2016) by multiplying hidden units by i.i.d Bernoulli noise such that neurons are randomly shut down for each iteration with a certain probability. The default probability of shutting down the neurons is setting at 0.5 as suggested by Srivastava (2014). In other words, using a subset of neurons to train, and acquiring a different model for each iteration. It is noted that the dropout is only applied in hidden layers, not in input or output layer (Srivastava et al, 2014). Besides, dropout is not used in prediction phase (Hinton, 2016). The comparison results between neural network models with and without dropout will be discussed in the next section.

## 5. The Proposed Classification Method & Extensive Analysis

We conducted our experiments on three Gradient Descent modules: Stochastic only, Momentum and Adam to study which module performs best on our training dataset with respect to accuracy, loss and runtime. We tested each module using different parameters such as various combinations of activation functions (Softmax is always used in the output layer), learning rates, number of neurons in the hidden layers (we used two hidden layers in all our experiments), and different sample sizes of training data. We tested our model with a subset of 5,000 samples first in order to narrow down the learning rates, the number of neurons, epochs and activation functions that give the optimal test accuracy and loss for each module before testing on the whole dataset to save computational time. For each test, we obtained and calculated the mean and maximum value of test accuracy, training accuracy, and loss across all epochs. To get the test accuracy, we divided the dataset into 9:1 parts for train set and test set respectively. To get the training accuracy, we used the same set for both training and testing. Details of the parameters used for testing are as follows:

| | |
|---|---|
| Sample sizes | 5000, 6000 |
| Number of hidden layers | 2 |
| Number of neurons in each hidden layer | 5, 20, 50, 100, 150, 300 |
| Learning rates | 0.000001, 0.0001, 0.001, 0.01 |
| Activations | ReLU, Leaky ReLU, Tanh, Softmax |
| Epochs | 100 |

**Table 3. Initial set of testing parameters**

The following sections detail our experiment results obtained in each gradient descent module. The results are then summarized in two tables for easier comparison of the difference among three modules.

### 5.1. Stochastic (SGD) only

We first tested our SGD-only module on 5,000 samples with 5 neurons in the hidden layers using Tanh activation and a learning rate of 0.000001. The model started off with a low test accuracy of 7% and reached 28% with the minimum loss of 2.25 after 100 epochs. With ReLU activation, it reached a slightly better accuracy of 32.6% after 100 epochs. Since this low accuracy might be due to the very small learning rate, we decided to increase the learning rate to 0.001, and the model converged after only 48 epochs and peaked at 82.60% accuracy. Increasing the learning rate further to 0.01, the model reached a maximum accuracy of 76.4% after 10 epochs and then suffered from vanishing gradients when its accuracy began to decrease. In this case, the loss also increased after every epoch and peaked at 15.1 after 100 epochs.

Switching to Tanh, we were able to increase the model's learning rate to 0.1 without suffering from vanishing gradients. However, it only reached a maximum accuracy of 81.5% after 50 epochs. It seems the vanishing gradients problem only affects our model if we use ReLU activation, this can be explained by the fact that ReLU returns a constant gradient of 0 for every input smaller than 0. We then attempted to train our model using Leaky ReLU, but the vanishing gradient issue still persisted. That said, using ReLU does give our model a slightly better accuracy than Tanh as it was able to reach a higher accuracy at 50 epochs, however, the maximum learning rate to be used for ReLU without incurring vanishing gradients is 0.001. We decided to continue our experiments with SGD using only ReLU and a fixed learning rate of 0.001, but varied the number of neurons in the hidden layer to see if the model was able to achieve a higher accuracy. Increasing the number of neurons to 20, the model reached a new peak of accuracy at 88.4% after 50 epochs. With 50 neurons, the model peaked at 88.9% accuracy after only 32 epochs, and with 100 neurons, it reached 89.2%, however, increasing the number of neurons further only increased the accuracy by a very small margin but increased the running time unnecessarily. We then tried to use different activations (Tanh and ReLU) for the two hidden layers to see if the results differ. Surprisingly, using the combination of Tanh and ReLU gave us slightly better accuracy than using ReLU or Tanh alone. The model reached a new record for test accuracy of 89.6% after 39 epochs with 100 neurons in the hidden layers.

We concluded that using two hidden layers with **100 neurons** in each with a **learning rate in range 0.0001, 0.001, 0.01** and a combination of **Tanh and ReLU** gave us the optimal results with test accuracy for our SGD module. We then tested this module on the whole dataset with the above set of parameters. The model runs in 50 minutes and the mean accuracy and loss obtained are very similar to the results tested using 5,000 samples. The learning rate 0.001 also gave us the optimal results on the whole dataset. For the other two advanced modules, as they would take more time to run, we tested them using these set of parameters only.

### 5.2. Momentum

We began testing our Momentum module with 5,000 samples, and a beta value of 0.9 on three different learning rates: 0.0001, 0.001, 0.01. With the learning rate 0.0001, the model runs in about 5 minutes and gave a mean accuracy of 85.1% and reached the maximum accuracy of 88.6% after 48 epochs. We then increased the learning to 0.001 and 0.01, both higher learning rates gave the model better mean accuracy and lower mean loss. However, the result for learning rate 0.001 is slightly better with a maximum accuracy of 90.2% and a mean of 87.27%. This result is similar to our SGD module in which learning rate 0.001 gave it the optimal accuracy. We then tried smaller values for beta, 0.7 and 0.5, and the mean accuracy dropped to 78.2% and 69.1% respectively. These results confirmed that the optimal values for beta and learning rate to be used in our Momentum module are 0.9 and 0.001 respectively. Running this module on the whole dataset took approximately 65 minutes for 100 epochs, and the results differed only slightly from using 5,000 samples.

### 5.3. Adam

Testing our Adam module took considerably more time comparing with both Momentum and SGD-only modules. On 5,000 samples, the module runs in 31 minutes, and on the whole dataset (60,000 data points), the module runs in nearly 6 hours, which is roughly 5 times more than our other two gradient descent modules. The accuracies in the first 10 epochs using Adam are higher than those of our other modules. However, it never reached the maximum accuracy that our other modules did. Despite that, it performed very well with decreasing the loss. After 100 epochs using learning rate 0.001, its mean loss was 0.046 with a minimum loss of 0 at epoch 89 – the lowest values among our three modules. However, it is interesting to note that with a higher learning rate than 0.001, Adam module did not do as well with decreasing the loss compared to other modules. This could indicate that Adam works better with lower learning rates compared to the other modules. The results on the whole dataset are again highly similar to using a smaller subset for this module.

### 5.4. Dropout

After testing with three different Gradient Descent modules and obtained the results on their accuracy, loss and runtime, we tested each of them with the Dropout module to see if it improved the results. We first tested on our Momentum module using the whole dataset with a dropout fraction of 0.9. While it did not increase the runtime nor the test accuracy, it lowered the training accuracy by 1% for each epoch in average compared to the non-dropout result. However, with a lower dropout fraction of 0.8, the test accuracy started to decrease by 1.5% on average. The result on SGD module is very similar to this but for Adam module, using dropout fraction of 0.8 decreased the test accuracy more significantly by 2.5% on average. We concluded that while using a high enough dropout fractions does help prevent our model from over-fitting as shown by the decreasing train accuracy, dropout does not play an essential role in our model since it did not improve our test accuracy, but may negatively affect our accuracy if using an incorrect fraction, which in this case is 0.8.

### 5.5. Summary of Experiment Results

The following Table 4 and 5 compare the mean/maximum accuracy, mean/min loss and runtime among three different gradient descent modules using three learning rates: 0.0001, 0.001, 0.01, with optimal results highlighted. The number of hidden layers and their activations, number of neurons, epochs, and values for betas are kept the same. Table 1 details the results using a subset of 5,000 samples while table 2 reports the results on the whole dataset of 60,000 samples. The results between two sample sizes do not differ much regarding mean accuracy and loss, however, running the model on a larger dataset takes noticeably longer time. Among three different learning rates, 0.001 consistently gave the highest maximum accuracy and lowest mean loss across all modules. Increasing it further to 0.01 may give a higher accuracy in the first few epochs but after that, it never converged to the maximum accuracy that using learning rate 0.001 was able to achieve.

| Hidden Layers: 1-Tanh, 2-ReLU \| Neurons = 100 \| Epochs = 100 \| Beta1 = 0.9 \| Beta2 = 0.999 \| Sample size = 5,000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Learning rate = 0.0001 | | | Learning rate = 0.001 | | | Learning rate = 0.01 | | |
| | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) |
| **SGD-only** | 84.90 (88.0) | 0.491 (0.340) | 2 mins | 87.17 (89.6) | 0.148 (0.011) | 3 mins | 85.38 (88.6) | 0.184 (0.082) | 2 mins |
| **Momentum** | 85.66 (88.8) | 0.487 (0.315) | 5 mins | 87.29 (90.2) | 0.142 (0.010) | 4 mins | 85.65 (88.8) | 0.160 (0.081) | 4 mins |
| **Adam** | 87.74 (89.2) | 0.278 (0.100) | 30 mins | 87.22 (89.4) | 0.046 (0.000) | 31 mins | 85.40 (87.8) | 0.168 (0.084) | 30 mins |

**Table 4. Comparison among 3 gradient descent modules using 5,000 samples**

| Hidden Layers: 1-Tanh, 2-ReLU \| Neurons = 100 \| Epochs = 100 \| Beta1 = 0.9 \| Beta2 = 0.999 \| Sample size = 60,000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Learning rate = 0.0001 | | | Learning rate = 0.001 | | | Learning rate = 0.01 | | |
| | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) | Test accuracy mean % (max) | Loss mean (min) | Runtime (±0.5) |
| **SGD-only** | 85.38 (87.09) | 0.561 (0.413) | 49 mins | 87.66 (88.36) | 0.169 (0.091) | 50 mins | 85.86 (87.68) | 0.210 (0.109) | 52 mins |
| **Momentum** | 86.11 (87.60) | 0.572 (2.385) | 66 mins | 87.86 (88.57) | 0.167 (0.090) | 66 mins | 86.10 (87.83) | 0.188 (0.101) | 63 mins |
| **Adam** | 87.58 (88.20) | 0.322 (0.150) | 311 mins | 87.67 (88.42) | 0.098 (0.002) | 308 mins | 86.04 (87.08) | 0.196 (0.124) | 320 mins |

**Table 5. Comparison among 3 gradient descent modules using 60,000 samples**

In terms of running time, SGD-only module has the lowest runtime, followed by Momentum and Adam. Noticeably, Adam has a much worse runtime than all other modules. This could be a huge disadvantage for Adam when running on a much larger dataset than the one we used as it could result in arbitrarily long runtime due to the size of the dataset. Regarding mean accuracy, Adam gave the highest value with learning rate 0.0001, followed by Momentum and SGD at learning rate 0.001, however, the differences between these mean values are very small as they all converge towards 87%. That said, regarding maximum accuracy, Momentum was able to obtain the record high of 90.2%, followed by SGD and Adam at 89.6% and 89.2% respectively, all with learning rate 0.001. Figure 3, 4 and 5 below show the loss plots of three different modules all using learning rate 0.001. Out of three modules, Adam has the lowest mean loss and was able to get 0 loss at epoch 78. Its loss plot shows a clear difference than the other two modules that it converges to the minimum loss faster.

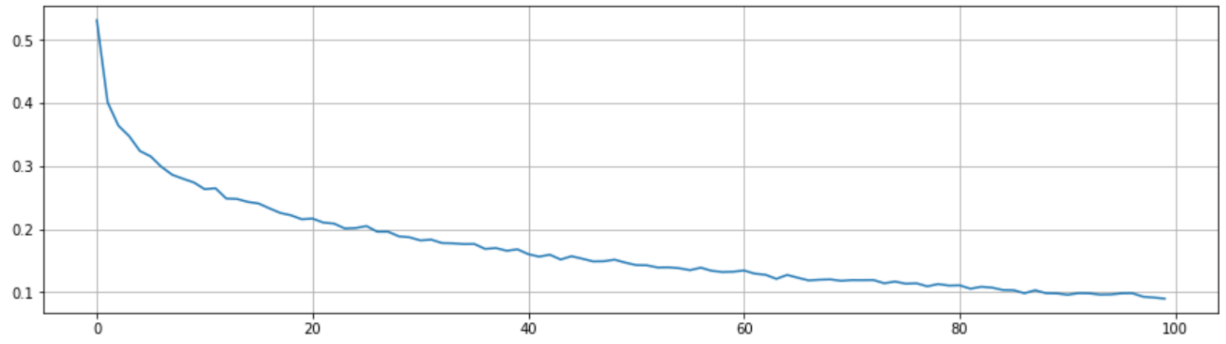

**Figure 3. Loss plot of SGD**

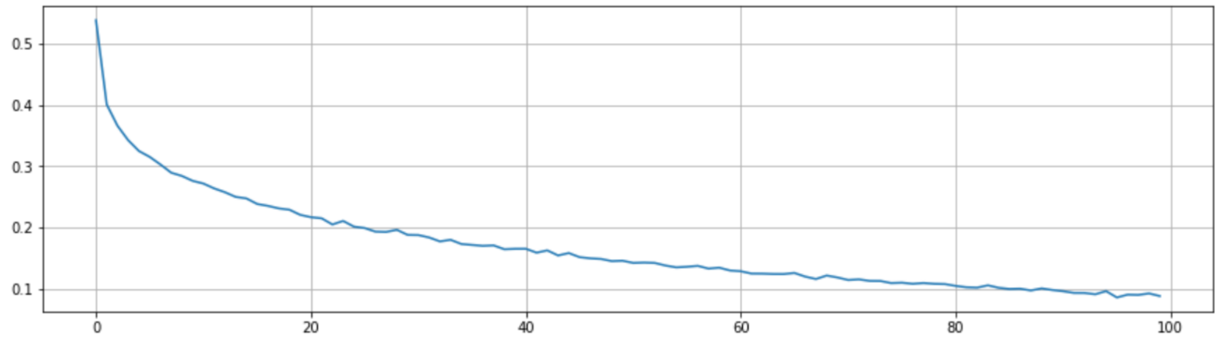

**Figure 4. Loss plot of Momentum**

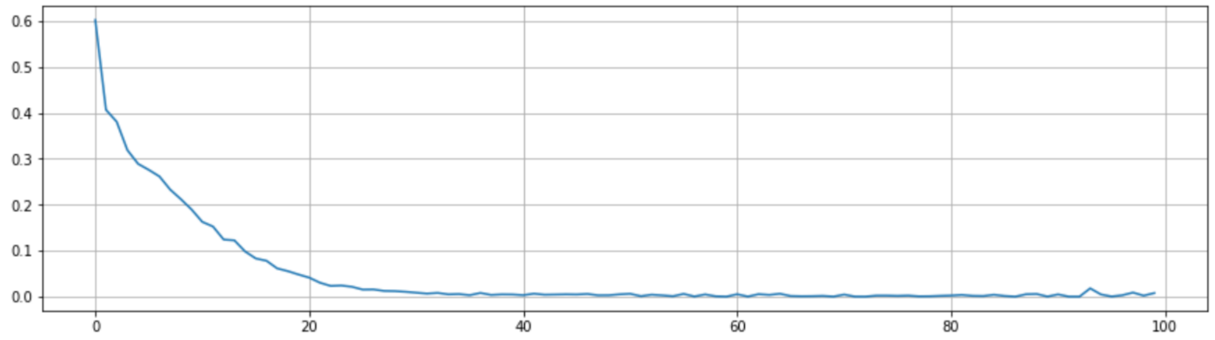

**Figure 5. Loss plot of Adam**

### 5.6. Proposed Classification Model

After completing rigorous testing on several combinations of different modules, we have finalized a classification model that gives the highest test accuracy on the training data and runs within a feasible time. The proposed method is a neural network with two hidden layers, with 100 neurons in each, where the output layer uses SoftMax activation while the two hidden layers use Tanh and ReLU activations respectively. The model uses Momentum module for gradient descent, with a fixed learning rate of 0.001 and beta value of 0.9 and a no dropout (dropout value = 1). The neural network begins to converge and reached the maximum accuracy of around 88.56 after 20-30 epochs and takes less than 35 minutes to run. The reason for this is, firstly, despite giving a similar mean accuracy compared to the other two modules, our Momentum module was able to reach the highest maximum accuracy. On the other hand, even though its mean loss is higher than that of Adam, considering Adam's much longer runtime, this small difference in loss is insignificant as that it does not affect the accuracy. Moreover, although our Momentum module's runtime is slightly higher than SGD-only, it starts converging to the maximum accuracy faster than SGD after around 30 epochs. This means in practice we could reduce its runtime by running fewer epochs until it reaches the desired accuracy.

## 6. Conclusions and Future Works

The team has managed to construct a multi-layer neural network and applied it to solving a multi-label classification problem. After testing with various parameters such as different activation functions, learning rates, gradient descent modules, we concluded with a proposed model that gave the optimal results on accuracy with a feasible running time. The model uses Tanh, ReLU activation functions for the hidden layers, Softmax for the output layer, and Momentum optimizer for gradient descent. On top of that, even though not included in our proposed model, dropout regularization and optimizations methods such as Adam have been confirmed that they can contribute to acquiring better accuracy and loss compared to using only SGD.

There is space for improvement, however, in terms of computational cost and final accuracy. The current time spent is acceptable when momentum and dropout techniques are applied in the neural network, but a bit longer if using Adam as optimization method of stochastic gradient descent. This means codes optimization is worth trying in the future. In addition to this, advanced modules for optimizing SGD, for example, Nesterov accelerated gradient and Adam can be employed and test the accuracy, compared to the results that produced by momentum and Adam. Finally, additional optimization strategies such as early stopping and batch normalization could be considered in the future works.

## References

Fumo, D. (2017). *A Gentle Introduction To Neural Networks Series — Part 1*. [online] Towards Data Science. Available at: https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc [Accessed 4 May 2018].

Glorot, X. and Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

Goodfellow, I., Bengio, Y., Courville, A. and Bengio, Y., 2016. Deep learning (Vol. 1). Cambridge: MIT press.

He, K., Zhang, X., Ren, S. and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).

Hinto, G. 2016. *Lecture 10.5 — Dropout [Neural Networks for Machine Learning]*. [video] Available at: https://www.youtube.com/watch?v=vAVOY8frLlQ [Accessed 9 May 2018].

Joshi, P. 2018. *Understanding Xavier Initialization In Deep Neural Networks*. [online] PERPETUAL ENIGMA. Available at: https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/ [Accessed 9 May 2018].

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kumar, S.K., 2017. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863*.

Li, Z., Gong, B. and Yang, T., 2016. Improved dropout for shallow and deep learning. In *Advances in Neural Information Processing Systems* (pp. 2523-2531).

Murphey, Y.L. and Luo, Y., 2002. Feature extraction for a multiple pattern classification neural network system. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on* (Vol. 2, pp. 220-223). IEEE.

Nasrabadi, N.M., 2007. Pattern recognition and machine learning. *Journal of electronic imaging*, *16*(4), p.049901.

Ng, A. 2016. *Deep Learning Specialization*. [video] Available at: https://www.coursera.org/specializations/deep-learning [Accessed 22 Apr. 2018].

Ou, G. and Murphey, Y.L., 2007. Multi-class pattern classification using neural networks. *Pattern Recognition*, *40*(1), pp.4-18.

Polyak, B.T., 1964. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), pp.1-17.

Ruder, S., 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Scharth, M. 2017. Statistical Learning and Data Mining (QBUS6810), lecture 8, week 7: Linear Method for Regression I [Lecture PowerPoint slides]. [Online] Available at: https://elearning.sydney.edu.au/bbcswebdav/pid-4941191-dt-content-rid-20942119_1/courses/2017_S2C_QBUS6810_ND/QBUS6810-8.pdf

Sharma, A., Sandooja, B., & Yadav, D. 2013. Extreme Machine Learning: Feed Forward Networks. International Journal of Advanced Research in Computer Science and Software Engineering, 3(8), pp.1366–1371.

Shamseldin, A.Y., Nasr, A.E. and O'Connor, K.M., 2002. Comparison of different forms of the multi-layer feed-forward neural network method used for river flow forecast combination.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), pp.1929-1958.

Xu, C. 2018. Deep Learning (COMP5329), lecture 3, week 3: Multilayer Neural Network [Lecture PowerPoint slides]. [Online] Available at: https://canvas.sydney.edu.au/courses/2442/pages/week-3-multilayer-neural-network?module_item_id=126565

Xu, C. 2018. Deep Learning (COMP5329), lecture 4, week 6: Optimization for Training Deep Models [Lecture PowerPoint slides]. [Online] Available at: https://canvas.sydney.edu.au/courses/2442/pages/week-6-optimization-for-training-deep-models?module_item_id=198599

## APPENDIX A - How to run the codes

1. Open the Jupyter Notebook 'Assignment_1.ipynb' in folder /Code/Algorithm/
2. Please replace the locations of the input files **train_128.h5**, **train_label.h5** and **Predicted_labels.h5** accordingly in line 8, 10 and 13
3. Click on Cell - Run All

The notebook should run our proposed model in less than 35 minutes and displays the following:

- Print out the test accuracy, train accuracy, loss in each of the 50 epochs
- Print out the mean and maximum values for test accuracy, mean and minimum values for loss over all epochs
- Display the total running time
- Plot the loss against 50 epochs

## APPENDIX B - Predicted outputs for test data

The predicted outputs for test data obtained from 'test_128.h5' are contained in file 'Predicted_labels.h5', which can be found in folder /Code/Output/

The file should contain numerical labels (ranging from 0 to 9) for 10,000 samples. Using Python package h5py to read the file and print its shape, type and first 100 predicted labels should display

```
(10000,)
<class 'numpy.ndarray'>
[9 2 1 1 0 1 4 6 5 7 4 5 5 3 4 1 2 2 8 0 2 5 7 5 1 4 4 0 9 4 8 8 3 3 8 0 7
 5 7 9 0 1 6 5 4 9 2 1 2 6 4 4 5 0 2 2 8 4 8 0 7 7 8 5 1 1 0 4 7 8 7 0 6 6
 2 1 1 2 8 4 1 8 5 9 5 0 1 2 0 0 5 1 6 7 1 8 0 1 4 2]
```

## APPENDIX C - Hardware and Software Specifications

All testings were carried out using a Microsoft Surface Pro 3 laptop with the following specifications:

**Hardware:**
Processor: Intel(R) Core(™) i5-4300U CPU @ 1.90GHz 2.50 GHz
RAM: 8 GB
System type: 64-bit Operating System, x64-based

**Software:**
Anaconda's Jupyter Notebook