THE UNIVERSITY OF SYDNEY

ASSIGNMENT 1 DEEP LEARNING

COMP4329/COMP5329

FINAL REPORT

Multi-class Classification using Multilayer Perceptron: a computational study

Authors:
Junzhi Ning, ID: 490059823
Yuhao Li, ID:520576076

Submitted to:
Lecturer: Chang Xu
Head Tutor: Linwei Tao

April 5, 2023

Contents

Abstract

Neural networks, as the foundation of the deep learning methodology, has been gained wide attention over the years, leading to the desired demand to understand the philosophy and idea behind the Neural Network. This report will mainly analyze and evaluate each of the self-implemented multilayer perceptron modules such as the regularization techniques, optimizers, activation functions, etc according to a comprehensive computational study. Experiments related to each module will be conducted for remedying a 10-class classification task on 10000 test data samples training in 50000 training data for examining the effectiveness when changing different hyperparameters while maintaining other modules constantly. Discussion related to experiment results, limitations of our work, and meaningful conclusions and reflection will also be provided.

1 Introduction

Neural networks achieves significant revolution over the last few decades in the era of big data and deep learning. It stems from the perceptron proposed in 1958 [9], the multi-layer perceptron has been gained attention not until 1986 [10], when the backpropagation for adjusting the weight parameter, as well as the introduction of more hidden layers significantly improved the accuracy of Neural networks. With improved computational power, deeper multi-layer perceptrons can be trained with a descent time efficiency for better results.

1.1 Purpose of the study

In this project, we have developed several modules in the neural network from scratch using numpy library, instead of the API function from PyTorch. The purpose of this study is to understand the fundamental structure and reasoning behind the neural network and perform a computational study to compare and test the performance when changing various experimental hyperparameters, such as learning rate, optimizer, etc. We havve developed multiple modules, including activation functions such as the ReLU function, and optimizers such as Stochastic gradient descent with momentum, regularization technique including batch normalization, dropout, and mini-batch training procedure. Using test accuracy, execution time, and validation loss as evaluation metrics, we would like to compare the effect of multilayer perceptron when deploying different combinations of modules to provide further guidance when training similar models in the future.

1.2 Significance of the study

The motivation for this study can be attributed to the following. Firstly, the black-box nature of the Neural network means tiny modifications can incur drastic fluctuations in empirical performance. hence we would like to study the empirical performance by changing non-linearity and optimizer function, which are pivotal parameters for test performance.
Secondly, the neural network is prone to overfitting, which means the model can squeeze the training error while enabling the continuous surge of validation loss instead. According to the universal approximation theorem, the neural network can fit any universal function, with the assistance of the activation function facilitating non-linearity. This means neural network tends to produce an over-complicated model that surpasses the actual demand for the data, causing over-estimated performance in the test set. As a consequence, various regularization techniques have been proposed such as dropout that abandons the use of neurons in a neural network to prevent model-producing complex models, weight decay that is equivalent to adding l_2 norm of weight to the loss function for penalizing large weight parameters, etc. Our goal is to study which regularization technique is suitable for our data and compare their performance accordingly. Thirdly, refining neural network structure is also important even though the computational power has been enhanced dramatically, efficient modules and structure can accelerate productivity and efficiency and benefit related researchers.

1.3 Outline

This report will be divided up into 4 sections. Section 2 will introduce the nature of given data distribution, preprocessing methods, and theoretical philosophy of modules including regularization techniques, optimizers, etc. Section 3 will investigate empirical performance when assigning various regularizations or hyperparameters to the modules and infer the best design model after a comprehensive ablation study and comparative study of related experiments and results. The last section will demonstrate limitations and future work of our work.

2 Methods

2.1 Dataset

In this project, the provided dataset consists of 50,000 data entries, each with 128 dimensions and associated with one of ten unique class labels. Although the dataset's origin and the nature of the data type have not been specified by the instructor, our preliminary data analysis indicates that most features follow a normal distribution with varying variances. Supporting evidence can be found in Figure 2 and Figure 3.

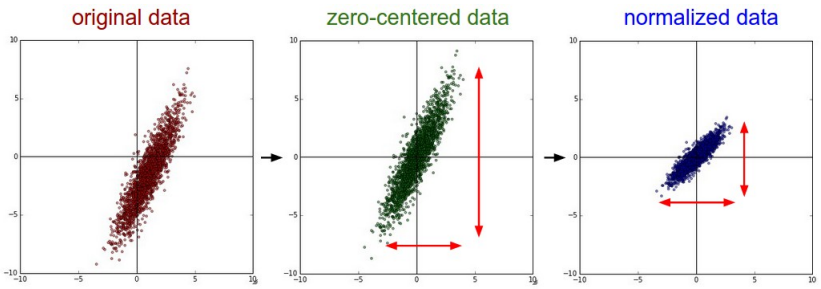

Figure 1: Comparison between normalized data and non-normalized data from [6]

2.2 Preprocessing

Preprocessing is a vital stage in the machine learning pipeline, as it readies raw data for model training and evaluation. This process ensures that the machine learning algorithm receives consistent and clean input, making it suitable for creating models. Preprocessing generally encompasses tasks such as feature scaling, handling missing values, encoding and decoding categorical variables, and noise reduction. In our specific project, the focus is on a 10-class classification problem. The supplied raw data is complete and devoid of missing values. Nonetheless, due to the unknown origin of the dataset, it is challenging to ascertain its nature and apply specialized preprocessing to particular data types. Our approach is to employ a Multilayer Perceptron (MLP) to tackle the classification task. To meet the output requirements of the MLP, we will standardize the data and transform the class labels into one-hot encoding vectors.

2.2.1 Standardisation to raw data

The standardization technique is one of the special feature scaling techniques. The main goal of the standardization technique is to control the range of values of the raw data by its mean and standard deviation before feeding into the model. It is widely applied in statistical machine learning algorithms such that after standardization of the dataset, each dimension of the dataset will not dominate the learning of the distance-based model shown in Figure 1 and the feature dimension could contribute to the update of the parameters invariant of its scale. Mathematically speaking, the standardization (also known as Z-score normalization) is defined as:

$$z_{x_i} = \frac{x_i - x_{mean}}{\sigma_x}, \tag{1}$$

where x_i is a data point in the dataset of X, x_{mean} is the mean of the data points across one dimension, and σ_x is the variance of the dimension. In a more practical sense, the process of standardization can be reformulated as a vectorized version so that it can be more effectively implemented to save computational resources and optimize for speed in different applications of model building.

$$z_{x_i} = \frac{x_i - x_{min}}{x_{max} - x_{min}}. \tag{2}$$

Alternatively, Min-max scaling defined in Equation 2 can be applied in the same setting, but we forgo the usage of this method for the following reasons.

- Robustness to the outliers in the case of the presence of extreme outliers.
- Preserving the distributional information after the standardization.
- Interpretability of the standardization with the statistical test and tools.
- The raw data provided follows the normal distributions shown in Figure 2 and Figure 3.

2.2.2 One hot encoding to class labels

To make raw data compatible with machine learning algorithms for multi-class classification tasks, it is often necessary to convert class labels into one-hot vectors, as these algorithms cannot distinguish between discrete and continuous variables. More specifically, a binary vector or array is created for each class label, with the size equal to the number of unique classes in the dataset. In this binary representation, a '1' is placed at the position corresponding to the class label, while all other positions are set to '0'. Essentially, this method converts a numerical value into a binary representation.

Consider the following example: assume a dataset with three class labels, "red", "blue", and "green", representing the size of an object. The possible combinations of one-hot encoding vectors are:

- $v_{red} = [1, 0, 0]$
- $v_{blue} = [0, 1, 0]$
- $v_{green} = [0, 0, 1]$

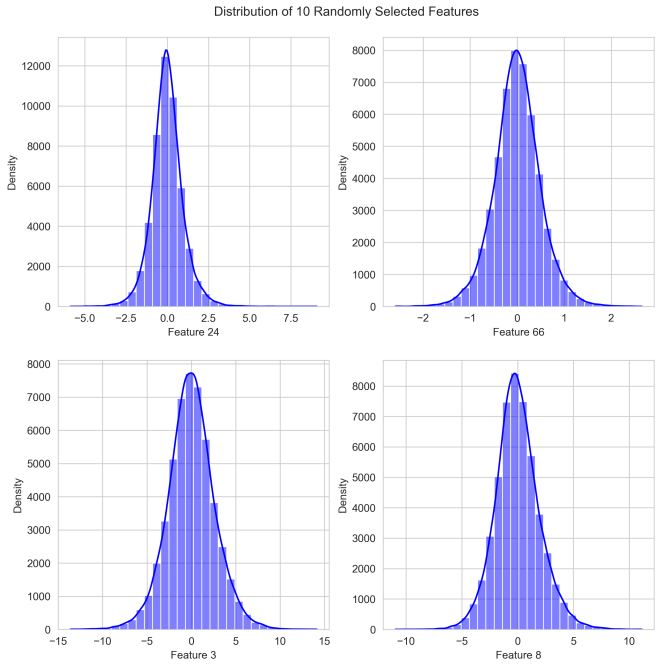

Figure 2: Visualisation: Randomly selection of 5 features distributions

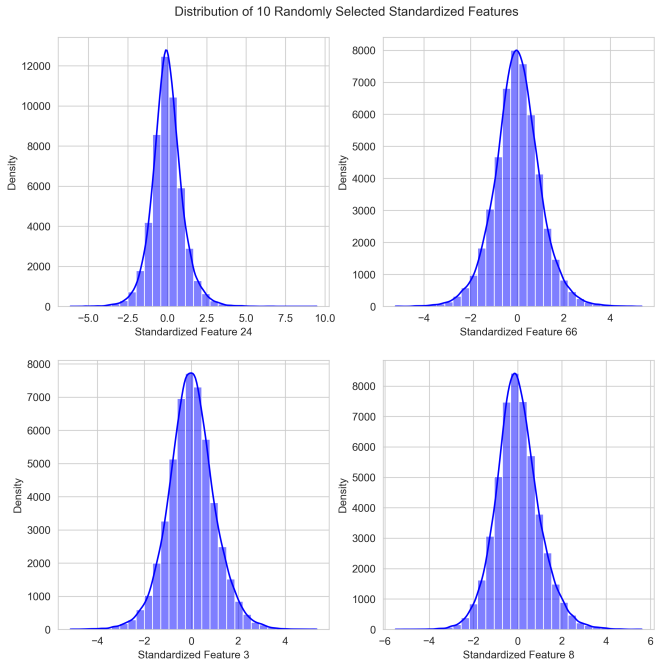

Figure 3: Visualisation: Random selection of 5 standardized distributions

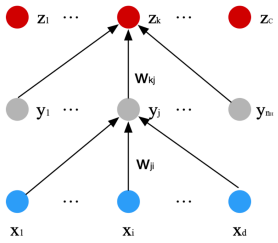

Figure 4: Multilayer Perceptron

In the context of a project with a dataset containing 10 classes, there would be ten unique one-hot vectors, each with a length of 10, defined as described above.

2.3 The principle of different modules

The following notation for the principle of modules has been cited from lecture slides in Deep Learning [12], [13], [14].

2.3.1 Mutlilayers Perceptron

The structure of a multilayer perceptron can be categorized into three parts as shown in Figure 4. Input layer: the layer that receives data X. The hidden layer indicates the layer with neurons embedded between the input layer and the output layer, which facilitates more complicated models and better results. The output layer is the final layer in the multilayer perception, which output the final result from neural networks.
Given the current layer x that has d units, f is the activation function, y_k refers to the j_{th} output of the next layer, w_{ji} refers to the weight from i_{th} neuron in current layer to j_{th} neuron in the next layer, b_j refers to the bias term for y_j. can be calculated via forward propagation by Equation 3 for a feedforward neural network.

$$y_j = f(\sum_{i=1}^{d} x_i w_{ji} + b_j), \tag{3}$$

For a deeper structure, y_j will be fed into the next layer and continue forward propagation until reaching the output layer. To learn the weight parameter W, the backward propagation procedure uses a chain rule to obtain gradient information recursively from the output layer to the input layer. Calculating the loss between actual output and current output, gradient information will be used by optimizers such as Stochastic Gradient Descent, etc, for obtaining optimal weight parameters that minimize the loss function.

2.3.2 Activation function

If the activation function is not assigned then Equation 4 becomes a linear classifier. As a matter of fact, the activation function enables the non-linearity property of the neural network, preventing it from being a linear classifier despite deepening the neural network.

$$y_k = \sum_{i=1}^{d} x_i w_{ki} + b_k, \tag{4}$$

There are additional requirements for constructing a reasonable activation function. Firstly, the activation function must be non-linear as discussed above. Secondly, the activation function must be continuous and differentiable for obtaining valid derivatives for backward propagation. Finally, establishing monotonicity guarantee the elimination of additional local extrema during the optimization process. Note we have only listed only some of the necessary features for a valid activation function instead of all properties. Common activation function choices involve the sigmoid function, tanh function, but Rectified linear unit (ReLU) activation and its variant Leaky ReLU function are our primary interests in this project. ReLU function can be defined as Equation 5, its' function can depicted in the left of Figure 5

$$f(x) = \max(0, x). \tag{5}$$

There are some merits to using ReLU activation function. Firstly, there is no vanishing gradient issue for ReLU since it doesn't have an area of the saturated region. Saturation means the derivative of the function reaches arbitrarily 0 for some intervals, hindering the calculation of derivatives and thus the backpropagation procedure. Secondly, the ReLU activation function can achieve a faster convergence speed since the functional form of its derivative is simple(1 when $x \geq 0$, undefined when $x < 0$), as opposed to the derivative of the sigmoid function and tanh function. Finally, it introduces sparsity that squeezes the output to 0. On the other hand, the sparsity nature of the ReLU function can cause detriments. If a negative number is fed into ReLU, it will return 0 with deadly neurons, that produce useless current neurons. There is a possible solution, which is attempting with a tiny learning rate and correct weight initialization, but a more effective alternative is to

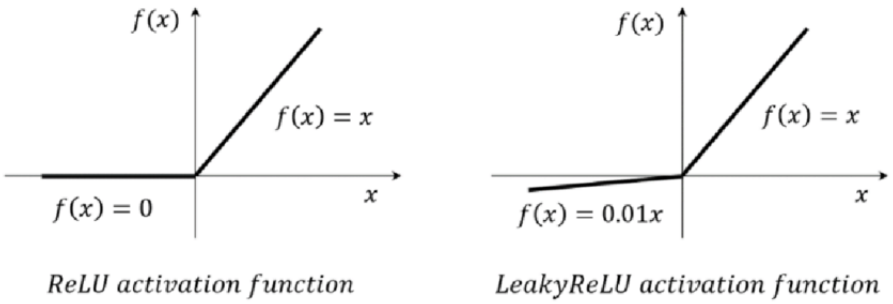

Figure 5: ReLU activation function

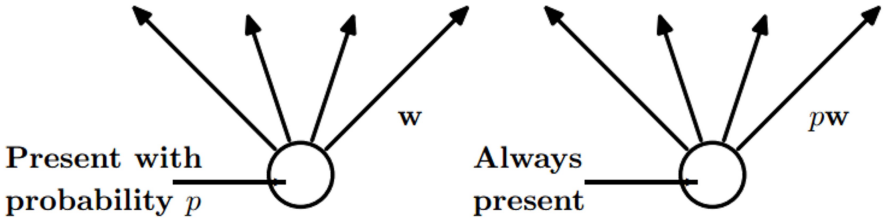

Figure 6: Dropout demonstration

substitute a Leaky ReLU, preventing the incidence of zero derivatives when there are negative inputs to ReLU function, as shown in the right of Figure 5 and Equation 5. The derivative of a negative number of the Leaky ReLU function becomes α instead. A frequent choice of α is 0.01.

$$f(x) = \begin{cases} s & \text{if } s \geq 0 \\ \alpha s & \text{if } s < 0 \end{cases}, \text{where } \alpha \text{ is a small constant.} \tag{6}$$

2.3.3 Weight Decay

Regularization prevents overfitting. Overfitting occurs when the training set error is small but the test set error is large. A potential solution for remedying this issue includes weight decay, where a l_2 norm penalization term is assigned to the original objective function for punishing huge gradient as demonstrated in Equation 7. Weight decay [5] has demonstrated its' huge success in preventing exploding gradients when training deep neural networks (for penalizing large weight) and overfitting issues. Demonstrating weight decay from the researcher's point of view, λ is the penalty parameter that controls the size of penalization, and larger λ normally infers smaller weight.

$$\min_\theta \hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\lambda}{2}||\theta||_2^2. \tag{7}$$

To update the parameter, a gradient descent step is necessary. Taking the gradient concerning θ for regularized objective $\hat{L}_R(\theta)$ and denoting η as the learning rate assigned, the gradient descent update step can be simplified as the following equation,

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}_R(\theta) + \lambda\theta. \tag{8}$$

So the gradient descent update formula for weight decay can be further simplified to:

$$\theta \leftarrow (1 - \eta\lambda)\theta - \eta\nabla\hat{L}(\theta). \tag{9}$$

2.3.4 Dropout

Motivated by the phenomenon in sexual evolution that genes can achieve connectivity with a small random set of genes [11], the dropout approach that activates each unit by certain probability could be another regularization approach for resolving the overfitting issue in the deep neural network. The most traditional approach for dropout in practice is to assign a retained random variable r from a Bernoulli distribution that indicates the probability of retaining each unit with a probability of p at the training stage. When $r_j^{(l)}$ is 0, then j_{th} unit in the layer L is deactivated, otherwise, it is activated. Scaling down the test stage probability by a factor of p can be essential, since all units will be utilized during the test stage, while only part of the units will be used in the training stage, maintaining the stability of the entire neural network, as illustrated in Figure 6. Defining the output of the next layer as $y^{(l+1)}$, f is the activation function, $w_i^{(l+1)}$ and $b_i^{(l+1)}$ as weight and bias parameter for the next layer, The feed-forward neural network operation using dropout can be written as

$$y_i^{(l+1)} = f(w_i^{(l+1)} r^{(l)} \times y^{(l)}), \text{where } r_i^{(l)} \sim \text{Bernoulli}(p). \tag{10}$$

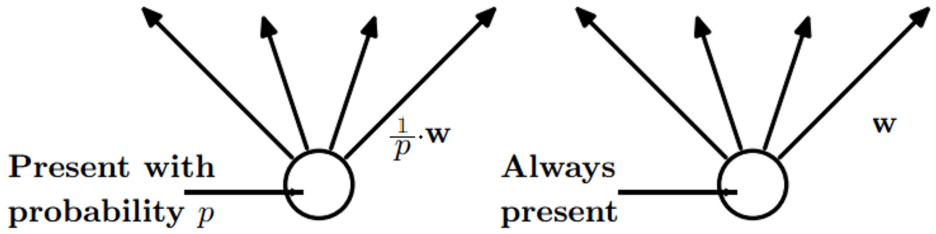

Figure 7: Demonstration for inverted dropout

2.3.5 Inverted Dropout

Another alternative method to dropout is the inverted dropout, where the only difference is that the scaling operation for the weight parameter has been transferred from the test phase to the training stage directly, achieving a similar effect as shown in Figure 7.

2.3.6 Softmax and Cross-entropy loss

The softmax function is normally used before the output layer for returning the conditional probability distribution of prediction of each class given by the current feature x as described in Equation 11,

$$\hat{P}(class_k|x) = \frac{e^{net_k}}{\sum_{i=1}^{K} e^{net_i}}. \tag{11}$$

where net_k is $w_j^T x$, $class_k$ is the k_{th} class of the data label, K represents a total number of distinct classes. It is mainly used for multi-classification problems for providing a probabilistic explanation of neural network prediction.

Additionally, the loss between the probabilistic prediction after flowing out the softmax layer and one hot encoding of the ground-truth label can be compared by the cross-entropy loss function as described in Equation 12, where t is the one-hot encoding form of ground truth label is z is the output after the softmax layer:

$$\text{Crossentropy}(t, z) = -\sum_{i}^{K} t_i \log(z_i). \tag{12}$$

The cross-entropy loss can be categorized into two segments. The left part of Equation 13 is Entropy: a metric for depicting the degree of disorder of randomness, higher entropy implies a higher chaotic pattern. The right part of Equation 13 depicts Kullback-Leibler divergence, which describes a distance metric for two probability distributions when distribution $t_i \in T$ is approximated opposed to another probability distribution $z_i \in Z$. Lower KL-Divergence implies two probability distributions are more similar:

$$\text{Crossentropy}(t, z) = -\sum_{i}^{K} t_i \log(t_i) + \sum_{i}^{K} t_i \log(\frac{t_i}{z_i}). \tag{13}$$

The use of a cross-entropy loss provides an effective evaluation metric for losses between output probability and ground truth label for multi-classification.

2.3.7 Mini-batch training

Mini-batch training first introduced in [1] is an optimization technique that divides the training set into mini-batches and mini-batch learning encourages the update of model parameters using the gradient descent method based on a batch instead of the whole dataset or a sample. It is a trade-off strategy between the Stochastic Gradient Descent method and the gradient descent method as it circumvents the disadvantages and combines the advantages of both methods. Unlike one sample stochastic gradient method, the mini-batch training is more robust to the estimation of the gradient noise and the update of the weight in the model will be smoothly conducted with a better probability of converging the global minimum in the hyperspace of the loss function. In addition, the minibatch training has a simpler theoretical analysis of the weight dynamic and rate of convergence. On other hand, compared to the whole-dataset gradient method, it demands less computational resources and empowers the optimization procedure on less desirable hardware devices. The graphical illustration of the mini-batch-based Gradient method with other methods is shown in Figure 8.

Mathematically speaking, given a dataset $\{x_i, y_i\}_{i=1}^{M} := \mathcal{D}$ with x_i and y_i representing the data point and its corresponding label respectively, the mini-batches with a size of $b \leq M$ are created by randomly shuffling and truncating into $K := ceil[M/b]$ subgroups such that each mini-batch is a subset of the dataset D and each mini-batch with index k is defined as:

$$B_k := \{(x_i, y_i)\}_{i \in I_k} \subseteq \mathcal{D}, \tag{14}$$

Algorithm 1 Mini-batch-based Gradient Descent Method in MLP [12]

Initialize variables: network topology (n_H), w, criterion $\theta, \eta, \mathrm{r} \leftarrow 0$
for $r \leftarrow r + 1$ (increment epoch), $\nabla J(\boldsymbol{w}) < \theta$ **do**
    $k \leftarrow 0$; $\Delta w_{ji} \leftarrow 0$; $\Delta w_{kj} \leftarrow 0$;
    for $k \leftarrow k + 1$ until $k = K$ **do**
        $x^k \leftarrow$ randomly chosen pattern
        $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_i$; $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_k y_j$
    end for
    $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$; $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$
end for
return w

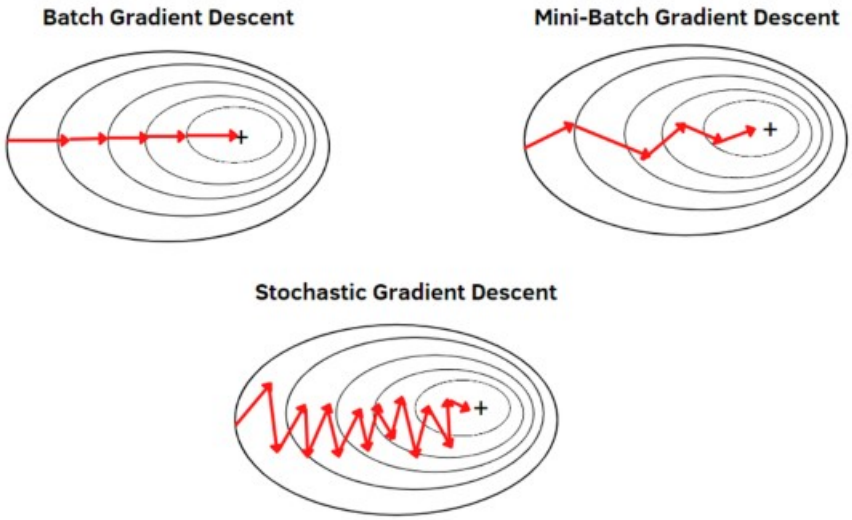

Figure 8: Comparison across three methods of Gradient descent with respect to different batch sizes obtained from [7]

where $I_k \subseteq \{1, 2, \ldots, M\}$ is a random subset of indices with size b chosen at iteration k of sampling from the randomly shuffle $\mathcal{D}$ in one epoch. The subset I_k represents the indices of the examples in the mini-batch B_k. One possible way to utilize this mini-batch paradigm is provided in 1.

Mini-batch training presents a proficient optimization approach that amalgamates the merits of both stochastic gradient descent and gradient descent techniques. This method facilitates dependable gradient approximation; however, certain issues persist, such as covariate shift and erratic loss trajectories during optimization. Consequently, the training of multilayer perceptrons may not be entirely addressed through the employment of the mini-batch training paradigm.

2.3.8 Momentum in Stochastic Gradient Descent (SGD)

During the backpropagation procedure, the weight would be updated repeatedly until the loss function converges. Batch gradient descent that calculates gradient using entire sample information is one solution for seeking optimal weight parameters, but it is notorious for slow convergence rate and computationally challenging cost due to iterating samples repeatedly. Therefore, mini-batch gradient descent that randomly uses subset of training examples is instead proposed. To achieve faster training speed, the stochastic gradient descent method that randomly picks and calculates the gradient of one sample is used. Details have been discussed in the last subsection. Stochastic gradient descent, mini-batch gradient descent, and vanilla gradient de-

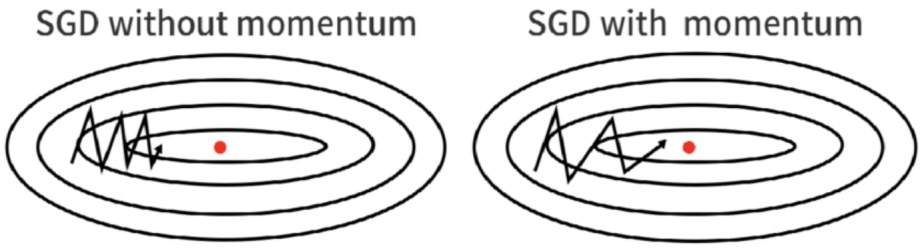

Figure 9: Comparison between SGD and SGD with momentum optimization trajectory

scent are fundamental optimization approaches for deep neural networks, but faster and more efficient optimizers are required for accelerating the training process. Classical optimizer such as gradient descent has a higher tendency for stocking in the local minimum point, deteriorating the found weight parameter. Inspired by mechanical systems and in physics, a heavy ball will continue sliding when reaching a minimum point in a valley, this property can be generalized into an optimization system as well, where a momentum term is added to the gradient descent step for reducing oscillations and escaping the current local minimum point as shown in Figure 8. The new formulation can be illustrated in the following Equation 15, supposing v_t is the momentum, γ controls the contribution of momentum from the last update that could carry to the current momentum, $J(\theta)$ is the loss function.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta), \tag{15}$$

$$\theta_t = \theta_{t-1} - v_t. \tag{16}$$

In theory, the addition of momentum term enhances when the direction of the previous gradient has the same direction as the current gradient while shrinking when the two gradient directions are opposite.

2.3.9 Batch Normalisation

Normalization of the input is often found useful in the training of the neural model as the diminishing or explosive magnitude of the weight accumulated in a neural network leads to the potential problems of model collapse and model explosion. An effective approach to addressing this problem in a neural network involves standardizing the output of each layer with respect to each mini-batch, a process known as batch normalization. This technique, introduced by Ioffe and Szegedy in 2015 [3], aims to reduce the impact of the input and output of each layer on the preceding and succeeding layers during the forward pass and backpropagation. In fact, we have briefly introduced a similar technique of data standardization in the process of preprocessing and the standardization of the input of data in the preprocessing process can be considered a special instance of batch normalization, which is also called whitening transformation.

To elaborate the batch normalization mathematically, batch normalization introduced in [3] is to ensure the output of each layer within a mini-batch is standardized with its means and standard deviation. Let's say we have a $B_k = \{(x_1, y_1), \cdots (x_b, y_b)\}$ with the size of b. For instance, the first input layer of the neural network l and this procedure is the same for the rest of the hidden layers in a neural network illustrated in Figure 10.

- **For forward passing [14]**:
  we compute the following quantities to standard the current batch inputs

$$\mu_{B_k} = \frac{1}{b} \sum_{i=1}^{b} x_i, \tag{17}$$

$$\sigma_{B_k} = \frac{1}{b} \sum_{i=1}^{b} (x_i - \mu_{B_k})^2, \tag{18}$$

$$\forall x_i \in B_k,\ \tilde{x_i} = \frac{x_i - \mu_{B_k}}{\sqrt{\sigma_{B_K}^2 + \epsilon}}. \tag{19}$$

  For each layer, we have two learn-able parameters γ and β to compute the input for the next layer:

$$x_i^{scaled} = \gamma \tilde{x_i} + \beta. \tag{20}$$

- **For back-propagation pass to update the parameters of γ and β** :
  The rules of updating w in the neural network still hold,

$$\nabla \gamma = \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{scaled}} \tilde{x_i}, \tag{21}$$

$$\nabla \beta = \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{scaled}}. \tag{22}$$

  Then, the update rules of η_{new} and γ_{new}:

$$\gamma_{new} = \gamma_{old} + \eta \nabla \gamma, \tag{23}$$

$$\beta_{new} = \beta_{old} + \eta \nabla \beta. \tag{24}$$

- **For inference**:
  If we have a testing dataset denoted as $D := \{(x_i, y_i)\}_{i=1}^n$ with the sample of n. The output of the first layer in a trained neural network with learned parameters of w, γ_1, and β_1 at this layer will be:

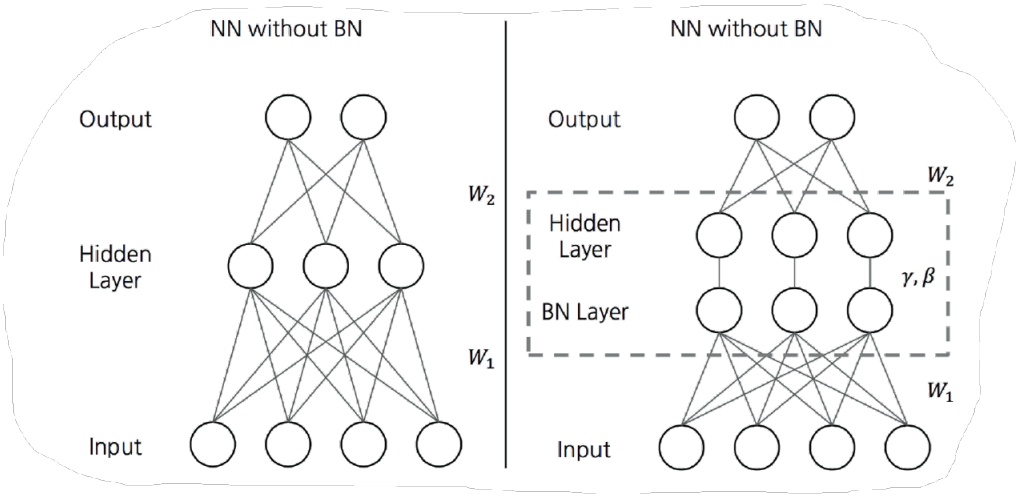

Figure 10: Demonstration of one layer implementation of batch normalization in a neural network [2]

1. Average the means of all mini-batches during the training denote as

$$\bar{\mu} = \mathbf{mean}(\{\mu_{B_k}\}_{k=1}^{K\times \ \#\ of\ epochs}) \tag{25}$$

and

$$\bar{\sigma} = \mathbf{mean}(\{\sigma_{B_k}\}_{k=1}^{K\times\ of\ epochs}). \tag{26}$$

2. Then, $\forall i \in \{1, ..., n\}$, the input of the next layer is:

$$\bar{x_i} = \gamma_1 \frac{x_i - \bar{\mu}}{\sqrt{\bar{\sigma}^2_{B_K} + \epsilon}} + \beta_1. \tag{27}$$

The fundamental motivation for employing batch normalization is to mitigate the internal covariate shift, as discussed in [3] which is the change in the distribution of activation of a neuron in a layer. Without batch normalization, the activation function's output deviates from a Gaussian distribution and exhibits a skewed shape. This deviation leads to heightened sensitivity when excessively active or inactive regions are fed to the activation functions. The ideal scenario for activation input distribution involves outputs from weights that encapsulate beneficial patterns for the ultimate task, irrespective of their magnitude range. In the absence of this distribution, the activation function of taking such inputs may exhibit imbalanced behavior, rendering it less effective and causing the clustering of the value around the saturating values like 1 or -1 for the activation function of $tanh$. In addition to the co-variate shift and its effect mentioned above, the absence of batch normalization might affect the range of the weight in the later layers, making them hard to adapt to the change in a skewed distribution. As a result, this inevitably increases the chance of over-fitting and deteriorating the capability of the model to generalize without learning the useful pattern inherited from the data implicit pattern. Consequently, incorporating batch normalization methodologies in neural networks is crucial and advantageous for multiple aspects of the training process. The benefits encompass a wide range, such as mitigating the necessity for regularization, increasing the tolerability of larger learning rates, and maintaining stability in the outputs propagated across layers.

2.3.10 *Rmsprop (Advanced operations)

RMSprop is an abbreviation of Root Mean Square Propagation, which is an optimization algorithm for training neural networks. It was an optimization method that introduces the adjustment of the learning rate by adding the square root of the exponential moving average. To be more specific, it can be mathematically written as [13]:

$$\theta_{t+1} = \theta_t + \Delta\beta_t, \tag{28}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{\mathbf{E}(g^2)_t + \epsilon}} g_t, \tag{29}$$

$$\mathbf{E}(g^2)_t = \gamma \mathbf{E}(g^2)_{t-1} + (1-\gamma) g_t^2. \tag{30}$$

In the original paper, the author suggests γ to be 0.9 as a parameter for the exponential decay rate and the default learning rate is 0.001. The aim of designing the algorithm in this way is to emphasize the update of the less frequent parameters and penalize the update for the frequent parameters in the previous iterations. One good thing about applying the exponential moving average rather than just solely relying on the summation of $g_{t,i}^2$ is that it circumvents the vanishing problem of $\frac{\eta}{G_{t,ii}=\sum_{i=1}^{t} g_{t,i}^2} \to 0$ as the training time extends, which will be, otherwise, leading to the cessation of the update of parameters.

2.3.11 *Adam (Advanced operations)

Adam which stands for the full name of "Adaptive Moment Estimation" is an optimization algorithm used in training a variety of machine learning models. It was first proposed in the paper of [4]. Adam combines adaptive learning rates from RMSprop and momentum from SGD with momentum, incorporating bias correction to enhance optimization during the training of machine learning algorithms, particularly those based on gradient descent methods. As we mentioned in the above subsection and in the context of MLP, the modified version of the stochastic gradient descent method with an additional term of momentum improves the mechanism of avoiding getting stuck into the local minimum by reducing the fluctuation of decent direction in the process of weight updating and preserving the certain level of inertia to the descent direction from the previous updates. The learning rate in the SGD with momentum has the problem of not being able to adjust the learning rate dynamically. The Rmsprop optimization algorithm addresses this weakness by introducing the computation of an additional term of the exponential moving average of the square of the gradient to regularize the value of the predefined learning rate.

In the language of mathematics, Adam takes both techniques used SGD with momentum and Rmsprop into account when performing the gradient descent of the parameters which can be formulated as,

- Given the parameters of the model as θ and a predefined learning rate η,

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t. \tag{31}$$

- Adam computes the exponentially decaying averages of the past squared gradients v_t and past gradients m_t using β_1 and β_2:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2, \tag{32}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t. \tag{33}$$

- Perform the correction of the bias for m_t and v_t,

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}. \end{aligned} \tag{34}$$

- Then update the parameters θ of the model:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t. \tag{35}$$

Here we present the pseudo-codes of Adam optimization algorithm:

Algorithm 2 Standard Adam implementation [4]

Input: Require: α : Stepsize
Input: Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Input: Require: $f(\theta)$: Stochastic objective function with parameters θ
Input: Require: θ_0 : Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right)$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / \left(1 - \beta_1^t\right)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / \left(1 - \beta_2^t\right)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t \left(\sqrt{\widehat{v}_t} + \epsilon\right)$ (Update parameters)
end while
return θ_+(Resulting parameters)

Theoretically speaking, Adam has a superior performance of optimization compared to SGD with momentum and Rmsprop. It enjoys a faster coverage rate with a dynamically adjusting learning rate and is successfully implemented in different machine learning problems not only in computer vision but also in natural language processes and so on.

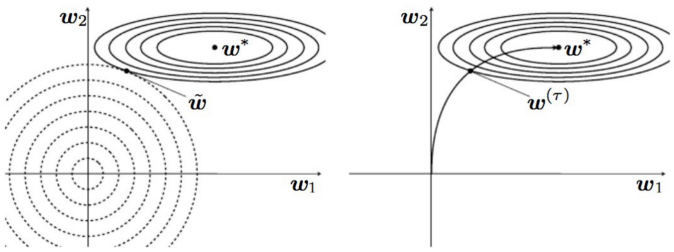

Figure 11: Graphical elaboration of early stopping: it is similar to the regularisation of l_2 norm[2]

2.3.12 *Early stopping (Advanced operations)

Early stopping [8] is a widely-used regularisation technique in deep learning and neural network. Like many other regularisations introduced above, the techniques of early stopping prevent the model from over-fitting, but by stopping the training time before the phenomenon of over-fitting happens. Generally, the sign of over-fitting can be identified through the consecutive epochs of training without any improvement in validation loss, accuracy, and error rate. When the sign of over-fitting happens, the early stopping will terminate the training process and store the copy of the parameters in the model.
The early termination of the training has the advantage of saving computational resources since it only stores only one copy of the weights. However, early training invariably demands a validation dataset to aid in determining the appropriate stopping criteria, which may not be readily available for certain machine-learning problems. A potential solution is to allocate a small fraction of the training set for validation purposes, subsequently re-utilizing the validation set for training upon achieving the correct early stopping point. In our study, we implemented early stopping during the training of our final best model, and a graphical representation of this process can be found in Figure 11.

3 Experiments and Results

3.1 Experiment Setting

We conducted a computational study to evaluate the performance of the MLP using eight dimensions. The hyper-parameter analysis will explore the impact of the number of layers, activation functions, and batch sizes on the model performance of the given dataset. Furthermore, we will examine optimization methods and analyze the pros and cons of well-known optimizers. Additionally, we will conduct an ablation study on the regularization techniques as well as the comparison methods sed in training the neural networks. Based on the insights and observations derived from the above analysis, we will compose and train the best model and evaluate its final performance on the given validation dataset. We did not set up a separate test set for this project, as we only rely on the validation set to evaluate the model's performance and ability to generalize. It is important to note that the validation set will not be directly involved in updating the model's weights [1].

3.2 The Default Design Model For Experiments

Aspect	Hyper-parameters Name	Setting
Neural Network Architecture	Number of Hidden Layers	2
	Layer Neurons	$[128, 150, 100, 10]$
	Activation Functions	Relu
Optimisation	Optimizer	SGD momentum
	Optimizer Parameter(s)	$\gamma = 0.9$
	Learning Rate(lr)	0.005
	Loss function	Cross Entropy with Softmax
Regularisation	Dropout Probability	0
	Batch Size	512
	Weight Decay	0
	Batch Normalisation	False

3.2.1 Implementation details

We established a default model profile shown in the table of the section 3.2.2 for the three main components of our experiments: hyper-parameter analysis, systematic comparison of optimizers, and their parameters. We only modify the

[1] In our project, we provide the links to our implementation of MLP and Data through the CoLab platform in Appendix A, please have a look!

dimensions we want to examine for each control experiment. Our main objective is to reduce confusion and isolate the effect of variation in each model evaluation. In some extreme cases, we make slight adjustments to ensure pathological cases are excluded. For instance, we lower the learning rate when examining the impact of setting the batch size to one; otherwise, the model may not converge even after 150 epochs.

In addition, we ran all experiments on the local machine with the hardware condition described in the table of the section 3.2 to ensure the running experiments were controlled and minimized the effect brought by the hardware conditions.

3.2.2 Hardware and Software specifications of the computer

Software	Python Version Cloud Computing Interface:	3.9.13 Google Colab
Hardware	Operating system Installed memory (RAM) Processor	Windows 11 Pro 16.00GB AMD R7-5800H CPU 8 Cores @3.2GHz

3.2.3 Evaluation Metrics

For comparison of different modifications of modules, the following evaluation criteria would be utilized.

- **Accuracy**: The performance of a multilayer perceptron with certain parameter settings will be compared by the mean of test accuracy over repeated 3 experiments.

$$\text{Validation Accuracy} = \frac{\text{number of correctly classified validation examples}}{\text{total number of validation example}} \times 100\%. \tag{36}$$

- **Loss**: The average training loss and validation loss over 3 repeated experiments, calculated by the cross-entropy loss function as discussed in subsubsection 2.3.6
- **Time**: The average of the model training time over repeated 3 experiments using different modules will also be examined and compared.

3.3 Ablation study Experimental Results

In this subsection, the goal is to examine the effectiveness of by varying each module a time based on a default model.

3.3.1 Learning Rate

No.	Learning Rate	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	0.5	2.133	25.64	2.442	23.31	742.45
2	0.05	0.465	83.06	2.988	47.71	710.75
3	0.005	0.873	69.88	1.486	51.64	736.62
4	0.0005	1.45	49.22	1.497	47.29	717.25

Table 1: Experiment results when using different learning rates: 0.5,0.05,0.005,0.0005

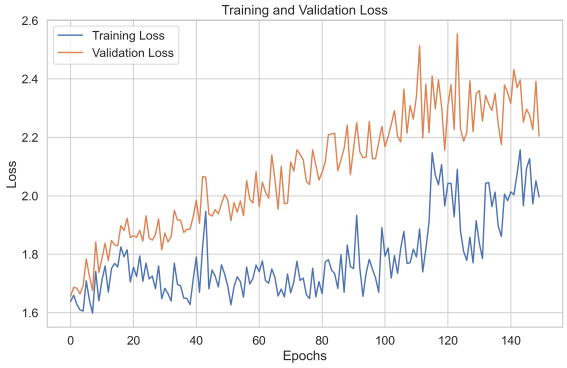

Figure 12: Learning Rate = 0.5

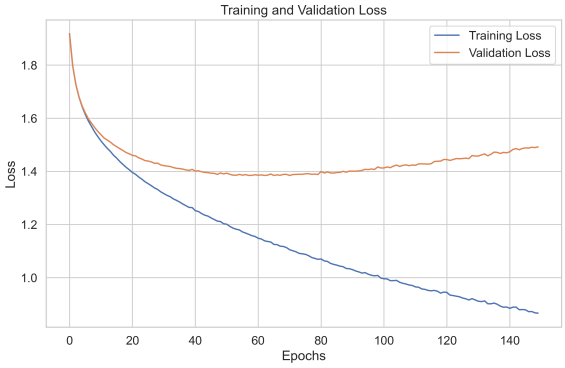

Figure 13: Learning Rate = 0.05

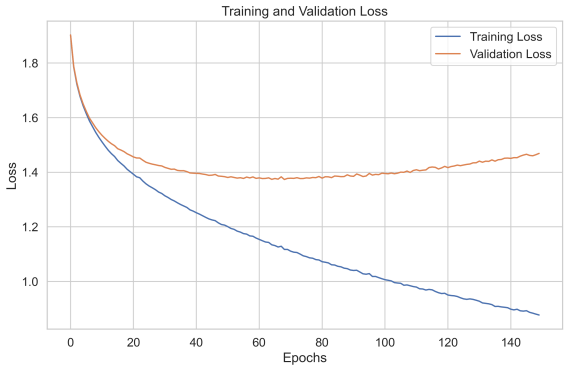

Figure 14: Learning Rate = 0.005

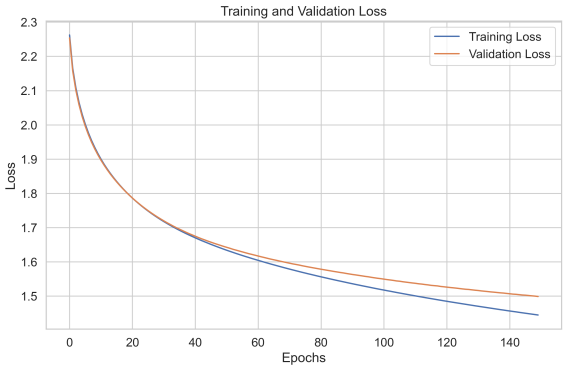

Figure 15: Learning Rate = 0.0005

Figure 16: Training Loss V.s. Validation Loss w.r.t learning rates

3.3.2 Activation Function

No.	Activation Function	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	(relu, relu)	0.871	69.94	1.483	52.12	750.14
2	(tanh, tanh)	1.145	61.57	1.618	46.81	751.45
3	(leakyrelu, leakyrelu)	0.880	69.57	1.474	52.05	739.19

Table 2: Experiment results when using different activation functions: ReLU, tanh, Leaky ReLU

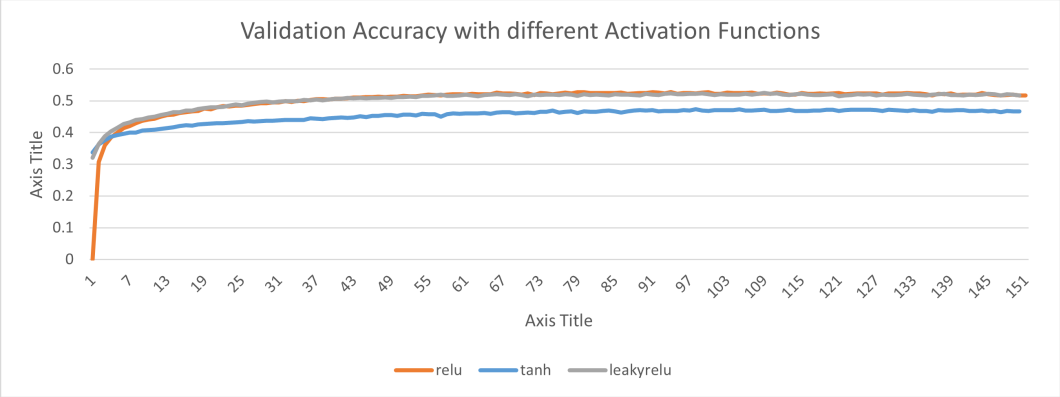

Figure 17: Validation Accuracy using different Activation Functions

3.3.3 Batch sizes

No.	Batch size	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	1	0.725	75.05	3.357	46.36	2424.92
2	256	0.653	77.75	1.805	50.55	763.45
3	512	0.878	69.84	1.480	51.71	759.89
4	1024	1.083	62.35	1.389	51.96	750.94

Table 3: Experiment results when using different batch sizes: 1, 256, 512, 1024

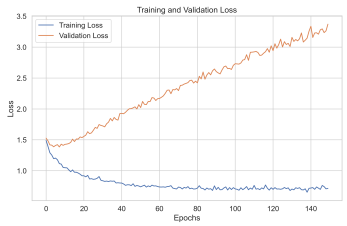

Figure 18: Batch size = 1

Figure 19: Batch Size = 256

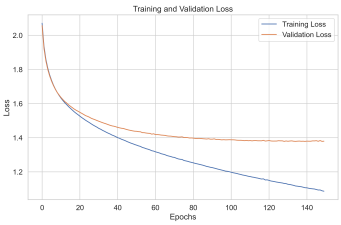

Figure 20: Batch Size = 1024

Figure 21: Training Loss V.s. Validation Loss w.r.t batch sizes

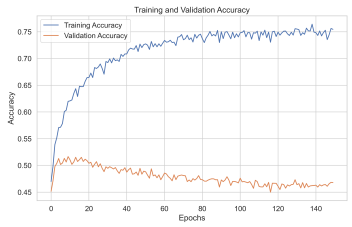

Figure 22: Batch size = 1

Figure 23: Batch Size = 256

Figure 24: Batch Size = 1024

Figure 25: Training Accuracy V.s. Validation Accuracy w.r.t batch sizes

3.3.4 Number of layers

No.	Number of Layer Neurons	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	(128, 150, 10)	1.090	62.42	1.373	52.49	555.37
2	(128, 150, 100, 10)	0.872	69.83	1.469	51.90	748.61
3	(128, 256, 150, 100,10)	0.331	89.62	2.487	49.52	1106.29

Table 4: Experiment results when using a different number of layer neurons: (128, 150, 10), (128, 150, 100, 10), (128, 256, 150, 100,10)

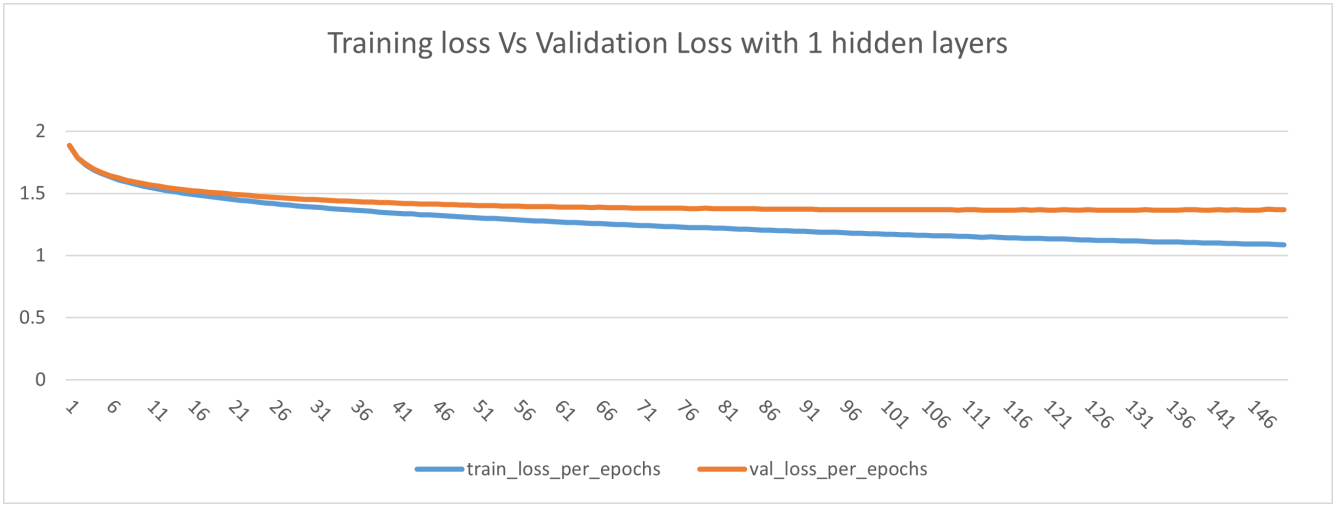

Figure 26: No.layers = 1

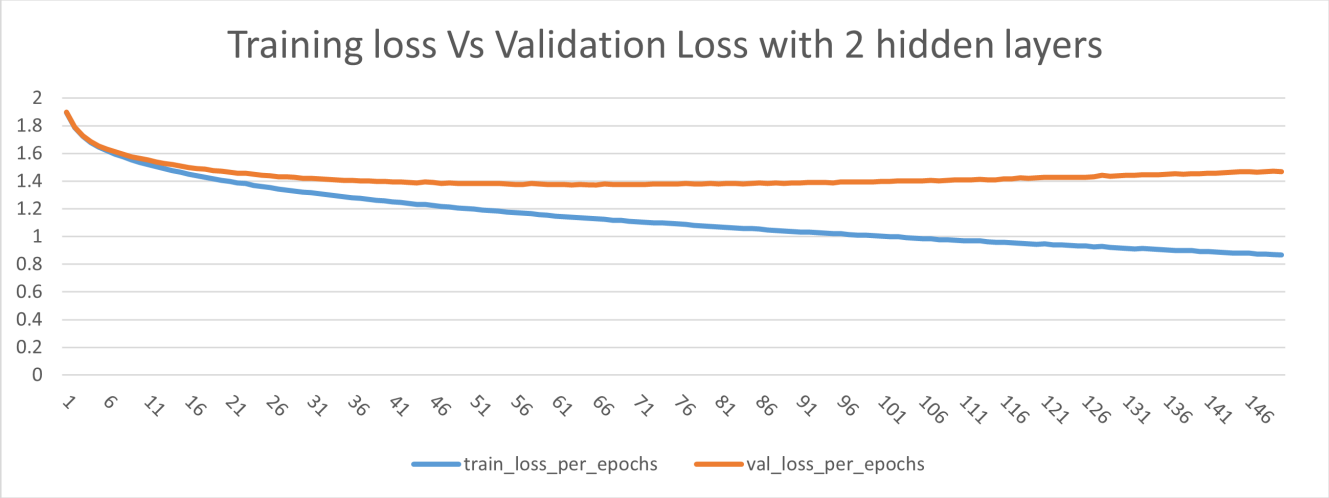

Figure 27: No.layers = 2

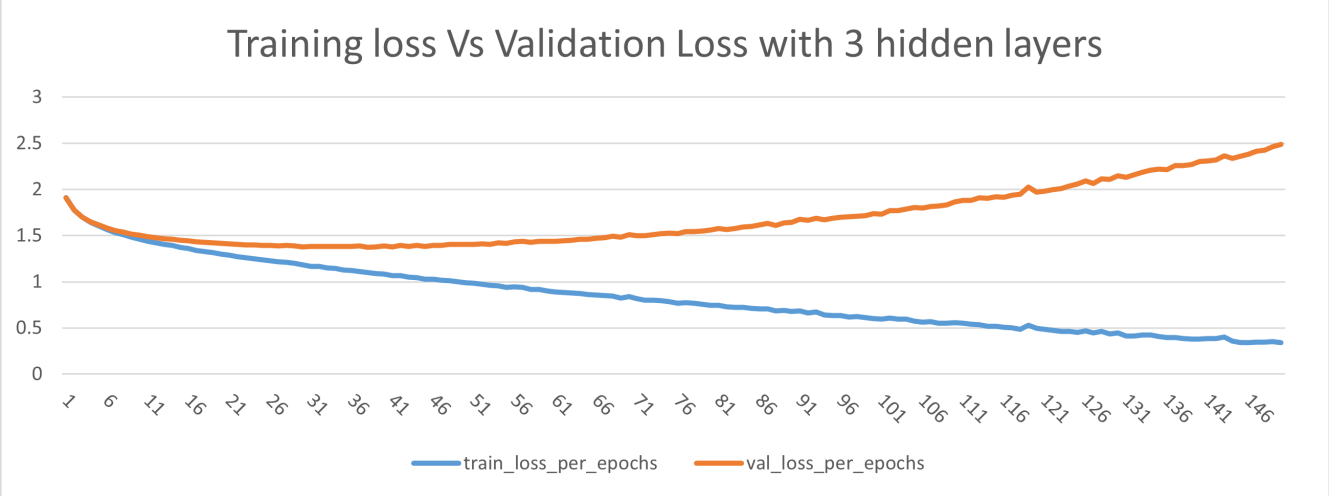

Figure 28: No.layers = 3

3.3.5 Optimizers

No.	Optimiser	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	SGD_Momentum	0.873	70.01	1.477	51.96	704.3
2	RMSprop	0.287	89.81	5.463	46.42	728.47
3	Adam	0.218	92.11	6.137	46.70	782.92

Table 5: Experiment results when using different optimizers: SGD_momentum, RMSprop, Adam

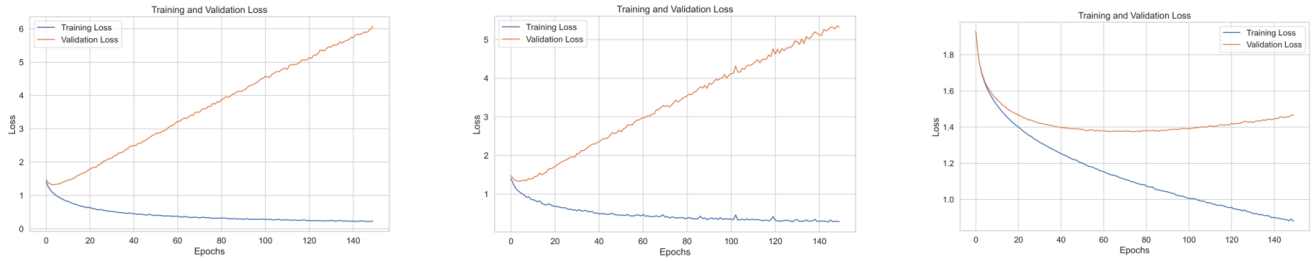

Figure 29: Training loss and validation loss for different optimizers: From left to right: Adam RMSprop, SGD_momentum

3.3.6 Batch Normalization

No.	Batch Norm or not	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	True	1.095	61.28	1.402	51.47	915.41
2	False	0.868	70.16	1.496	51.65	783.42

Table 6: Experiment results when using batch normalization or not

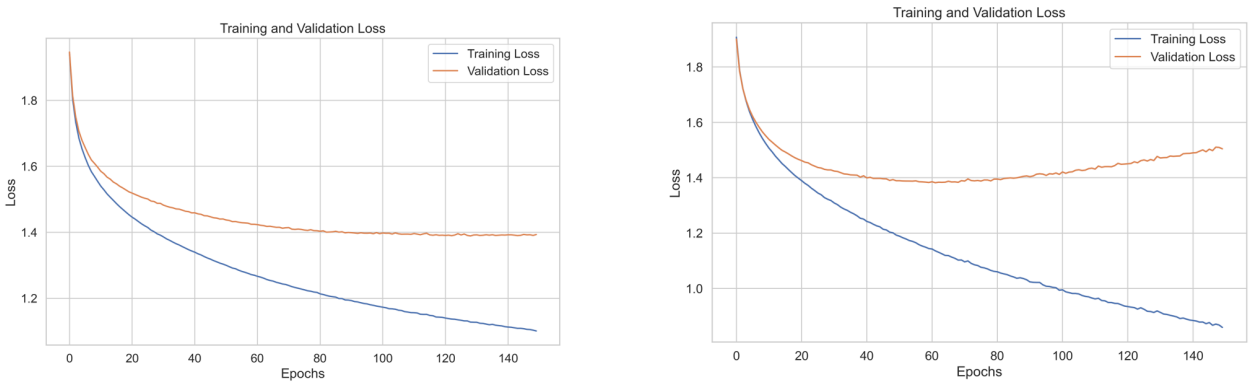

Figure 30: Training loss and validation loss for switching batch normalization: left: batch normalization is True, right, batch normalization is False

3.3.7 Dropout

No.	Dropout probability	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	1	0.661	52.33	1.103	39.02	789.89
2	0.9	0.815	47.43	0.969	41.01	785.46
3	0.7	1.043	41.34	1.084	38.99	696.35
4	0.5	1.263	36.41	1.271	35.71	749.82

Table 7: Experiment results when using different dropout rates: 1,0.9,0.7,0.5

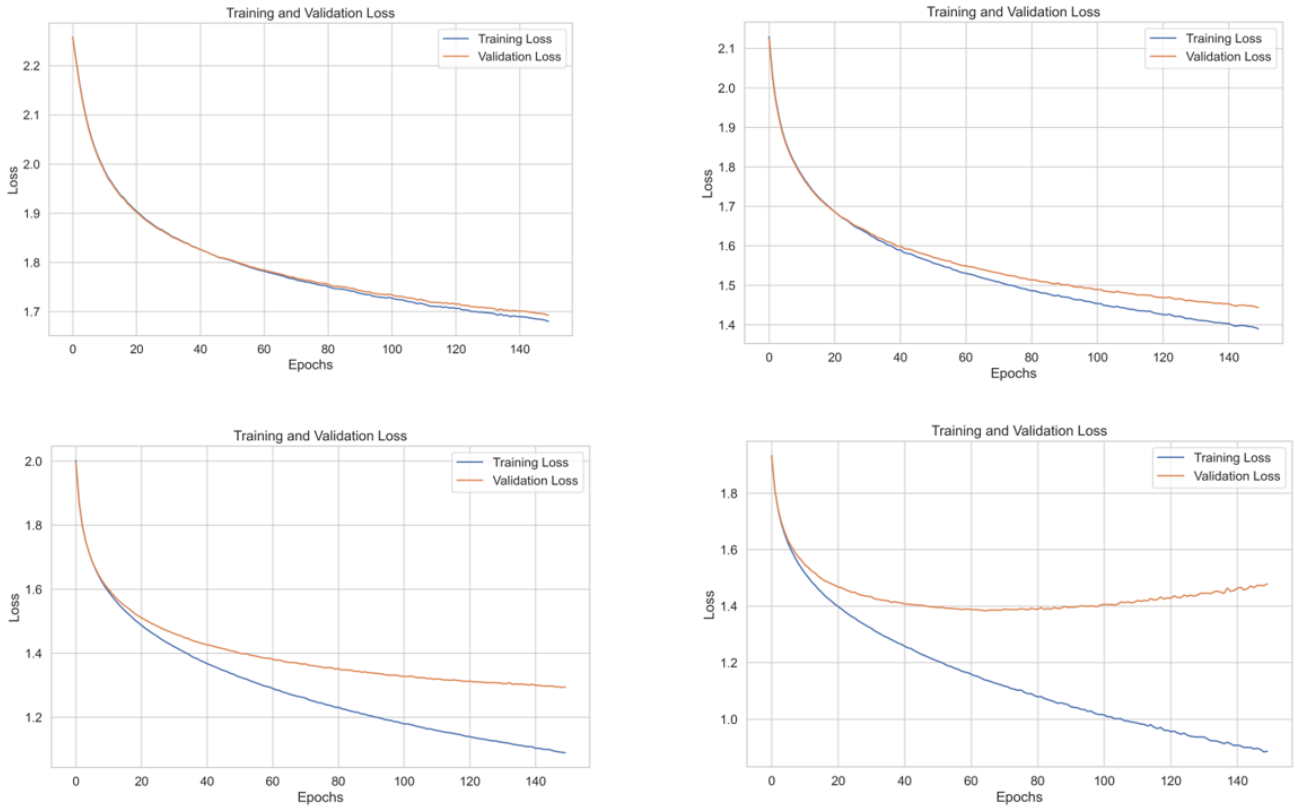

Figure 31: Training loss and validation loss for different dropout probability parameters: top left: 0.5, top right 0.7, bottom left: 0.9, bottom right: 1

3.3.8 Weight Decay

No.	Weight Decay	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)	Run Time(s)
1	0	0.872	69.94	1.479	52.33	792.62
2	0.02	1.573	45.11	1.577	45.01	763,94
3	0.04	1.825	35.67	1.822	35.59	690.58
4	0.06	1.969	27.58	1.968	27.11	738.3

Table 8: Experiment results when using different weight decay parameters: 0,0.02,0.04,0.06

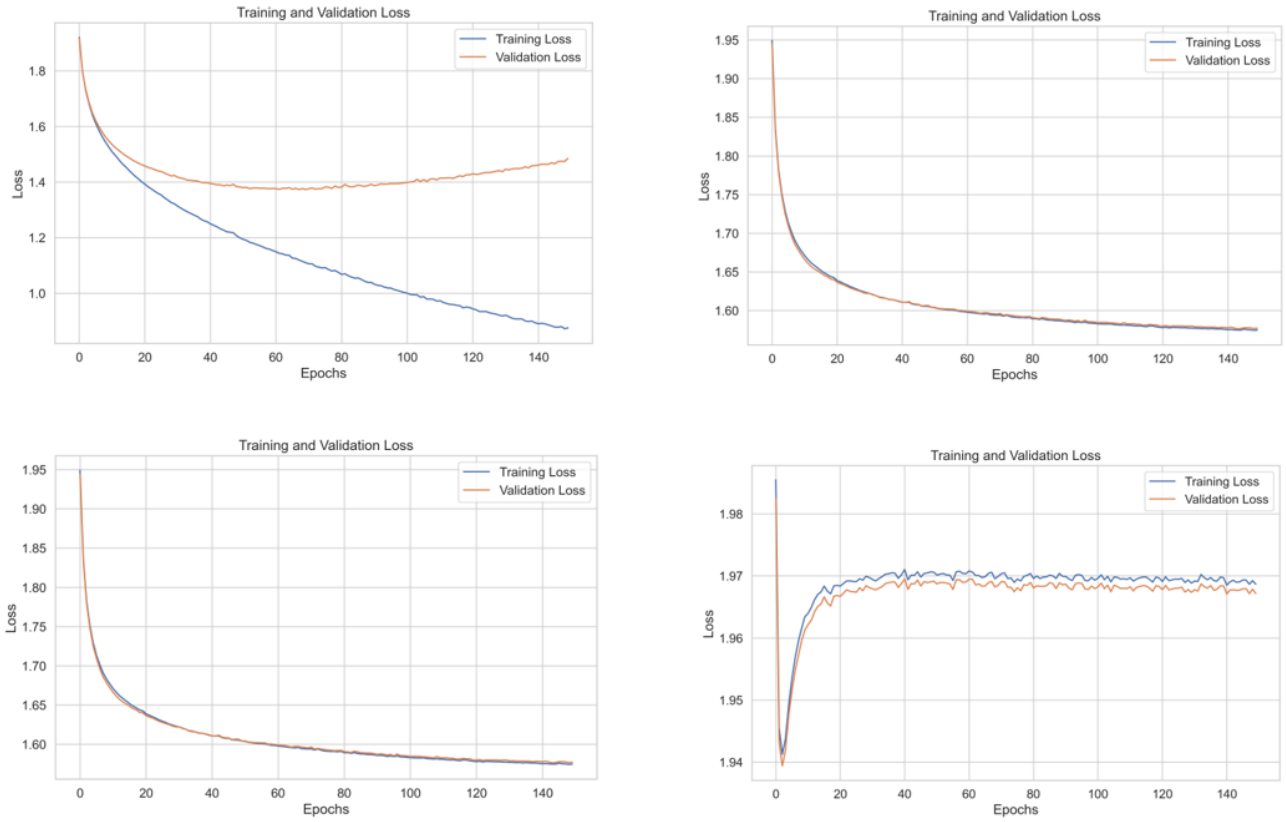

Figure 32: Training loss and validation loss for different weight decay parameter: top left: 0, top right 0.02, bottom left: 0.04, bottom right: 0.06

3.4 Experiment Observation and Analysis

- **Learning Rate:** Referring to the Table 1, the default models with four different learning rates have been trained for the purposes of comparison. The experiment of 0.005 using the optimizer of SGD with momentum achieves the best results in terms of validation accuracy which has an accuracy of 51.64 %. The learning rates of 0.0005 and 0.05 has results that are slightly inferior to the experiment of 0.005, reaching the validation accuracies of 47.71 % and 23.31 %. The running times across the experiments with different learning rates are almost the same, suggesting that the factor of learning rate does not affect the running time of MLP training when the setting of the MLP is controlled.
  For a more thorough investigation, we notice, from Figure 16, we found that the training loss and the validation loss experience frequent fluctuation of losses during the training of 150 epochs when the learning rate is larger like in the case of 0.5. As the learning rate decreases, the fluctuation of the losses reduces and tends to maintain smooth lines of a monotonically increasing curve. This behavior is expected as the learning rate decreases, the parameters of MLP in the space of loss space will be updated as smaller steps that reduce the changes in the losses in each iteration of the epoch. In addition, we observed the smaller learning rates for MLP, even though it needs longer training time to reach the same performance compared with the models with larger learning rates, the difference between validation loss and validation losses is smaller compared to other settings, implying that the chance of overfitting the model training can be largely reduced.

- **Activation function:** Referring to the Table 2, two types of ReLu-related activation functions do not exhibit a conspicuous difference in terms of model performance on our final validation actuaries by more than 0.5 shown in Figure 17%. However, the tanh activation function performs worse in comparison to others, only reaching the final validation accuracy of 46.81 %. One reason accounts for this inferior performance is that Relu-type activation functions are robust against the problem gradient vanishing as the tanh tends to cluster the activation value in the saturating regions while the relu activation functions simply preserve the positive value as it is, which might, otherwise, result in a 0 value after a long period of training time. Despite the Relu function not being affected by the gradient vanishing problem, simply rounding the negative value to zero imposes the problem of "dying Relu" sometimes which is undesirable in some cases. LeakyRelu addresses this problem by multiplying a small decimal number and keeping a proportion by a negative value instead of rounding it to zero. In the case of our experiments, we do not observe the occurrence of the "Dying Relu" Problem so there are no substantial differences between Relu and LeakyRelu activation functions.

- **Batch Size:** Referring to the Table 3, we explore the four choices of batch size in training an MLP, the experiment of a batch size of 1 serves as a controlled experiment to see the role of mini-batch training one sample point training. We observe that the techniques of mini-batch which have a batch size of more than 1 improve the computational efficiency by a huge margin and reduce the training time by a factor of 3. Additionally, the implementation of the mini-batch improves the generalization of the model on the given shown by a huge increase in terms of its validation accuracies and a substantial decrease in terms of validation loss. From Figure 21 and Figure 25, it is noted that the experiments trained with a smaller batch size converge very quickly and are followed by an obvious rebound in training loss, revealing the problem of over-fitting occurs severely. In contrast, the behavior of the over-fitting problem mitigates as the batch size increases and the convergence rate decreases, and the number of iterations and updating of parameters reduces in one epoch of training. Moreover, the large batch size gives a more reliable direction in gradient descent using SGD with mini-batch techniques, leading to a more smooth and more stable trajectory of training loss and validation loss over the training time. Overall, we can see that batch size is an important hyper-meter to choose from in the MLP training. The general rule is that a large batch size will improve an update of parameters in a more correct direction, but the bottleneck of the large batch size is the limitation posed by the hardware conditions and computational resources, therefore, sometimes, we decide to reduce the batch size to achieve a faster convergence rate at the expense of having less stable training curves and increasing the change of being overfitting the dataset.
- **Number of layer neurons:** Referring to the Table 4, we set up the control experiments against the three sets of predefined network architecture, with each set of neurons increasing by one layer. We found that the number of Layer Neuros in an MLP is an important factor affecting the running time and it is reasonable to see why this is the case. As adding ao the layers to MLP essentially increase the number of parameters of the model, in principle, is equivalent to making the model more complex and larger, harder to update. Meanwhile, although a complex model increases the representational power of fitting the pattern, it inevitably suffers from higher demand for training time and computational resources as a result, similar disadvantages related to hardware conditions are mentioned in the case of the higher batch, we are here to skip them though. Empirically, we observe an increase over the training time of more than 60% when the number of layers increases from 2 to 3 layers, while the final validation accuracy even decreases by 2 percent from 51.90% to 49.52%. This happens because the absence of regularisation is in our default setting of the MLP for these experiments and it is not surprising to see the over-fitting occur in this case and the magnitude of the model over-fitting intensifies upon the number of layer neurons, and this pattern is clearly shown in Figure 28, Figure 27 and Figure 26.
- **Optimizer:** Referring to the Table 5, three optimizers have been deployed for comparison, the validation accuracy of Stochastic gradient descent for momentum achieves the highest value of 51.96%, while RMSprop reaches the lowest with 46.42%.

  Additionally, the training time for Adam optimizer and RMSprop optimizer are slower than SGD_momentum optimizer, with 783s, 728.5s, and 704.3s respectively.

  The result is unexpected since the Adam optimizer is theoretically superior to other optimizers in terms of validation accuracy and execution time as discussed above.

  For further investigation, Figure 29 has shown the training loss and validation loss of the proposed model using different optimizers. While Adam and RMSprop demonstrate a drastic increasing trend for validation loss starting from the 20th epoch approximately, the validation loss of the SGD_momentum didn't increase until the 100th epoch. Thus, it is claimed that the unsatisfying experiment performance of the Adam optimizer and RMSprop optimizer stems from the overfitting problem. Therefore, an early stopping procedure can be deployed to stop the training process at the appropriate time required for the best model.
- **Batch Normalization:** Referring to the Table 6, the validation accuracy when using batch normalization for the model is 51.47%, costing 915s for training, while the validation accuracy without batch normalization reaches 51.65%, causing 783.4s for training. The validation accuracy without batch normalization is better. Moreover, Batch normalization might result in longer training time due to extra normalization parameters for training without a dramatic increase in accuracy in terms of the experiment performance. Furthermore, Figure 30 has also illustrated that the model using batch normalization keeps a constantly decreasing trend for validation loss meaning this model is appropriately regularized, while the other model demonstrates the overfitting issue, which also proves the regularization merit for the model using batch normalization, even though the regularized model doesn't perform expectedly. One potential reason can be attributed to the nature of the normal-distribution-shaped empirical data distribution.
- **Dropout Probability**[2]**:** According to Table 7, the validation accuracy illustrates a slight increase first, followed by a significant decrease in the decrease of dropout probability at around 35% roughly with the increase of dropout probability, while the execution time for each model is maintained at around 750s as well, due to less computational-intensive implementation of the module. For further investigation, Figure 31 has drawn the training loss and validation loss for each parameter setting. Dropout has also shown its' regularization behavior when the dropout probability is enhanced accordingly. The model with the lowest retaining rate demonstrates a similar decreasing loss behavior between training loss and validation loss, followed by the increase of validation loss accompanied by the increase of dropout probability. This has also shown the regularized behavior from dropout probability, while the model is suspected to overfit when the dropout module is removed, as shown in the bottom right corner of Figure 31

[2]Dropout Probability we used for our project is the Probability of preserving a neuron, this definition might differ from other literature.

- **Weight decay:** As shown in the Table 8, the validation accuracy decrease with the increase of the weight decay parameter from 0 to 0.06, with approximately similar execution time at around 750 seconds. The execution time is similar due to computational-efficient implementation for the weight decay module since its' equivalence with l_2 regularization module. While the shrinkage of validation accuracy might be due to different aspects such as the simple data distribution or other hyperparameter tunings. Moreover, with the weight parameter increase, Figure 32 indicates the validation loss maintains descending due to the generation of a more regularized model, showing the effectiveness of the regularization of the weight decay module. However, the weight decay parameter should be chosen carefully to prevent over-regularized behavior according to the empirical performance when the weight decay parameter is maximized to 0.06 in our experiment, providing strong evidence of demonstrating an underfitting problem(performance is unsatisfactory in the train set and test set) in terms of the descending behavior for both training loss and validation loss from the right corner of the Figure 32.

3.5 Comparison Methods(Combine different modules)

Modules	Train. Loss	Train. Acc(%)	Val. Loss	Val. Acc(%)	Run Time(s)
SGD	0.873	69.88	1.486	51.64	736.62
SGD + Weight Decay(0.002)	1.15	59.85	1.33 3	53.08	883.96
Adam + Weight Decay(0.002)	0.87	70.17	1.24	55.81	997.83
Adam + Weight Decay(0.002) + Dropout (0.9)	1.045	65.36	1.22	56.68	897.60
Adam + Weight Decay(0.002) + Dropout (0.9) + Early Stopping(20 epochs)	1.1058	62.50	1.2574	55.76	132.63

Table 9: Comparative Studies of Modules in MLP

In this subsection, we aim to experiment with the effect after combing modules sequentially, including weight decay, Adam optimizer, batch normalization, dropout, and early stopping. We seek the model with the most satisfactory performance mainly according to the validation accuracy, supported by other metrics such as validation loss and execution time.

3.5.1 Basic model

As a baseline approach and a method for comparative study, this approach achieves a 51.64% validation accuracy with a descent execution time of 736.62s. Nevertheless, it is interesting to observe that the training loss is moderately lower than that of validation loss, which indicates a potential overfitting issue.

3.5.2 Basic Model + Weight Decay

Due to the potential overfitting problem mentioned in the last subsection, we aim to further improve the validation accuracy by mitigating the effect of the overfitting issue to achieve a more regularized model. According to the ablation study in the previous section, it is discovered that the advantage of weight decay over all other regularization methods, particularly for the case when the weight decay parameter is at 0.02, achieves a tradeoff between a regularized model and model complexity. To further enhance validation accuracy, it is purposed that the 0.02 will be diminished further to 0.002 to explore the improvement of validation accuracy. According to the conclusion of the ablation study in the weight decay section, a large weight decay parameter might lead to an underfitting problem, while a small weight decay parameter might be useless for regularization. A smaller weight parameter: 0.002 is selected instead, which is consistent with our hyperparameter analysis that a smaller parameter might achieve a decent validation accuracy.

3.5.3 Basic Model + Weight decay + Adam

Replacing the optimizer from SGD_momentum to Adam despite the unsatisfactory validation loss for the Adam optimizer in the ablation study before, it is believed that the model that changes the Adam optimizer only suffers overfitting in subsection 3.4. After alleviating the overfitting effect, it is expected to observe a better performance than the aforementioned result. The combined experiment result as shown in Table 9 has demonstrated a significant improvement of validation accuracy by 2.73% with no considerable amount of increase in execution time as expected, despite a potential overfitting problem still exists.

3.5.4 Basic Model + Dropout + Weight Decay + Adam

To further alleviate the overfitting issue, the dropout strategy is embedded to observe if they can achieve better chemistry and regularized model. The dropout probability is set as 0.9 which achieves the balance between model complexity and generalization as suggested by the ablation study for dropout. The result indicates a better performance than before from 558.1 to 56.68, with a compromise between generalization and model complexity and a moderate training time (897.6s).

3.5.5 Basic Model + Weight Decay + Dropout + Adam + Early Stopping

It is obvious to notice that the running time for all of the previous models cost more than 700 seconds to train. Early stopping has been deployed to minimize the training time and validate if the model could achieve a similar performance using minimum

Name of metric	Value
Training Loss	1.1058
Training Acc	62.50%
Validation Loss	1.2574
Validation Acc	55.76%
No.Epochs of Early Stopping	20
Running time(s)	132.63s

Table 10: Model Results for our final proposed model

execution time. The experiment results suggest a significant enhancement of the time efficiency of the model, without a drastic decrease in model accuracy, at 55.76%, enabling this model as an optimal design model of our suggestion.

3.6 Settings For The Best Model

Aspect	Hyper-parameters Name	Setting
Neural Network Architecture	Number of Hidden Layers	2
	Layer Neurons	$[128, 150, 100, 10]$
	Activation Functions	Relu
Optimisation	Optimizer	Adam
	Optimizer Parameter(s)	$\beta = (0.9, 0.999)$
	Learning Rate(lr)	0.005
	Loss function	Cross Entropy with Softmax
	Batch Size	1024
Regularisation	Dropout Probability	0.1
	Weight Decay	0.002
	Batch Normalisation	False
	Early Stopping	20 Epochs

3.7 Justification for The Best Model

In this section, we present our best optimal model detailed in subsection 3.6 based on the comparison methods from previous experiments. To tailor the model to the given data, we configure it with two hidden layers and a network architecture of [128, 150, 100, 10]. Since the "Dying ReLU" issue was not observed in our experiments, we opt for the "ReLU" activation function over the "LeakyReLU" to improve computational efficiency. Regarding the optimizer selection, our first choice is the Adam optimizer with parameters of $\beta = (0.9, 0.999)$, we believe the design principles of the Adam optimizer provide robustness and reliability to the optimization process, such as escaping the local minima. To address potential issues of excessive training time, we employ early stopping in the final model training. This approach allows us to fully utilize the Adam optimizer's convergence speed without overfitting the model and wasting computational resources. In terms of regularization techniques, we follow the experimental insights and fine-tune the model with appropriate hyperparameters. Specifically, we lower the dropout probability to 0.1, and since batch normalization is not utilized, we enable the Research version's weight decay of 0.002 to prevent uneven parameter updates in the MLP.

As a result, our proposed optimal model achieves the highest validation accuracy of 55.76 % among all experiments in this project. Thanks to the implementation of early stopping, the ideal number of epochs is determined to be around 20, while we do not employ the best model for full training in the above comparison experiments. The details of our results for this model are presented in Table 10.

4 Self-reflection and Conclusion

4.1 Reflection

There are still limitations in our work. Firstly, more combinations of module modification can be examined and compared with results from default models, which can provide more pervasive experimental reasoning and comparison. Secondly, more visualization for comparing the validation classification accuracy can be drawn for better interpretation and explanation of the experiment results. Thirdly, the design of the experiments can be more effective in terms of examining the usage of the techniques of early stopping.

4.2 Conclusion

In this project, we have developed strong coding skills for building a neural network from scratch. Apart from that, we have touched on the foundation of deep learning technology: multilayer perceptron, while examining the functionality, theoretical

principles, and experiment setup of each module, equipping us with a solid theoretical guarantee to understand more complicated deep learning models and modules. We have conducted an intensive empirical analysis of the performance of each module while holding another experiment setup as the default setting. We have also examined the effect of combining modules and obtaining the best design model according to the comparative experiment, noticing weight decay and optimizer are the most pivotal modules for achieving high accuracy and generalization. It is observed that suitable regularization techniques such as Weight decay, dropout, and batch normalization can improve the generalization of the machine learning algorithms and remedy the issue of overfitting. Additionally, the trade-off between the choice of hyperparameters such as dropout probability can also make a huge difference to the output model, thus having to be chosen after intensive experiments. Finally, the performance of the "normal" model with appropriate regularization doesn't lead to the corresponding high classification accuracy. This can be contributed to the initial experiment's default setting, randomization of initialization, or the effect from the almost normally-distributed dataset. More experiment setups and modules such as the Adadelta optimizer can be implemented and assessed in future research work.

References

[1] Léon Bottou. "Online algorithms and stochastic approxima-p tions". In: *Online learning and neural networks* (1998).

[2] Tracy Chang. "Implementing Batch Normalization in Python". In: *Towards Data Science* (Jan. 2020). Accessed: 2023-03-25. URL: https://towardsdatascience.com/implementing-batch-normalization-in-python-a044b0369567.

[3] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[4] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[5] Anders Krogh and John A. Hertz. "A Simple Weight Decay Can Improve Generalization". In: *NIPS*. 1991.

[6] Zafarullah Mahmood. *Training Deep Neural Networks with Batch Normalization*. Accessed: 2023-03-25. 2023. URL: https://zaffnet.github.io/batch-normalization.

[7] Naveen. *What is GD, Batch GD, SGD, Mini-Batch GD?* https://www.nomidl.com/machine-learning/what-is-gradient-descent-batch-gradient-descent-stochastic-gradient-descent-mini-batch-gradient-descent/. Accessed: 2023-03-25. Aug. 2022.

[8] Lutz Prechelt. "Early stopping—but when?" In: *Neural networks: tricks of the trade: second edition* (2012), pp. 53–67.

[9] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408.

[10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699. ISBN: 0262010976.

[11] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[12] Chang Xu. *Week 2 Lecture Slides: Multilayer Neural Network*. Lecture slides. Available from: School of Computer Science, University of Sydney. School of Computer Science, University of Sydney, 2023.

[13] Chang Xu. *Week 3 Lecture Slides: Optimization for Training Deep Models*. Lecture slides. School of Computer Science, University of Sydney, 2023.

[14] Chang Xu. *Week 4 Lecture Slides: Regularizations for Deep Models*. Lecture slides. Available from: School of Computer Science, University of Sydney. School of Computer Science, University of Sydney, 2023.

5 Appendix

A Links to Codes and Raw Data

- Colab Link
- Link to the folder of data and Colab

B Intstructions to run the codes

1. The implementation of MLP used for Deep learning assignment 1 at the University of Sydney adopted the base code provided in the DL tutorial. We extend the implementation to meet the criteria specified by the assessment.
2. We present a runnable Jupyter notebook in the above link for submission and demonstration purposes. We used the python files to run the experiment on the local machine, the results are provided in the report.
3. To run the notebook, make sure that you run the code block sequentially, this is a crucial step you have to follow!
4. **Note that you might be asked to log in to a google account to gain permission to download the data files. If so, please do not hesitate to do so. Otherwise, the notebook can not be executed successfully.**