# INT304: LAB3 REPORT - FINAL COURSEWORK

**Seungyeop Kang**
Student ID: 1825478

## ABSTRACT

In machine learning, there are mainly two different classification tasks (i.e., binary classification and multi-class classification). Depends on the required task, the proper classification algorithm can be decided to maximise the accuracy and efficiency. In this coursework, three famous classifiers, which are Multilayer Perceptron (MLP), K-Nearest Neighbors (KNN) and Support Vector Machine (SVM), are required to complete a clothing image classification task with the Fashion-MNIST dataset.

## 1 INTRODUCTION

Classification is a process of categorizing a given structured or unstructured dataset into correct classes. Generally, classification can be defined as two types. If it has two outcomes, then it is called binary classification. On the other hand, another type is called multi-class classification because it has more than two classes (Waseem, 2022). In this coursework, we are required to implement the clothing image classification which is multi-class classification.
To conduct this task, the first required classifier is the Multilayer Perceptron (MLP). MLP is a type of Artificial Neural Network (ANN) (Simplilearn, 2022). Here, perceptron is a binary linear classifier used in supervised learning. It consists of four components (i.e., input layer, weights and bias, weighted sum and activation function) as shown in Fig 1. Since it can classify the given input data into a certain binary output based on the unit step function, perceptron is also called a single-layer neural network (Bhardwaj, 2020).

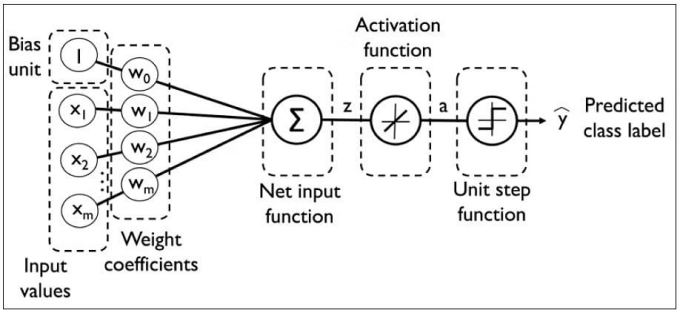


Figure 1: Single-layer ANN (Simplilearn, 2022).

Then, MLP is an artificial neural network consisted of more than one layer with perceptron. The basic structure of MLP has three layer types (i.e., input layer, hidden layer and output layer). By using linear and non-linear activation functions, we can train the given dataset and predict multiple classes of the test values (Simplilearn, 2022).
The second classifier is the K-Nearest Neighbors (KNN) algorithm. KNN is a non-parametric classifier based on supervised learning. It simply uses proximity of the train dataset to predict or classify about the grouping of data points (Sharma, 2021).
Lastly, Support Vector Machine (SVM) will be used to separate the dataset via a hyperplane based on different kernels, such as polynomial kernel and radial basis function kernel (RBF). Based on the defined hyperplane, the test data can be classified (Gandhi, 2018).
After implementing these algorithms, finally, the accuracy and efficiency results of each classifiers will be compared and analysed.

## 2 METHODOLOGY

### 2.1 FASHION-MNIST DATASET

Fashion-MNIST is a dataset provided in Zalando's article images. The total number of examples in this dataset is 70,000 consisting of 60,000 training samples and 10,000 test samples. All images are $28 \times 28$ grayscale with 10 labels. Fig 2 and Fig 3 show the example images and complexity of the Fashion-MNIST. The main purpose of using this dataset is to test algorithms with more complex dataset because the original MNIST can achieve high accuracy easily and overused in many papers (Xiao et al., 2017).

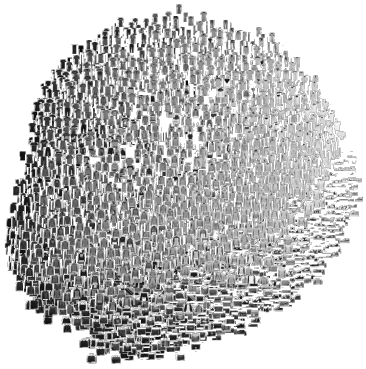

Figure 2: PCA on Fashion-MNIST (Xiao et al., 2017).

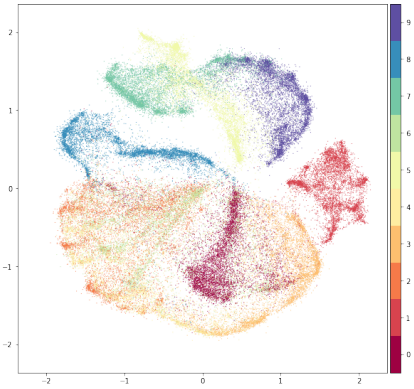

Figure 3: PyMDE on Fashion-MNIST (Xiao et al., 2017).

### 2.2 MULTILAYER PERCEPTRON (MLP)

In the previous section, the brief explanation of the Single-Layer Perceptron (SLP) was provided. The working principle of MLP is based on SLP but has more than one layer to consist the input, hidden and output layers. In this coursework, we are required to design a 3-layer MLP to perform the multi-class classification. It means that we need to design a neural network with one hidden layer. Thus, I designed the MLP model based on the requirement and drew a schematic diagram of my model as shown in Fig. 4:

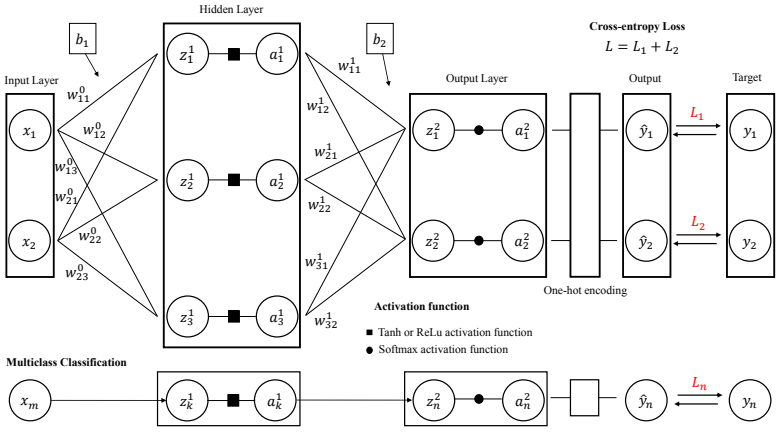


Figure 4: Self-designed model of MLP.

Note that it is important to check the parameters written on Fig. 4 because the following mathematical derivations are based on this model.
From the diagram, we can see that the MLP model consists of two different activation functions in the hidden and output layer to convert the linear relationship of the data in each layers to nonlinear. In the hidden layer, I decided to use a hyperbolic tangent (tanh) or ReLU functions. Tanh function is similar to sigmoid function but the centre of function is moved to the origin to optimize the weight better than a sigmoid function during the gradient descent. On the other hand, ReLU function simply outputs the values greater than 0 and replace negative values to 0. Unlike the sigmoid and tanh functions, ReLU can avoid the vanishing gradient problem. Hence, it is widely used in modern neural network designs (e.g., CNN and DNN). Note that ReLU can have a problem called dying ReLU sometimes (Bishop, 2006). The equations of two activation functions are provided as below:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad \& \quad \frac{\partial}{\partial z} tanh(z) = 1 - tanh^2(z) \tag{1}$$

$$ReLU(z) = max(0, z) \quad \& \quad \frac{\partial}{\partial z} ReLU(z) = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases} \tag{2}$$

After the hidden layer, the data will be processed in the output layer with the softmax activation function.

$$S(z_1, z_2, \cdots, z_n; z_j) = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}} \quad \& \quad \frac{\partial}{\partial z_i} S(z_j) = \begin{cases} S(z_j) \times (1 - S(z_j)) & if \quad i = j \\ -S(z_j) \times S(z_i) & if \quad i \neq j \end{cases} \tag{3}$$

The above equations are the softmax activation function and its derivative. Softmax function simply denotes the generalized version of the sigmoid function. Then, its output will be regularized as the probability between 0 and 1. That is, the sum of output layers is equal to 1. Hence, it is suitable for multi-class classification because we can determine the class based on the obtained probabilities. Additionally, I applied one-hot encoding based on the given labels. One-hot encoding will create new binary columns as per all the possible labels. Then, a label with the maximum probability will be left as 1 and others will be stored as 0. By converting numerical categorical variables into binary vectors, the performance of this model can be improved (Ganji, 2022).
After that, the loss function will calculate the errors of output values. In this model, I applied the categorical cross-entropy for the loss function as follows:

$$L = -\frac{1}{n} \sum_{i=1}^{n} y_i log(\hat{y}_i) \tag{4}$$

where $n$ is the number of the output size. Since the cross-entropy can consider a example to belong to a specific class with probability 1 and other categories with probability 0, this is suitable for classification tasks (Koech, 2020).
Now, we can derive the feed forward and backpropagation of this model. Firstly, as per Fig. 1, the input values will be computed as follows based on the forward propagation:

$$z_k^1 = b_1 + \sum_{i=1}^{m} w_{ij}^0 x_i \quad \Rightarrow \quad \frac{\partial z_k^1}{\partial w_j} = x_i, \quad \frac{\partial z_k^1}{\partial b_1} = 1 \tag{5}$$

$$z_n^2 = b_2 + \sum_{i=1}^{k} w_{ij}^1 a_i \quad \Rightarrow \quad \frac{\partial z_n^2}{\partial w_j} = a_i, \quad \frac{\partial z_n^2}{\partial b_2} = 1 \tag{6}$$

Here, $x$ is the input value and $z$ denotes the value passed the linear equation. By applying the non-linear activation function, $a$ can be obtained. $w$ and $b$ mean the weight and bias of each layer ($i$ and $j$ are used to describe the location of weights in each layer).
And then, the loss function can be calculated based on equation 4.
These are the general steps of the forward propagation. For the sake of simplicity, we will only check one example of each steps and assume that there are only two output classes from now. Then, we derive the

backpropagation based on this condition with only two output classes. Backpropagation propagates errors from the output layer to the input layer and updates the weights of each layer during the process. Generally, the weight can be updated as follows:

$$w = w - \alpha \times \frac{\partial L}{\partial w} \tag{7}$$

where $L_1 = \frac{1}{2}(y_1 - a_1^2)^2$, $a_1^2 = activation(z_1^2)$ and $z_1^2 = w_{11}^1 a_1^1 + w_{21}^1 a_2^1 + w_{31}^1 a_3^1$. Here, $\alpha$ is the learning rate. From this equation, we can notice that it requires the derivative of $L$ in terms of $w$. However, $L$ is not consisted of $w$. In this case, we apply the chain rule as follows:

$$\frac{\partial L_1}{\partial w_{i1}^1} = \frac{\partial L_1}{\partial a_1^2} \frac{\partial a_1^2}{\partial z_1^2} \frac{\partial z_1^2}{\partial w_{i1}^1} \tag{8}$$

To simply this equation, the equation can be rewritten as follows:

$$\frac{\partial L_1}{\partial w_{i1}^1} = \beta_1^2 \times a_i^1 \quad where \quad \beta_1^2 = \frac{\partial L_1}{\partial a_1^2} \frac{\partial a_1^2}{\partial z_1^2} \tag{9}$$

This can be satisfied because of equation 6. Then, now we multiply the learning rate to equation 9 and subtract this value from the weight to update the new weight after forward propagation. As the steps of $L_1$, the error of $L_2$ can be also updated as follows:

$$\frac{\partial L_2}{\partial w_{i2}^1} = \frac{\partial L_2}{\partial a_2^2} \frac{\partial a_2^2}{\partial z_2^2} \frac{\partial z_2^2}{\partial w_{i2}^1} = \beta_2^2 \times a_i^1 \quad where \quad \beta_2^2 = \frac{\partial L_2}{\partial a_2^2} \frac{\partial a_2^2}{\partial z_2^2} \tag{10}$$

After updating the weights between the hidden layer and the output layer, we can derive the weights propagated from the hidden layer to the input layer. Firstly, the gradient of the loss function in terms of $w_{11}^0$ can be derived as follows:

$$\frac{\partial L}{\partial w_{11}^0} = \frac{\partial L}{\partial a_1^1} \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^0} = \left( \frac{\partial L_1}{\partial a_1^1} + \frac{\partial L_2}{\partial a_1^1} \right) \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^0} \tag{11}$$

From this equation, we can notice that the errors propagated to the input layer will be influenced by both $L_1$ and $L_2$. By applying the chain rule, we derive the following equations:

$$\frac{\partial L_1}{\partial a_1^1} = \frac{\partial L_1}{\partial a_1^2} \frac{\partial a_1^2}{\partial z_1^2} \frac{\partial z_1^2}{\partial a_1^1} = \beta_1^2 \times w_{11}^1 \tag{12}$$

$$\frac{\partial L_2}{\partial a_1^1} = \frac{\partial L_2}{\partial a_2^2} \frac{\partial a_2^2}{\partial z_2^2} \frac{\partial z_2^2}{\partial a_1^1} = \beta_2^2 \times w_{12}^1 \tag{13}$$

Then, finally, the equation 11 can be redefined as follows:

$$\frac{\partial L}{\partial w_{11}^0} = \left( \frac{\partial L_1}{\partial a_1^1} + \frac{\partial L_2}{\partial a_1^1} \right) \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^0} = (\beta_1^2 w_{11}^1 + \beta_2^2 w_{12}^1) \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^0} = \beta_{11}^1 x_1 \tag{14}$$

where $\beta_{11}^1 = (\beta_1^2 w_{11}^1 + \beta_2^2 w_{12}^1) \frac{\partial a_1^1}{\partial z_1^1}$. Likewise the weight updating from the output layer to the hidden layer, we multiply the learning rate to equation 14 and subtract it from the current weight.
Based on this process, the designed MLP model can train and predict proper classes by repeating the feed forward and backpropagation as per the defined number of epoch (Bishop, 2006). Note that the real model has multiple number of input values, hidden nodes and output classes. Hence, the number of nodes in each layer can be referred to the multiclass classification section described in Fig. 4.

### 2.3 K-NEAREST NEIGHBORS (KNN)

Likewise the MLP, K-Nearest Neighbors (KNN) is also a type of the supervised learning algorithm widely used in both regression and classification models. The working principle of KNN is simple. It firstly calculates the distance between the test sample and all the training data points. And then, it selects the K number of the closest points from the test data. Based on the selected train points, it will calculate the probability of each classes and determine the appropriate class of the test point as per the highest class probability (Christopher, 2021).
In this coursework, we are required to use the Euclidean distance to measure the distance. Hence, the following equation will be used in the algorithm of KNN:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2} \tag{15}$$

After obtaining the K-nearest points, the posterior probability can be calculated as follows:

$$P(w_k|x) = \frac{K_i}{K} \tag{16}$$

where $K_i$ and $K$ denote the number of the $i$-th class data points in the selected group and the defined value of $K$, respectively. However, since the KNN algorithm requires significant computation cost to calculate the distance of data points, it is not necessary to calculate equation 16 because $K$ is same for every data points. Thus, we can neglect the division by $K$ and simply the equation as follows:

$$P(w_k|x) = K_i \tag{17}$$

Then, the predicted class of the test point will be determined as the class which appears the most in the selected data points (Bishop, 2006).

### 2.4 SUPPORT VECTOR MACHINE (SVM)

Support Vector Machine (SVM) is a classification algorithm to find a hyperplane in N-dimensional space to classify the data points. It is normally a binary classification method. Thus, it can separate the two classes of data points. However, there are various possible hyperplanes that could be selected. Therefore, the main purpose of this algorithm is to find a hyperplane that can maximise the margin space between both classes (Bishop, 2006). In this section, I will briefly explain the SVM design.
Firstly, I drew two figures to show the margin maximization of SVM with two different cases.

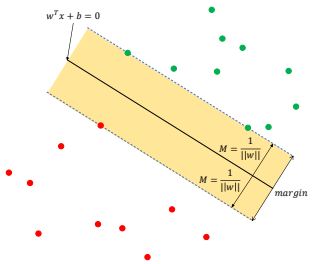


Figure 5: Hard-margin SVM.

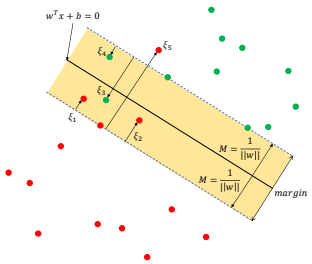


Figure 6: Soft-margin SVM.

Then, we define the hyperplane as follows:

$$hp = w^T x + b \tag{18}$$

From the above figures, we can see that a hyperplane separates the two classes. If Hyperplane is a subspace of "P-1" dimension, it can separate P-dimensional space. However, as we can see from the figures, there could be a lot of hyperplanes. Therefore, the hyperplane should maximize the distance between each classes for better classification performance, and that is called margin maximization. The equations below describe the hyperplane in two-dimensional space:

$$y = ax + b \tag{19}$$

Then, we can vectorize it as:

$$x_2 = ax_1 + b \tag{20}$$

Based on this equation, we can derive the following equations:

$$ax_1 - x_2 + b = 0 \tag{21}$$

$$\begin{bmatrix} a & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b = 0 \tag{22}$$

Finally, it can be sorted out as below:

$$w^T + b = 0 \tag{23}$$

where $w = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^T$ and $x = w = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$. Here, the vectors are transposed because general vectors are column vector. There vectors are perpendicular to the line we defined previously. Note that the slope of the line will be assumed as a constant (Matsuzaki, 2020).
Now, we calculate the distance between the hyperplane and a object point as follows:

$$d = \frac{|w^T x + b|}{||w||} \tag{24}$$

where $||w|| = \sqrt{w_1^2 + w_2^2}$. Since the classification performance will be better if the margin is larger, we can maximize $d$. Firstly, from Fig. 5 and Fig. 6, the classification result of the hyperplane will be as follows:

$$\begin{cases} w^T x_i + b > 0, y_i = +1 \\ w^T x_i + b < 0, y_i = -1 \end{cases} \tag{25}$$

This equation can be simplified as:

$$s.t. \quad y_i(w^T x_i + b) > 0 \quad where \quad i = 1, 2, \cdots, n \tag{26}$$

Based on the above equations, the maximum interval can be derived as follows:

$$\begin{cases} \underset{w,b}{max} \ \underset{x_i}{min} \frac{|w^T x + b|}{||w||} = \underset{w,b}{max} \ \underset{x_i}{min} \frac{1}{||w||} \cdot y_i(w^T x_i + b) \\ s.t. \quad y_i(w^T x_i + b) > 0 \quad where \quad i = 1, 2, \cdots, n \end{cases} \tag{27}$$

Here, we apply the minimum to improve the classification accuracy for different categories and the maximum distance to obtain the maximum interval to optimize the SVM model. For the maximal hyperplane, its geometric distance from all sample points should be the minimum distance to satisfy the margin condition. Hence, the following equation should be satisfied:

$$\exists \gamma > 0, \quad s.t. \quad \underset{x_i}{min} \ y_i(w^T x_i + b) = \gamma \tag{28}$$

Then, by setting $\gamma = 1$, we can notice that the hyperplane will not move and the coefficient can be scaled in the same proportion if we divide both sides of the equation in the mean time and reduce $w$ by times. Then, we can obtain the following equations:

$$\begin{cases} \underset{w,b}{max} \frac{1}{||w||} \ \underset{x_i}{min} \ y_i(w^T x_i + b) \rightarrow \underset{w,b}{max} \frac{1}{||w||} \rightarrow \underset{w,b}{min} \ \frac{1}{2}||w||^2 \\ s.t. \quad \underset{x_i}{min} \ y_i(w^T x_i + b) = 1 \rightarrow s.t. \quad y_i(w^T x_i + b) \geq 1 \end{cases} \tag{29}$$

In this paper, we assume that the maximization and minimization are equivalent for the optimization process. Then, we simply the equations as below:

$$\begin{cases} \min\limits_{w,b} \frac{1}{2}||w||^2 \\ s.t. \quad y_i(w^T x_i + b) \geq 1 \quad where \quad i = 1, 2, \cdots, n \end{cases} \tag{30}$$

Here, the first equation is the objective function and the second equation is the constraint condition in the optimization problem. Also, according to Fig. 6, we derive the equations of soft-margin SVN as follows:

$$\begin{cases} \min\limits_{w,b} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i \\ s.t. \quad y_i(w^T x_i + b) \geq 1 - \xi_i \quad where \quad i = 1, 2, \cdots, n \ \ and \ \ \xi_i \geq 0 \forall i \end{cases} \tag{31}$$

where $C$ is a positive trade-off parameter and $\xi_i$ is a slack variable. However, these are typical Quadratic Programming problems. Hence, we need to solve the optimization problems (i.e., constrained and unconstrained optimization). We can further divide the constrained optimization into the inequality and equality constrained optimization (Matsuzaki, 2020).
For the equality constrained optimization problems, we can apply Lagrange multiplier method to transform this into unconstrained optimization problem. About the inequality constrained optimization problem, we apply KKT condition to transform it into the unconstrained optimization problem. Lastly, for the unconstrained optimization problems, we simply take the derivative of the function and convert it to zero. Then, we can select the optimal value among the candidates and verify it. The model we derived is a inequality constrained optimization problem. Hence, the Lagrange multiplier method, dual problem and KKT condition will be introduced as follows. Firstly, according to Lagrange multiplier, we transform the derived optimization problem as below:

$$L(w, b, \lambda) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \underset{\lambda_i \geq 0}{\lambda_i} \ (1 - \xi_i - y_i(w^T x_i + b)) - \sum_{i=1}^{n} \underset{\gamma_i \geq 0}{\gamma_i} \ \xi_i \tag{32}$$

Through Lagrange function, the above equations will be simplified as follows:

$$\begin{cases} \min\limits_{w,b} \max\limits_{\lambda} L(w, b, \lambda) \\ \quad s.t. \quad \lambda_i \geq 0 \end{cases} \tag{33}$$

And then, we can transform this problem into a dual problem as per the duality.

$$\begin{cases} \max\limits_{\lambda} \min\limits_{w,b} L(w, b, \lambda) \\ \quad s.t. \quad \lambda_i \geq 0 \end{cases} \tag{34}$$

Then, the objective function will be strongly dual (equivalent) and satisfy the condition below:

$$\min_{w,b} \max_{\lambda} L(w, b, \lambda) \geq \max_{\lambda} \min_{w,b} L(w, b, \lambda) \tag{35}$$

After transforming into a dual problem, we find the partial derivative of above functions:

$$\frac{\partial L}{\partial b} = 0 \rightarrow \sum_{i=1}^{n} \lambda_i y_i = 0 \tag{36}$$

$$\frac{\partial L}{\partial w} = 0 \rightarrow w = \sum_{i=1}^{n} \lambda_i y_i x_i = 0 \tag{37}$$

$$\frac{\partial L}{\partial \xi_i} = 0 \rightarrow \lambda_i + \gamma_i = C \rightarrow 0 \leq \lambda_i \leq C \tag{38}$$

By substituting these equations into equation 32, we can obtain the following equation:

$$L(w, b, \lambda) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j \tag{39}$$

From this, the final optimization problem after the derivation can be obtained as follows:

$$W(\lambda) = \underset{\lambda}{max} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j \Leftrightarrow \underset{\lambda}{min} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j - \sum_{i=1}^{n} \lambda_i \tag{40}$$

where this equation is subject to $C \geq \lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$. Thus, based on the final optimization problem, $w$ can be recovered as $w = \sum_{i=1} \lambda_i y_i x_i$. However, this is still a Quadratic Programming problem. Hence, now we apply KKT condition to solve this optimization problem. If the primal problem and the dual problem are almost symmetric, the problem will satisfy the following KKT conditions:

$$\begin{cases} 1. \quad \lambda_i = 0 \Rightarrow y_i f(x_i) \geq 1 \\ 2. \quad 0 < \lambda_i < 0 \Rightarrow y_i f(x_i) = 1 \\ 3. \quad \lambda_i = C \Rightarrow y_i f(x_i) \leq 1 \end{cases} \tag{41}$$

As a result, the samples outside the boundary satisfy the condition 1, samples exactly on the boundary satisfy the condition 2, and samples within the boundary satisfy the condition 3.

And the bias $b$ can be computed based on the following equation:

$$b = 1 - \frac{1}{|i : 0 < \lambda_i < C| \sum_{i:0<\lambda_i<C} \sum_{j=1}^{s} \lambda_j y_j x_j^T x_i} \tag{42}$$

By applying the KKT conditions, the training process of SVM can be faster. Additionally, Sequential Minimal Optimization (SMO) can be applied to solve the Quadratic Programming (QP) problem that arises during the training process of SVM. Then, for each time during the training process, two samples violating the KKT conditions will be chosen through SMO and the weights $\lambda$ will be updated until all data points satisfied KKT conditions. This algorithm will be implemented in this coursework via LIBSVM and Scikit-learn libraries (Bishop, 2006).
The above SVM derivations are for linear SVM. However, the requirement in this coursework requires to implement the kernel SVM based on the polynomial kernel and Radial Basis Function (RBF) kernel. Thus, we use the following two equations to apply kernel trick of SVM:

$$Polynomial : k(x_1, x_2) = (x_1^T x_2 + 1)^d \quad \& \quad RBF : k(x_1, x_2) = exp(-||x_1 - x_2||^2 / 2\gamma^2) \tag{43}$$

where $d$ is degree which controls the flexibility of the decision boundary by considering in $d$-dimension, and $\gamma$ defines the distance of the influence by a single training example. However, from Fig. 5 and Fig. 6, we can notice that the original SVM is only designed for the binary classification. Hence, for the multi-class classification, we should apply additional methods, which are One-Vs-One or One-Vs-Rest classifications (Bishop, 2006). These two methods can be simply implemented via Scikit-learn library. Therefore, in this paper, the performance and efficiency of SVM based on the two different methods will be analysed as well.

## 3 Design and Results

In this section, the detailed steps and process of the self-designed algorithms will be explained step-by-step with the clipped running images. According to the requirements, the MLP and KNN algorithm will not use any third-party libraries except "train_test_split" function from Scikit-learn. On the other hand, kernel SVM will be implemented using Scikit-learn library. Additionally, to check the pure performance of SVM based on LIBSVM, the kernel SVM performance of LIBSVM will be measured as well although Scikit-learn is based on LIBSVM using SMO algorithm. Firstly, the algorithms designed for MLP will be introduced.

### 3.1 THE DESIGN OF MLP

**Algorithm 1** Function to implement one hot encoding: **one_hot_en()**

1: **Input:** label array - *y_train, y_val and y_test*
2: b = zero arrays scaled by the label array
3: $0 \rightarrow 1$ where the array index is equal to the labels
4: **Return** b

**Algorithm 2** Function to obtain the eigenvalues and eigenvectors of a dataset for PCA: **pca_fit()**

1: **Input:** dataset
2: means = mean of each sample in the dataset
3: data = data - means
4: cov_mat = $data.T \cdot data$
5: evals = eigenvalues of $cov\_mat$
6: evecs = eigenvectors of $cov\_mat$
7: **Return** evals, evecs

**Algorithm 3** Function to transform the dataset into n-dimensional principle components by PCA: **pca_transform()**

1: **Input:** dataset, eigenvectors, n_dim
2: result = $dataset \cdot eigenvectors[:, 0 : n\_dim]$
3: **Return** result

Algorithm 1 is designed for one hot encoding. In Algorithm 2, the input dataset will be firstly normalizing via mean centering. And then, its covariance matrix will be obtained. Through eigendecomposition of this matrix, we obtain its eigenvalues and eigenvectors. After that, Algorithm 3 will take this eigenvectors and the dataset to transform the dataset into $n\_dim$-dimensional principle components. Based on these two functions, we can implement PCA for dimensionality reduction of the given dataset. Now, the MLP model can be designed as **class MLP**. The following algorithms are the methods used in this MLP model based on the mathematical derivations provided in the previous section.

**Algorithm 4** Initial parameters: **__init__(self, learning_rate, hidden_node, batch_size)**

1: w1, b1, w2, b2 = None
2: train_losses, val_losses = []
3: train_accuracy, val_accuracy = []
4: lr = learning_rate
5: node = hidden_node
6: batch_size = batch_size

**Algorithm 5** Initializing weights and biases: **initial_wb(self, m, n)**

1: k = node
2: w1 = Initialize (m×k) array via normal distribution
3: b1 = $k$ scaled-zero array
4: w2 = Initialize (k×n) array via normal distribution
5: b2 = $n$ scaled-zero array

**Algorithm 6** Hyperbolic tangent activation function: **htan(self, z1)**

1: z1 = Clipped z1 between [-100, None]
2: **Return** $\frac{e^{z1}-e^{-z1}}{e^{z1}+e^{-z1}}$

**Algorithm 7** Derivative of hyperbolic tangent function: **der_htan(self, z1)**

1: **Return** $1 - htan(z1)^2$

**Algorithm 8** ReLU activation function: **ReLU(self, z1)**

1: a1 = max(0, z1)
2: **Return** a1

**Algorithm 9** Derivative of ReLU function: **der_ReLU(self, z1)**

1: **Return** $\begin{cases} 0 & if \quad z1 < 0 \\ 1 & if \quad z1 > 0 \end{cases}$

**Algorithm 10** Softmax activation function: **softmax(self, z2)**

1: a2 = $\frac{e^{z2}}{\sum_{i=1}^{n} e^{z2}}$
2: **Return** a2

**Algorithm 11** Forward propagation from input to hidden layer: **forward1(self, x)**

z1 = $x \cdot w1 + b1$
**Return** z1

**Algorithm 12** Forward propagation from hidden to output layer: **forward2(self, a1)**

z2 = $x \cdot w2 + b2$
**Return** z2

**Algorithm 13** Backpropagation algorithm: **backpropagation(self, x, y)**

z1 = **forward1(x)**
a1 = **htan(z1)** or **ReLU(z1)**
d_a1 = **der_htan(z1)** or **der_ReLU(z1)**
z2 = **forward2(a1)**
a2 = **softmax(z2)**
der_w2 = $\frac{\partial z_2}{\partial w_2}\frac{\partial a_2}{\partial z_2}\frac{\partial L}{\partial a_2} = a_1^T(-(y - a_2))$
der_b2 = $\frac{\partial z_2}{\partial b_2}\frac{\partial a_2}{\partial z_2}\frac{\partial L}{\partial a_2} = 1^T(-(y - a_2))$
hidden_error = $\frac{\partial a_1}{\partial z_1}\frac{\partial z_2}{\partial a_1}\frac{\partial a_2}{\partial z_2}\frac{\partial L}{\partial a_2}$
$= -(y - a_2)w_2^T \cdot a_1 \cdot d\_a1$
der_w1 = $\frac{\partial z_1}{\partial w_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_2}{\partial a_1}\frac{\partial a_2}{\partial z_2}\frac{\partial L}{\partial a_2}$
$= x^T \cdot hidden\_error$
der_b1 = $\frac{\partial z_1}{\partial b_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_2}{\partial a_1}\frac{\partial a_2}{\partial z_2}\frac{\partial L}{\partial a_2}$
$= 1^T \cdot hidden\_error$
**Return** der_w1, der_b1, der_w2, der_b2

**Algorithm 14** Mini-batch implementation: **mini_batch(self, x, y)**

iter = Ceil$\left(\frac{len(x)}{batch_size}\right)$
x = shuffled x
y = shuffled y
**for** *i* in **range**(iter) **do**
    start = $batch\_size * i$
    end = $batch\_size * (i + 1)$
    **yield** x[start:end], y[start:end]
**end for**

**Algorithm 15** Predicting the label: **predict(self, x_data)**

z1 = **forward1(x_data)**
a1 = **htan(z1)** or **ReLU(z1)**
z2 = **forward2(a1)**
**Return** Index of the maximum z2

**Algorithm 16** Accuracy calculation: **acc_score(self, x_data, y_data)**

**Return** Accuracy of the predicted labels of x_data compared to y_data

**Algorithm 17** Final MLP model based on the above methods: **model(self, x_data, y_data, epochs, x_val, y_val)**

**initial_wb(x_data, y_data)**
**for** *epoch* in **range**(epochs) **do**
    loss = 0
    **for** *x, y* in **mini_batch(x_data, y_data) do**
        loss += **cross_entropy_loss(x, y)**
        der_w1, der_b1, der_w2, der_b2 = **backpropagation(x, y)**
        w2 -= $lr \times \frac{der\_w2}{len(x))}$
        b2 -= $lr \times \frac{der\_b2}{len(x)}$
        w1 -= $lr \times \frac{der\_w1}{len(x)}$
        b1 -= $lr \times \frac{der\_b1}{len(x)}$
    **end for**
    val_loss = **val_loss(x_val, y_val)**
    Append $\frac{loss}{len(y\_data)} \rightarrow$ **train_losses**
    Append $val\_losses \rightarrow$ **val_losses**
    Append **acc_score(x_data, y_data)** $\rightarrow$ **train_accuracy**
    Append **acc_score(x_val, y_val)** $\rightarrow$ **val_accuracy**
**end for**

From Algorithm 4 to Algorithm 17, the class of MLP model is designed. Here, **mini_batch()** method is designed to implement Mini-Batch Gradient Descent during the training process. It is a mixture of Batch Gradient and Stochastic Gradient Descent (SGD). In this method, we create a batch of a fixed number of training samples that is less than the actual training dataset. This is called a mini-batch. Then, in every epoch, the model will pick one mini-batch for the neural network. After that, the model will calculate the mean gradient of the mini-batch and use it to update the weights. Since we train less dataset, the training process of this model can be efficient while maintaining the accuracy (Patrikar, 2019).
Then, finally, we can implement the MLP model as follows:

**Algorithm 18** Main part of the MLP code:

```
x_train, y_train = mnist_reader.load_minist('train')
x_test, y_test = mnist_reader.load_minist('t10k')
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, 20%)
x_train, x_val, x_test = 28 × 28 → 1 × 784 array
x_train, x_val, x_test = Converted values from the range (0, 255) to (0, 1)
y_train, y_val, y_test = one_hot_en(y_train), one_hot_en(y_val), one_hot_en(y_test)
eig_vec = pca_fit(x_train)[1]
n_dim = 120
x_train = pca_transform(x_train, eig_vec, n_dim)
x_val = pca_transform(x_val, eig_vec, n_dim)
x_test = pca_transform(x_test, eig_vec, n_dim)
model = MLP(learning_rate = #, batch_size = #, hidden_node = #)
model.model(x_train, y_train, epochs= #, x_val = x_val, y_val = y_val)
Plot the result
```

In Algorithm 18, **mnist_reader()** is a code provided by (Xiao et al., 2017) to load the MNIST and Fashion-MNIST datasets. After loading Fashion-MNIST dataset, the training dataset will be split into train set and validation set via Scikit-learn library. Since the datasets are in $28 \times 28$ scaled 2-dimensional array, it will be converted into $1 \times 784$ scaled 1-dimensional array. Note that **mnist_reader()** already processes the dataset into one-dimensional array. Thus, these codes above could be deleted if we use **mnist_reader()**.
And then, by dividing the dataset by 255, we can normalize the input values. Lastly, the labels will be converted into $1 \times 10$ array through one-hot encoding. Then, the data-preprocessing is completed.
From this process, we can notice that the dimension of the dataset is significantly high. Hence, the dimensionality reduction could be applied to simplify the relationship of each data points and improve both accuracy and efficiency. In this coursework, I applied the Principal Component Analysis (PCA) via Algorithm 2 and Algorithm 3. For the Fashion-MNIST dataset, the defined number of principal components is 120 because the experiment result provided in (Bilogur, 2018) shows that 120 principal components performs a good PCA reconstruction result with low errors.
By using these datasets, we can implement the class **MLP()**. The following figures show the process of data-preprocessing.

```
length of x_train: (48000, 784)
length of x_val: (12000, 784)
length of x_test: (10000, 784)
```

Figure 7: Fashion-Mnist dataset.

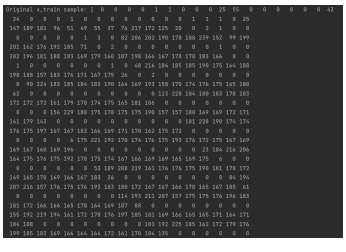

Figure 8: Sample of the training dataset.

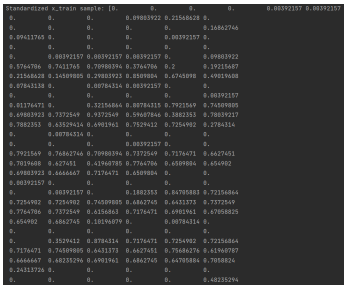

Figure 9: Normalized training dataset sample.

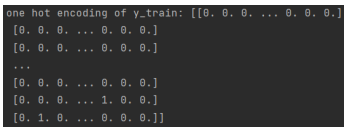


Figure 10: One hot encoding result of y_train.

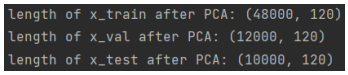


Figure 11: PCA dimensionality reduction.

After the data-preprocessing, the MLP model can be implemented. Generally, the bigger batch size requires bigger learning rate. The appropriate values can be determined based on the experimental result. In this coursework, the default values of the learning rate, batch size and hidden nodes are 0.1, 64 and 150, respectively. Table 1 shows that the performance of the default hyperparameter set is the best among 9 cases. About the number of hidden nodes, there is no theoretical formula to calculate it. However, generally, the number of hidden nodes is between [# of input values - $\frac{2*\text{\# of input values}}{3}$, # of input values + $\frac{2*\text{\# of input values}}{3}$] (Bishop, 2006). Lastly, the number of epochs is defined as 130 after several tests. If the epoch is too less, it will cause underfitting. On the other hand, the number of epoch higher than 130 could cause overfitting. Based on these hyperparameters, we can train the MLP model. The following figure shows the training process. Note that the final results will be provided in the next section for the analysis.

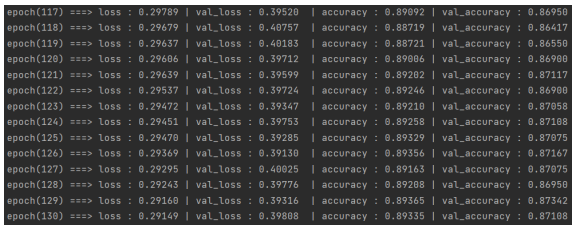


Figure 12: An example of the MLP training process.

### 3.2 The design of KNN

For the KNN model, there are total 10 different self-defined functions. However, six functions are similar to the explained functions in the previous part or the assignment 2. Hence, their codes will be provided in Appendix instead of detailed description here. Then, this section can focus on the design of KNN model. Firstly, the following algorithms are the two key functions of KNN model:

**Algorithm 19** Function to calculate the Euclidean distance: **euc_dist()**

1: **Input:** X, Y
2: distance = $\sqrt{\sum_{i=1}^{n}(X-Y)^2}$
$= \sqrt{X^2+Y^2-2XY}$ via **np.einsum()**
3: **Return** distance

**Algorithm 20** Function to implement KNN: **KNN()**

1: **Input:** tr_d, tr_l, tt_d, tt_l, k
2: distances = **euc_dist**(tt_d, tr_d)
3: min_ind = index of $k$-number of closest data points
4: best_neighbours = tr_l[$min\_ind$] $\Rightarrow$ Get the training label of $min\_ind$
5: pred = highest $K_i$ in $best\_neighbours$
6: accuracy = **score**(pred, tt_l)
7: **Return** pred, accuracy

As explained before, we apply the Euclidean distance to calculate distance as Algorithm 19. Since the computation cost of square root is significant in numpy, the equation is decomposed based on the trick of algebra. Also, using the Einstein summation convention of numpy can decrease the memory usage and computation time (Daniel, 2018).
Based on this function, Algorithm 20 will find the $k$-nearest neighbours and predict the class of data points based on equation 17. Then, by comparing the predicted and actual labels, we can compute the classification accuracy.

**Algorithm 21** Function to implement 5-fold cross-validation: **cross_val()**

```
Input: data, label, k
acc_list = []
for i in range(len(data)) do
    com_data, com_label = split train. & val.
    train_data, train_label = com_data[0], com_label[0]
    val_data, val_label = com_data[1], com_label[1]
    result =
      KNN(train_data, train_label, val_data, val_label, k)
    acc = result[1] & acc_list.append(acc)
    avg_acc = mean of acc_list
end for
Return avg_acc, acc_list
```

**Algorithm 22** Function to find the optimal value of k: **optimal_k()**

```
Input: data, label, re
mix_trd, mix_trl = shuffle(data), shuffle(label)
split_data, split_label = split mix_trd & mix_trl into 5 sets
accuracy = [] & ind_acc = []
for k in range(1, re, 1) do
    cv_result = cross_val(split_data, split_label, k)
    k_acc = cv_result[0]
    Append cv_result[1] → ind_acc[]
    Append k_acc → accuracy[]
end for
opt_k_index = index of max(accuracy)
opt_k = opt_k_index + 1
Return opt_k, accuracy, ind_acc
```

Here, 5-fold cross-validation is applied to find the optimal $k$ value because we cannot use the test dataset for the optimization (i.e., test dataset should be unknown until the final test). Now, based on the designed functions, we can implement KNN with finding the optimal hyper-parameter. Note that the data preprocessing will be based on Algorithm 18. In Algorithm 23, $re$ is a parameter to define the range of $k$ during

**Algorithm 23** Main part of the KNN code:

```
Data-preprocessing with reshaping, normalization and PCA(n_dim = 120)
re = 21
optimal_result = optimal_k(x_train, y_train, re)
k = optimal_result[0]
k_effect = optimal_result[1]
final_result = KNN(x_train, y_train, x_test, y_test, k)
final_acc = final_result[1]
Plot the result
```

the optimization process. After finding the optimal $k$, the final **KNN** model will classify the test dataset. The following figures show the general process of KNN:

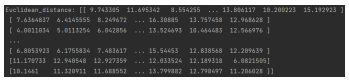


Figure 13: Calculation result of Euclidean distance.

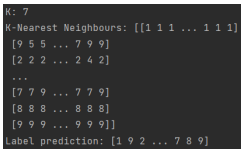


Figure 14: Computation of the K-Nearest Neighbours.

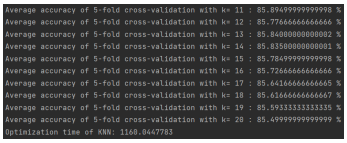


Figure 15: Optimization process of KNN.

After 1,160s, we can obtain the optimal $k$ based on the optimization process. Note that the final result of KNN will be provided and analysed in the next section.

### 3.3 The design of SVM

In this coursework, we are allowed to use the Scikit-learn library to implement the kernel SVM model. Therefore, this design section will focus on how to implement the kernel SVM model and find the optimal hyperparameters. The detailed explanation of the SVM working principal and mathematical derivations is already provided in the previous section.

For the SVM model, the self-defined functions designed for data-preprocessing and cross-validation are almost similar to the KNN model, but SVM applies 3-fold cross-validation to decrease the optimization time. In this section, three key self-defined functions will be explained as follows:

**Algorithm 24** Function to find a group of best hyper-parameter sets: **optimal_params()**

**Input:** name, data, label, C, para
mix_trd, mix_trl = **shuffle**(data), **shuffle**(label)
split_data, split_label = split mix_trd & mix_trl into 3 sets
accuracy = [], param_set = [], avg_coord = []
**for** $c$ in C **do**
    **for** $p$ in para **do**
        mean_acc = **cross_val**(name, split_data, split_label, c, p)
        Append mean_acc → accuracy[]
        set_param = (c, p)
        Append set_param → param_set[]
        Append [c, p, mean_acc] → avg_coord[]
    **end for**
**end for**
opt_p_index = index of max(accuracy)
opt_hp = param_set[opt_p_index]
**Return** opt_hp, avg_coord

**Algorithm 25** Function to find the best hyper-parameter set: **deep_optimal_params()**

**Input:** name, data, label, C, para
mix_trd, mix_trl = **shuffle**(data), **shuffle**(label)
split_data, split_label = split mix_trd & mix_trl into 3 sets
accuracy = [], param_set = [], avg_coord = []
**for** $id, c$ in **enumerate**(C) **do**
    p = para[id]
    mean_acc = **cross_val**(name, split_data, split_label, c, p)
    Append mean_acc → accuracy[]
    set_param = (c, p)
    Append set_param → param_set[]
    Append [c, p, mean_acc] → avg_coord[]
**end for**
opt_p_index = index of max(accuracy)
opt_hp = param_set[opt_p_index]
**Return** opt_hp, avg_coord

**Algorithm 26** Function to implement kernel SVM: **kernel_SVM()**

**Input:** name, data, label, C, param
**if** name == 'poly' **then**
    d = param
    clf = svm.SVC(kernel='poly', C=c, degree=d)
    clf = OvO-Classifier(clf) or OvR-Classifier(clf)
    clf.fit(data, label)
    **Return** clf
**else if** name == 'rbf' **then**
    g = param
    clf = svm.SVC(kernel='rbf', C=c, gamma=g)
    clf = OvO-Classifier(clf) or OvR-Classifier(clf)
    clf.fit(data, label)
    **Return** clf
**else**
    Error
**end if**

**Algorithm 27** Main part of the SVM code

Data-preprocessing as MLP & KNN
C = [0.1,1,10,100,1000]
Gamma = [0.01, 0.1, 1, 10, 100]
Degree = [2,3,4,5,6]
x_train, x_test, y_train = Reshaping (len(dataset), n_dim)
xtr, x_val, ytr, y_val = **train_test_split**(x_train, y_train, 20%)
name = 'poly' or 'rbf'
optimization = **optimal_params**(name, x_val, y_val, C, Degree or Gamma)
coord_1 = np.array(optimization[1])
avg_coord = tranpose(coord_1)
stand_opt_id = four best hyper-parameter sets
stand_c = avg_coord[0][stand_opt_id]
stand_k = avg_coord[1][stand_opt_id]
deep_optimization = **deep_optimal_params**(name, x_train, y_train, stand_c, stand_k)
hyperparameters_set = deep_optimization[0]
opt_C, opt_K = hyperparameters_set[0] & [1]
clf = **kernel_SVM**(name, x_train, y_train, opt_C, opt_K)
y_pred = clf.predict(x_test)
result = clf.score(x_test, y_test)
Plot the result

From Algorithm 27, we can firstly use 12,000 validation samples to find the four best hyperparameter sets among 25 sets via the standard optimization process with 3-fold cross-validation. And then, the deep optimization will find the best cost and kernel parameters for the total training dataset (Khandelwal, 2018). Finally, we can train SVM based on this best hyperparameter set and predict the test dataset.

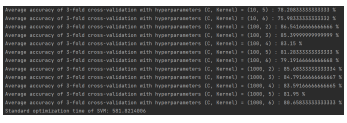

Figure 16: Example of standard SVM optimization.

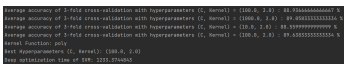

Figure 17: Example of deep SVM optimization.

Figure 18: SVM training process in a '.sav' file.

# 4 ANALYSIS

## 4.1 ANALYSIS OF THE MLP RESULT

To implement a good MLP model, we need to determine the hyper-parameters based on the experimental result. Hence, the following table shows the test results of nine different parameter sets:

Table 1: The experiment result of the MLP hyper-parameters selection

| **Using ReLU / (lr, batch size)** | **(0.05, 32)** | **(0.1, 64)** | **(0.15, 128)** |
|---|---|---|---|
| ***Accuracy (hidden node = 80)*** | 85.89 % | 85.92 % | 85.22 % |
| ***Accuracy (hidden node = 150)*** | 86.15 % | 86.34 % | 85.51 % |
| ***Accuracy (hidden node = 180)*** | 85.97 % | 86.17 % | 85.56 % |

Since the highest accuracy appears when the learning rate, batch size and number of hidden nodes are equal to 0.1, 64 and 150, the default parameter set in the previous section is appropriate.
Now, the hyper-parameters of the designed MLP model are selected. Therefore, we can lastly check and compare the MLP performance of two different activation functions in the hidden layer (i.e., Tanh and ReLU). The activation function of the output layer is set to Softmax.
Then, the final results of the hyperbolic tangent and ReLU activation function are measured as follows:

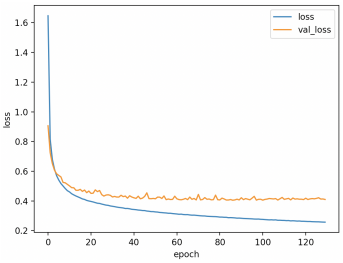


Figure 19: Loss graph of the MLP training with Tanh.

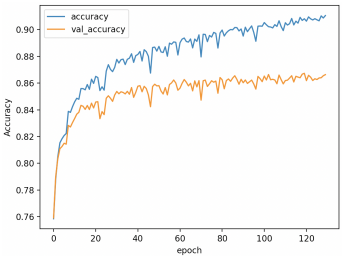


Figure 20: Accuracy graph of the MLP training with Tanh.

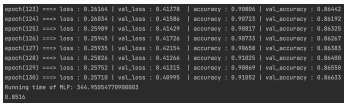

Figure 21: Final result of MLP with Tanh.

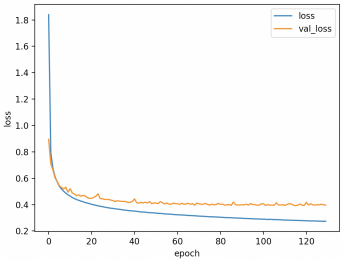


Figure 22: Loss graph of the MLP training with ReLU.

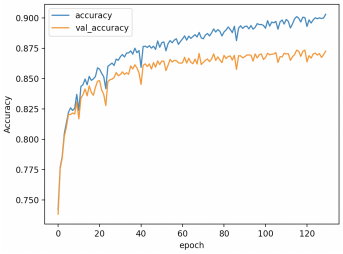


Figure 23: Accuracy graph of the MLP training with ReLU.

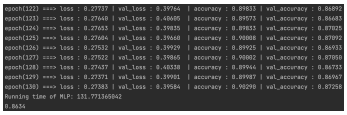

Figure 24: Final result of MLP with ReLU.

From the above figures, we know that both two activation functions can perform well with high accuracy. Fig. 21 shows that the training loss and validation loss are converged around 0.257 and 0.409, respectively. Also, the training accuracy and validation accuracy are about 90.8% and 86.5% after the training process. Finally, the classification accuracy of the test dataset is obtained as 85.16%.
Next, we can see Fig. 24 to check the performance of ReLU activation function. After training, the training

and validation loss are converged to about 0.27 and 0.39, respectively. The training and validation accuracies are approximately 89.9% and 87.2%, respectively. Based on the training result, the test accuracy of ReLU function is 86.34%.
From those two figures, we can also notice that the running time of ReLU is shorter than Tanh. It is because ReLU requires significantly low computational cost compared to the hyperbolic tangent function. Additionally, the results show that the accuracy of ReLU is higher than Tanh. For the most cases, this result is correct because ReLU activation function generally performs better than both sigmoid and hyperbolic tangent functions although there is no exact theoretical proof.
Lastly, the figures below are the ReLU based-MLP classification result without the dimensionality reduction (PCA):

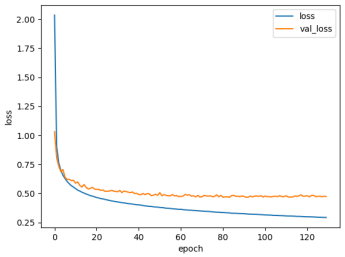


Figure 25: Loss graph of the MLP training with ReLU (not using PCA).

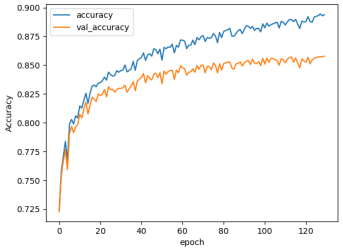


Figure 26: Accuracy graph of the MLP training with ReLU (not using PCA).

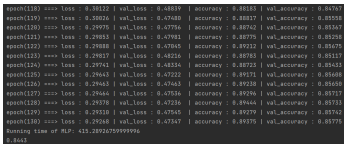

Figure 27: Final result of MLP with ReLU (not using PCA).

Fig. 25 and Fig. 26 show that the MLP model can still perform well without the dimensionality reduction. However, the training performance is worse than the MLP classification with PCA. By comparing Fig. 24 and Fig. 27, it is clear that the accuracy of MLP without PCA is almost 2% lower than MLP with PCA. Also, the running time increases dramatically if we do not apply PCA. The running time of ReLU-based MLP without PCA is even higher than MLP using Tanh in Fig. 21. Hence, although the test accuracy in Fig. 27 could be slightly increased if we increase the number of epochs, the dimensionality reduction (PCA) can improve both accuracy and efficiency of the classification. Note that the loss and accuracy graphs are fluctuated because of Mini-batch, but it will not influence the performance of MLP considerably.

### 4.2 Analysis of the KNN result

In the design part, the clipped running images of the computation and optimization processes of KNN are provided. The following figure is the accuracy graph of the optimization process with 5-fold cross-validation to find the optimal hyper-parameter $k$:

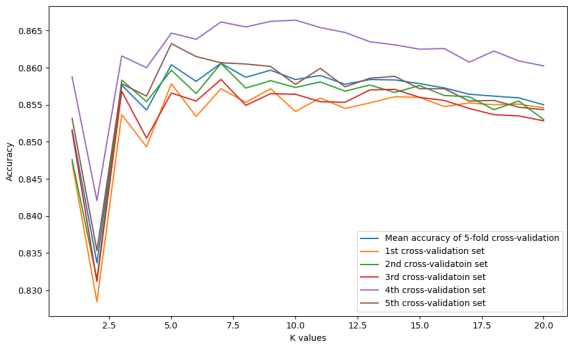


Figure 28: The 5-fold cross-validation result of KNN to find the optimal $k$.

Note that all the values of $k$ are set to natural number from 1 to 20 for the optimization process. Then, the designed optimization algorithm of KNN will calculate the average accuracy of 5-fold cross-validation and select the hyper-parameter when the average accuracy is the highest. As a result, the final output of KNN is given as follows:

```
Optimal value of K: 7
Final test accuracy with the optimal K: 86.04 %
```

Figure 29: Final classification result of KNN with the optimal $k$.

As per the optimization result, the optimal value of $k$ is 7. Hence, the final KNN classification accuracy based on the optimal hyper-parameter is equal to 86.04%. Compared to the best accuracy of MLP, it is slightly lower, but the KNN accuracy is still good. Additionally, the following test is conducted to compare the running time of KNN based on the self-designed model and Scikit-learn library which is based on the K-D trees:

```
Accuracy of the self-designed KNN: 86.04 %
Final running time of the self-designed KNN: 8.2275455
Accuracy of the Scikit-library KNN: 86.41 %
Final running time of the Scikit-library KNN: 9.722062334
```

Figure 30: Comparison of the self-designed KNN and Scikit-learn KNN running times.

Fig. 30 shows that the self-designed KNN can compute the classification faster than Scikit-learn library. Therefore, it proves the designed KNN model in this coursework can classify the large dataset efficiently based on Algorithm 19. Although its accuracy is slightly lower than Scikit-learn, the self-designed KNN also has good classification accuracy, which is above 86%. Thus, it can be stated that the KNN design introduced in this report can perform well with both high efficiency and accuracy.

### 4.3 Analysis of the SVM result

In the coursework description, there is a provided SVM code. However, the designed SVM code in this report only refers to the data-reshaping code to input the Fashion-MNIST dataset. Other parts are self-designed to implement kernel SVM with Scikit-learn library. To apply SVM for multi-class classification, we firstly need to determine which method is more appropriate between OvO and OvR classifiers. The default setting of Scikit-learn SVM is based on OvO classifier, but we can compare the performance of two methods based on the following figures:

```
Kernel Function: poly
SVM Model: OneVsOneClassifier(estimator=SVC(C=100, degree=2, kernel='poly'))
Training time of SVM: 89.456159
Testing time of SVM: 70.1766076
Test accuracy: 89.51 %
Finished!
```

Figure 31: Polynomial kernel SVM performance with OvO classifier.

```
Kernel Function: poly
SVM Model: OneVsRestClassifier(estimator=SVC(C=100, degree=2, kernel='poly'))
Training time of SVM: 1374.4516836
Testing time of SVM: 51.15301039999986
Test accuracy: 89.27000000000001 %
Finished!
```

Figure 32: Polynomial kernel SVM performance with OvR classifier.

Here, the cost and kernel parameters are chosen arbitrary. By comparing the above results, it is clear that the total running time of OneVsOne classifier is considerably shorter than OneVsRest classifier, and the accuracy is better as well. RBF kernel will also have similar result, but the difference will increase dramatically because RBF kernel requires significant computational cost if the dataset is too large. Thus, as the default model, we use OneVsOneClassifier in this report.

After deciding the multi-class classification method of SVM, I compared the kernel SVM performance of using LIBSVM directly and through Scikit-learn library. Note that both methods apply the SMO algorithm because LIBSVM is based on SMO, and Scikit-learn library uses LIBSVM for the SVM classification.

```
.....................................................
optimization finished, #iter = 498781
nu = 0.170228
obj = -161928.957969, rho = -0.943613
nSV = 2672, nBSV = 1575
.............................*.................*
optimization finished, #iter = 46025
nu = 0.020959
obj = -14263.040116, rho = -0.601605
nSV = 673, nBSV = 85
............*......*
optimization finished, #iter = 19192
nu = 0.009569
obj = -6295.477017, rho = -0.071235
nSV = 364, nBSV = 36
Total nSV = 15441
```

Figure 33: Sequential Minimal Optimization process of LIBSVM.

```
Training time of SVM: 255.39189869999998
Accuracy = 89.54% (8954/10000) (classification)
Testing time of SVM: 54.36328689999999
Finished!
```

Figure 34: Classification result of the polynomial kernel SVM via LIBSVM.

Fig. 33 shows some examples of the SMO process using LIBSVM. From Fig. 34, we know that the total running time of the polynomial kernel SVM with the direct use of LIBSVM is longer than Scikit-learn. On the other hand, the accuracy in Fig. 34 is slightly higher than Scikit-learn. By comparing the efficiency and accuracy of both results, the final SVM model is determined to use Scikit-learn library because its efficiency is more benefit than the slightly better accuracy of LIBSVM (i.e., trade-off between efficiency and accuracy) (Chang & Lin, 2011).
Since we finished the detailed design of the final kernel SVM model, we implement this model for both polynomial and RBF kernels. Firstly, for the polynomial kernel, the following figures can be obtained:

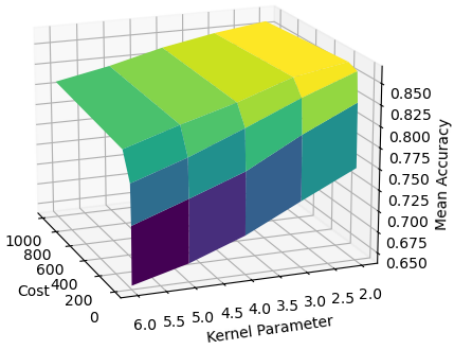


Figure 35: Standard optimization result of the polynomial kernel SVM.

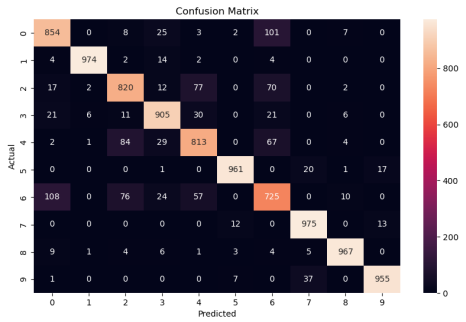


Figure 36: Confusion matrix of the polynomial kernel SVM.

Fig. 35 is the standard optimization result of polynomial kernel SVM. Among 25 different sets, this algorithm will find four best parameter sets with the 12,000 validation samples. And then, as shown in Fig. 17, the best hyper-parameter set will be chosen based on the total training dataset. For the polynomial kernel, the best hyper-parameter set is (Cost = 100, Degree = 2).
Fig. 36 illustrates the confusion matrix of the polynomial kernel SVM based on the best hyper-parameter set. Then, we can see that the test dataset is well classified compared to its actual labels. From Fig. 37, the test accuracy of this model is equal to 89.49%.

```
Training output with optimal hyperparameters: OneVsOneClassifier(estimator=SVC(C=100.0, degree=2.0, kernel='poly'))
Training time of SVM: 82.43959519999999
Testing time of SVM: 62.807543099999975
Accuracy:  89.49000000000001 %
Finished!
```

Figure 37: The final result of Polynomial kernel SVM.

Now, we implement the RBF kernel SVM as follows:

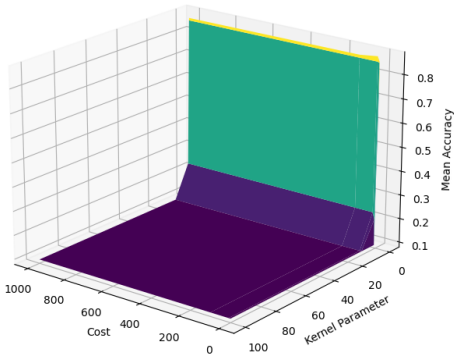


Figure 38: Standard optimization result of the RBF kernel SVM.

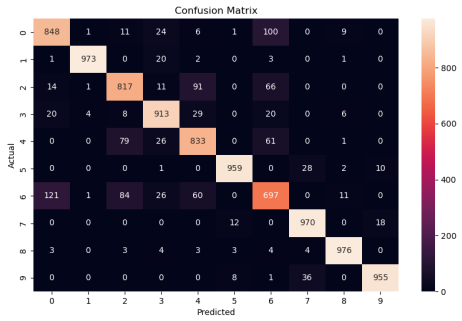


Figure 39: Confusion matrix of the RBF kernel SVM.

As per the same process, the best hyper-parameter set of the RBF kernel model is (Cost = 10, Gamma = 0.01). Then, we can get a similar confusion matrix to the polynomial kernel.

```
Training output with optimal hyperparameters: OneVsOneClassifier(estimator=SVC(C=10.0, gamma=0.01))
Training time of SVM: 64.95771859999877
Testing time of SVM: 202.92568310000024
Accuracy:  89.41 %
Finished!
```

Figure 40: The final result of RBF kernel SVM.

The final classification accuracy of the RBF kernel SVM is around 89.41%. Theoretically, the accuracy of RBF kernel should be higher than polynomial because RBF kernel is so called one of the most powerful kernel trick. Thus, we can notice that the chosen parameter of RBF kernel is not the best value. In this case, we can simply redefine the range of hyper-parameter list in the code. From Fig. 35 and Fig. 38, we can observe that the effect of cost values on the prediction accuracy is not as significant as the kernel parameter. Hence, by keeping the current cost range, we simply modify the Gamma range to improve the RBF kernel SVM. In general, the decision boundary will be more flexible if we increase the hyper-parameters because the bias decreases and variance increases (Bishop, 2006).
However, the following figure shows that the RBF kernel took around 9,318 seconds to find the optimal hyper-parameters. Since the RBF kernel SVM with the large scale of Fashion-MNIST dataset causes poor efficiency, thus, it is more reasonable to use the polynomial kernel because of its better efficiency and good accuracy that is still above 89%.

```
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1000, 0.1) : 85.44166666666668 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1000, 1) : 23.191666666666666 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1000, 10) : 9.45 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1000, 100) : 9.45 %
Standard optimization time of SVM: 6801.1400242
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1.0, 0.01) : 88.27833333333332 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (1000.0, 0.01) : 89.29666666666667 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (100.0, 0.01) : 89.93333333333334 %
Average accuracy of 3-fold cross-validation with hyperparameters (C, Kernel) = (10.0, 0.01) : 90.06 %
Kernel Function: rbf
Best Hyperparameters (C, Kernel): (10.0, 0.01)
Deep optimization time of SVM: 2517.6104405000005
```

Figure 41: The optimization result of RBF kernel SVM.

### 4.4 Comparison

Based on the results of three classification methods provided in this section, now we can compare the advantages and disadvantages of these algorithms. Firstly, MLP can perform non-linear classification well

with good efficiency. Also, we can simply add more hidden layers to improve its performance. However, the design of MLP model should be careful because the hyper-parameters can cause many issues such as slow convergence, over-fitting, under-fitting, local minimum, vanishing gradient (for sigmoid and tanh) and dying ReLU. However, due to the advantages of its simple design structure, we can solve these problems by modifying the MLP structure or changing activation functions.
About KNN, the core advantage is its simple implementation. Since it does not require any training process, we can simply find the optimal $k$ and check $k$-nearest neighbours. However, KNN requires to calculate the distance of every training data points. Thus, to reduce its computation complexity and memory usage, we can apply some speeding-up solutions. In Scikit-library, KNN has high efficiency by using K-D trees algorithm. However, in this coursework, I applied the algebraic decomposition method of Euclidean distance to reduce the memory usage significantly, avoiding the square root computation. And then, using Numpy Einstein summation convention, the distance will be calculated without any For loop because the For loop in python has a poor computational speed.
Finally, the detailed experimental results of SVM shows both advantages and disadvantages of this method. Compared to other two methods, the SVM with both two kernels achieved the highest classification accuracy. Also, the kernel trick allows SVM to classify non-linear data accurately with good stability. However, as shown in Section 2, SVM is difficult to design due to its mathematical complexity. Additionally, SVM requires long training time and high memory usage although we apply an optimization method called SMO. Among these issues, the most critical problem is that it is really difficult to find an appropriate hyper-parameter set of kernel SVM because we need to find the best parameter set in the infinite range. Therefore, its computational complexity and uncertainty will decrease the performance and efficiency of SVM (Bishop, 2006).

## 5 Conclusion

This report explains the basic knowledge and mathematical derivations of three different classification algorithms (i.e., 3-layer Perceptron, KNN and SVM). After that, the detailed explanations and design descriptions of the three algorithms are provided in the design and result section. Based on the designed algorithms, the classification results are obtained as shown in Section 4 with thorough analysis. Comparing the performance of these three different classification methods, SVM has the highest accuracy, but its efficiency is poor. On the other hand, KNN can obtain the classification result in less than 15 seconds, but its accuracy is lower than other two methods. As a result, the 3-layer Perceptron model could be recommended in this task because it can achieve high accuracy with good efficiency.

## References

A. Bhardwaj. What is a perceptron?-basics of neural networks, 2020. Available from: https://towardsdatascience.com/what-is-a-perceptron-basics-of-neural-networks-c4cfea20c590 [accessed 27 May 2022].

A. Bilogur. Dimensionality reduction and pca for fashion mnist, 2018. Available from: https://www.kaggle.com/code/residentmario/dimensionality-reduction-and-pca-for-fashion-mnist/notebook. [accessed 4 June 2022].

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Cambridge, 2006.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

A. Christopher. K-nearest neighbor, 2021. Available from: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4. [accessed 28 May 2022].

F. Daniel. How to do n-d distance and nearest neighbor calculations on numpy arrays, 2018. Available from: https://stackoverflow.com/questions/52366421/how-to-do-n-

d-distance-and-nearest-neighbor-calculations-on-numpy-arrays. [accessed 4 June 2022].

R. Gandhi. Support vector machine-introduction to machine learning algorithms, 2018. Available from: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47. [accessed 27 May 2022].

L. Ganji. One hot encoding to treat categorical data parameters, 2022. Available from: https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/. [accessed 27 May 2022].

P. Khandelwal. Mnist svm, 2018. Available from: https://github.com/prashantkh19/MNIST_SVM. [accessed 5 June 2022].

K. E. Koech. Cross-entropy loss function, 2020. Available from: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e. [accessed 27 May 2022].

T. Matsuzaki. Mathematical introduction for svm and kernel functions, 2020. Available from: https://tsmatz.wordpress.com/2020/06/01/svm-and-kernel-functions-mathematics/. [accessed 30 May 2022].

S. Patrikar. Batch, mini batch and stochastic gradient descent, 2019. Available from: https://tsmatz.wordpress.com/2020/06/01/svm-and-kernel-functions-mathematics/. [accessed 3 June 2022].

S. Sharma. K-nearest neighbor: The distance-based machine learning algorithm, 2021. Available from: https://www.analyticsvidhya.com/blog/2021/05/knn-the-distance-based-machine-learning-algorithm/#:~:text=Data%20Science%20Blogathon.-,Introduction,classification%20and%20regression%20problem%20statements. [accessed 27 May 2022].

Simplilearn. An overview on multilayer perceptron (mlp), 2022. Available from: https://www.simplilearn.com/tutorials/deep-learning-tutorial/multilayer-perceptron [accessed 27 May 2022].

M. Waseem. How to implement classification in machine learning?, 2022. Available from: https://www.edureka.co/blog/classification-in-machine-learning/ [accessed 26 May 2022].

H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

# Appendices

## A CLASS MLP() IN MLP.PY

```
class MLP: # Self-designed 3-layer Perceptron model
  def __init__(self, learning_rate=0.1, batch_size=64, hidden_node=150,):
                                    # Initial parameters (default values)
    self.w1 = None  # Set the weights of hidden layer
    self.b1 = None  # Set the bias of hidden layer
    self.w2 = None  # Set the weights of output layer
    self.b2 = None  # Set the bias of output layer
    self.train_losses = []  # Loss of the training dataset
    self.val_losses = []  # Loss of the validation dataset
    self.train_accuracy = []  # Accuracy of the training dataset
    self.val_accuracy = []  # Accuracy of the validation dataset
    self.lr = learning_rate  # Define the Learning rate
    self.node = hidden_node  # Define the number of hidden nodes
    self.batch_size = batch_size # Define the batch size

  def initial_wb(self, m, n):  # Initialization method of weights and
                                    biases
    k = self.node # Number of hidden nodes
    self.w1 = np.random.normal(0, 1, (m, k))  # Initialize (m x k) array
                                    via Gaussian distribution
    self.b1 = np.zeros(k)  # Initialize to 0
    self.w2 = np.random.normal(0, 1, (k, n))  # Initialize (k x n) array
                                    via Gaussian distribution
    self.b2 = np.zeros(n)  # Initialize to 0

  def htan(self, z1): # Hyperbolic tangent activation function
    z1 = np.clip(z1, -100, None) # To avoid NaN
    return (np.exp(z1) - np.exp(-z1)) / (np.exp(z1) + np.exp(-z1))

  def der_htan(self, z1): # Derivative of Tanh (htan)
    return 1 - self.htan(z1) * self.htan(z1)

  def ReLU(self, z1): # ReLU activation function
    a1 = np.maximum(0, z1)
    return a1

  def der_ReLU(self, z1): # Derivative of ReLU
    return (z1 > 0).astype(int)

  def softmax(self, z2):  # Softmax activation function
    # z2 = np.clip(z2, -700, 700) # To avoid overflow
    exp_z2 = np.exp(z2)
    a2 = exp_z2 / np.sum(exp_z2, axis=1).reshape(-1, 1)
    return a2

  def forward1(self, x):  # Forward propagation from input layer to hidden
                                    layer
    z1 = np.dot(x, self.w1) + self.b1
    return z1

  def forward2(self, a1):  # Forward propagation from hidden layer to
                                    output layer
    z2 = np.dot(a1, self.w2) + self.b2
    return z2
```

```
def cross_entropy_loss(self, x, y):  # Method to calculate the cross-
                                     entropy loss
  z1 = self.forward1(x)
  a1 = self.ReLU(z1)
  z2 = self.forward2(a1)
  a2 = self.softmax(z2)

  a2 = np.clip(a2, 1e-10, 1 - 1e-10) # To avoid the zero argument of Log
  return -np.sum(y * np.log(a2))  # Cross-Entropy equation

def val_loss(self, x_val, y_val):  # Method to calculate the validation
                                   loss
  val_loss = self.cross_entropy_loss(x_val, y_val)
  return val_loss / len(y_val) # Divide by the number of datasets

def backpropagation(self, x, y):  # Backpropagation algorithm
  z1 = self.forward1(x) # Forward propagation from input layer to hidden
                          layer
  a1 = self.ReLU(z1) # Hidden layer activation function
  d_a1 = self.der_ReLU(z1)
  z2 = self.forward2(a1) # Forward propagation from the hidden layer to
                           output layer
  a2 = self.softmax(z2) # Output layer activation function

  der_w2 = np.dot(a1.T, -(y - a2))  # Calculate the gradient of hidden
                                    layer weights
  der_b2 = np.sum(-(y - a2))  # Calculate the gradient of hidden layer
                              biases

  hidden_error = np.dot(-(y - a2), self.w2.T) * d_a1 # Calculate the
                                    errors propagated to the hidden layer

  der_w1 = np.dot(x.T, hidden_error)  # Calculate the gradient of the
                                      input layer weights
  der_b1 = np.sum(hidden_error, axis=0)  # Calculate the gradient of the
                                         input layer biases

  return der_w1, der_b1, der_w2, der_b2

def mini_batch(self, x, y): # Method to implement the Mini-Batch
  iter = math.ceil(len(x) / self.batch_size) # Number of iterations

  randomize = np.arange(len(y)) # Shuffle the dataset
  np.random.shuffle(randomize)
  x = x[randomize]
  y = y[randomize]

  for i in range(iter):
    start = self.batch_size * i # Define the Mini-Batch size as per '
                                  batch_size'
    end = self.batch_size * (i + 1)
    yield x[start:end], y[start:end] # Yield the Mini-batch sequence

def predict(self, x_data): # Method to return the maximum prediction
                           value after training
  z1 = self.forward1(x_data)
  a1 = self.ReLU(z1)
  z2 = self.forward2(a1)
  return np.argmax(z2, axis=1)  # Return the maximum index value
```

```
    def acc_score(self, x_data, y_data): # Method to calculate the
                                         prediction accuracy
      return np.mean(self.predict(x_data) == np.argmax(y_data, axis=1))

    def model(self, x_data, y_data, epochs=40, x_val=None, y_val=None): #
                                         Final model of MLP
      self.initial_wb(x_data.shape[1], y_data.shape[1])  # Initializing the
                                         weights and biases
      for epoch in range(epochs): # Iterate for the number of epochs
        loss = 0  # Initialize the loss value
        for x, y in self.mini_batch(x_data, y_data): # Apply the Mini-batch
                                         function
          loss += self.cross_entropy_loss(x, y) # Accumulate the loss in
                                         each epoch
          # Calculate the weights and biases of each layers via
                                         Backpropagation
          der_w1, der_b1, der_w2, der_b2 = self.backpropagation(x, y)

          # Update the weights and biases of the hidden layer
          self.w2 -= self.lr * (der_w2) / len(x)
          self.b2 -= self.lr * (der_b2) / len(x)

          # Update the weights and biases of the input layer
          self.w1 -= self.lr * (der_w1) / len(x)
          self.b1 -= self.lr * (der_b1) / len(x)

        # Calculate the validation loss
        val_loss = self.val_loss(x_val, y_val)

        self.train_losses.append(loss / len(y_data))
        self.val_losses.append(val_loss)
        self.train_accuracy.append(self.acc_score(x_data, y_data))
        self.val_accuracy.append(self.acc_score(x_val, y_val))

        print(f'epoch({epoch + 1}) ===> loss : {loss / len(y_data):.5f} |
                                         val_loss : {val_loss:.5f}',
          f' | accuracy : {self.acc_score(x_data, y_data):.5f} |
                                         val_accuracy : {self.acc_score(x_val,
                                         y_val):.5f}')
```

## B SELF-DEFINED FUNCTIONS IN KNN.PY

```
def shuffle(data): # Function to shuffle the dataset
  shuf_data = data
  np.random.seed(97) # Define the seed value first to shuffle the data in a
                                         same way
  np.random.shuffle(shuf_data)
  return shuf_data

def combine_data(ind, split_data):  # Method to separate a train dataset and
                                         a validation dataset
  cn = 0
  for i in range(len(split_data)): # For 5-fold cross-validation, combine
                                         four datasets for a train set
    if i != ind:
      if cn == 0:
```

```
        a = split_data[i]
      elif cn == 1:
        b = split_data[i]
      elif cn == 2:
        c = split_data[i]
      else:
        d = split_data[i]
      cn += 1
    else:
      test = split_data[i] # 'i'th dataset will be the validation dataset

  train = np.vstack((a, b, c, d)) # Combine the train datasets into one set
                                    vertically
  return train, test

def combine_label(ind, split_label):  # Method to separate a train label set
                                      and a validation label set
  cn = 0
  for i in range(len(split_label)): # Define the four train label sets
    if i != ind:
      if cn == 0:
        a = split_label[i]
      elif cn == 1:
        b = split_label[i]
      elif cn == 2:
        c = split_label[i]
      else:
        d = split_label[i]
      cn += 1
    else:
      test = split_label[i] # 'i'th label set will be the validation label
                              set

  train = np.hstack((a, b, c, d)) # Combine the train label sets into one
                                    set horizontally
  return train, test

def euc_dist(X, Y): # Efficient way to calculate the euclidean distance with
                      Einstein summation convention
  distances = np.sqrt(  # (X - Y) ** 2
    np.einsum('ij, ij ->i', X, X)[:, None] +  # = X ** 2
    np.einsum('ij, ij ->i', Y, Y) -  # + Y ** 2
    2 * X.dot(Y.T))  # - 2 * X * Y
  return distances

def cross_val(data, label, k): # Function to implement KNN with 5-fold cross
                                 -validation
  acc_list = []
  for i in range(len(data)):
    com_data = combine_data(i, data) # Preparing the train set and
                                       validation set
    com_label = combine_label(i, label)
    train_data = com_data[0]
    val_data = com_data[1]
    train_label = com_label[0]
    val_label = com_label[1]
    # Implement KNN with train and validation sets
    result = KNN(train_data, train_label, val_data, val_label, k)
    acc = result[1] # Store the accuracy
    acc_list.append(acc)
```

```
    avg_acc = np.mean(acc_list) # Compute the average accuracy of the cross-
                                        validation
    return avg_acc, acc_list

def optimal_k(data, label, re): # Funtion to find the best hyper-parameter '
                                        k'
    mix_trd = shuffle(data) # Shuffle the dataset
    mix_trl = shuffle(label) # Shuffle the labels
    split_data = np.array_split(mix_trd, 5) # Split dataset into 5
    split_label = np.array_split(mix_trl, 5) # Split labels into 5
    accuracy = []
    ind_acc = []
    for k in range(1, re, 1): # Iterations with k from 1 to 're'
        cv_result = cross_val(split_data, split_label, k) # 5-fold cross-
                                        validation result
        k_acc = cv_result[0] # Average accuracy of the cross-validation when k =
                                        'k'
        ind_acc.append(cv_result[1])
        accuracy.append(k_acc)
        print("Average accuracy of 5-fold cross-validation with k=", k, ":",
                                        k_acc*100, "%")
    opt_k_index = accuracy.index(max(accuracy)) # Index of the maximum mean
                                        accuracy between k=1 and k='re'
    opt_k = opt_k_index + 1 # Best hyper-parameter 'k'
    return opt_k, accuracy, ind_acc

def score(predict, y_test): # Calculate the prediction accuracy
    return np.mean(predict == y_test)

def KNN(tr_d, tr_l, tt_d, tt_l, k): # Function to implement the designed KNN
                                         Model
    distances = euc_dist(tt_d, tr_d) # Calculate the euclidean distances of
                                        the given train and test datasets
    min_ind = np.argpartition(distances, k-1)[:,:k] # Find k-nearest neighbor
                                        data points
    best_neighbours = tr_l[min_ind] # Get labels of each k-nearest neighbor
                                        data points
    # Predicted label of the test sample will be the label which appears the
                                        most in 'best_neighbours[]'
    pred = np.array([Counter(sorted(row, reverse=True)).most_common(1)[0][0]
                                        for row in best_neighbours])
    accuracy = score(pred, tt_l) # Calculate the prediction accuracy
    return pred, accuracy

def pca_fit(data): # To get the eigenvalues and eigenvectors of the input
                                        dataset
    means = np.mean(data, axis = 0) # Mean of each data samples
    data = data - means # Normalization via the mean centring
    sq_m = np.dot(data.T, data) # the covariance matrix
    (evals, evecs) = np.linalg.eig(sq_m) # Eigen-decomposition
    return evals, evecs # (Eigenvalues, Eigenvectors)

def pca_transform(data, evecs, n_dim): # To implement PCA based on the
                                        obtained eigenvectors
    result = np.dot(data, evecs[:, 0:n_dim]) # 'n_dim' principal components
    return result
```

## C SELF-DEFINED FUNCTIONS IN SVM_MAIN.PY

The repeated functions are not provided here.

```
    def combine_data(ind, split_data):  # Method to separate a train dataset
                                        and a validation dataset
      cn = 0
      for i in range(len(split_data)): # For 3-fold cross-validation, combine
                                        two datasets for a train set
        if i != ind:
          if cn == 0:
            a = split_data[i]
          else:
            b = split_data[i]
          cn += 1
        else:
          test = split_data[i] # 'i'th dataset will be the validation dataset

      train = np.vstack((a, b)) # Combine the train datasets into one set
                                        vertically
      return train, test

    def combine_label(ind, split_label): # Method to separate a train label
                                        set and a validation label set
      cn = 0
      for i in range(len(split_label)): # Define the two train label sets
        if i != ind:
          if cn == 0:
            a = split_label[i]
          else:
            b = split_label[i]
          cn += 1
        else:
          test = split_label[i] # 'i'th label set will be the validation label
                                         set

      train = np.hstack((a, b)) # Combine the train label sets into one set
                                        horizontally
      return train, test

    # 3-Fold Cross-validation
    def cross_val(name, data, label, c, p): # 'name'=kernel type, 'c'=Cost, 'p
                                        '=kernel parameter
      acc_list = []
      for i in range(len(data)):
        com_data = combine_data(i, data)
        com_label = combine_label(i, label)
        train_data = com_data[0]
        val_data = com_data[1]
        train_label = com_label[0]
        val_label = com_label[1]
        clf = kernel_SVM(name, train_data, train_label, c, p) # Implement
                                        kernel SVM
        acc = clf.score(val_data, val_label) # SVM prediction accuracy
        acc_list.append(acc)
      avg_acc = np.mean(acc_list)
      return avg_acc

    def optimal_params(name, data, label, C, para): # To find the four best
                                        hyper-parameter sets
```

```
    mix_trd = shuffle(data) # Shuffle the dataset
    mix_trl = shuffle(label) # Shuffle the labels
    split_data = np.array_split(mix_trd, 3) # Split dataset into 3
    split_label = np.array_split(mix_trl, 3) # Split labels into 3
    accuracy = []
    param_set = []
    avg_coord = []
    for c in C: # Iterations for the candidates of the Cost parameter 'C'
      for p in para: # Iterations for the candidates of Kernel parameter 'P'
        mean_acc = cross_val(name, split_data, split_label, c, p) # Mean
                                    accuracy of 3-fold cross-validation
        accuracy.append(mean_acc)
        set_param = (c, p) # Set of the hyperparameters
        param_set.append(set_param)
        avg_coord.append([c, p, mean_acc]) # Hyper-parameter sets and their
                                    mean accuracy
        print("Average accuracy of 3-fold cross-validation with
                                    hyperparameters (C, Kernel) =",
                                    set_param, ":", mean_acc * 100, "%")
    opt_p_index = accuracy.index(max(accuracy))
    opt_hp = param_set[opt_p_index] # An optimal hyper-parameter set
    return opt_hp, avg_coord

  def deep_optimal_params(name, data, label, C, para): # To find the best
                                    hyper-parameter set
    mix_trd = shuffle(data) # Shuffle the dataset
    mix_trl = shuffle(label) # Shuffle the labels
    split_data = np.array_split(mix_trd, 3) # Split dataset into 3
    split_label = np.array_split(mix_trl, 3) # Split labels into 3
    accuracy = []
    param_set = []
    avg_coord = []
    for id, c in enumerate(C): # Iterations for the candidates of the Cost
                                    parameter 'C' and their indices
      p = para[id] # Kernel parameter when the index is equal 'id'
      mean_acc = cross_val(name, split_data, split_label, c, p)  # Mean
                                    accuracy of 3-fold cross-validation
      accuracy.append(mean_acc)
      set_param = (c, p)  # Set of the hyperparameters
      param_set.append(set_param)
      avg_coord.append([c, p, mean_acc])
      print("Average accuracy of 3-fold cross-validation with
                                    hyperparameters (C, Kernel) =",
                                    set_param, ":", mean_acc * 100, "%")
    opt_p_index = accuracy.index(max(accuracy)) # Index of the maximum
                                    accuracy hyper-parameter set
    opt_hp = param_set[opt_p_index]  # Best hyper-parameter set
    return opt_hp, avg_coord

  def kernel_SVM(name, data, label, c, param): # Function to determine the
                                    type of kernel SVM model
    if name == 'poly': # If name is 'poly', implement the polynomial kernel
                                    SVM
      d = param # Kernel parameter = Degree
      clf = svm.SVC(kernel='poly', C=c, degree=d)

      # Decide the multi-classification method of SVM
      clf = OneVsOneClassifier(clf) # One-Vs-One Classifier (Default)
      # clf = OneVsRestClassifier(clf) # One-Vs_Rest Classifier
      clf.fit(data, label)
```

```
        return clf

    elif name == 'rbf': # Else if name is 'rbf', implement the RBF kernel
                                    SVM
        g = param # Kernel parameter = Gamma
        clf = svm.SVC(kernel='rbf', C=c, gamma=g)

        # Decide the multi-classification method of SVM
        clf = OneVsOneClassifier(clf) # One-Vs-One Classifier (Default)
        # clf = OneVsRestClassifier(clf) # One-Vs_Rest Classifier
        clf.fit(data, label)
        return clf

    else: # Else Error
        print("Kernel name is incorrect!")
```

## D KEY PART OF SVM_MAIN.PY

Here, the implementation code of SVM in the main part is provided.

```
# Hyper-parameter candidates
C = [0.1, 1, 10, 100, 1000]
Gamma = [0.01, 0.1, 1, 10, 100]
Degree = [2, 3, 4, 5, 6]

# Reshaping the datasets
x_train = np.asmatrix(x_train[:(60000 * n_dim)]).reshape(60000, n_dim)
x_test = np.asmatrix(x_test).reshape(10000, n_dim)
y_train = y_train[:60000]

# Creating a random validation set (12,000 samples)
xtr , x_val, ytr, y_val = train_test_split(x_train, y_train, stratify=
                                    y_train, test_size=0.2)

########################## Standard Optimization ################

start_opt = timeit.default_timer()  # Start timer to count the standard
                                    optimization time
name = 'poly' # Choose 'poly' or 'rbf'
optimization = optimal_params(name, x_val, y_val, C, Degree) # Standard
                                    optimization process

stop_opt = timeit.default_timer()  # Stop timer for the standard
                                    optimization time of SVM
print('Standard optimization time of SVM:', stop_opt - start_opt)

coord_1 = np.array(optimization[1])
avg_coord = np.transpose(coord_1, axes = None)

stand_opt_id = np.argpartition(avg_coord[2], -4)[-4:] # Select four best
                                    hyperparameter sets
stand_c = avg_coord[0][stand_opt_id] # Four different cost parameters
stand_k = avg_coord[1][stand_opt_id] # Four different kernel parameters

########################## Deep Optimization ##########################

start_deep = timeit.default_timer()  # Start timer to count the deep
                                    optimizatoin time
```

```
deep_optimization = deep_optimal_params(name, x_train, y_train, stand_c,
                                        stand_k)
hyperparameters_set = deep_optimization[0] # Best hyper-parameter set

opt_C = hyperparameters_set[0] # Optimal Cost penalty
opt_K = hyperparameters_set[1] # Optimal Kernel parameter
print("Kernel Function:", name)
print("Best Hyperparameters (C, Kernel):", hyperparameters_set)

stop_deep = timeit.default_timer()  # Stop timer for the deep optimization
                                     time of SVM
print('Deep optimization time of SVM:', stop_deep - start_deep)

################################## Training ####################

start_train = timeit.default_timer()  # Start timer to count the training
                                      time
clf = kernel_SVM(name, x_train, y_train, opt_C, opt_K) # Final SVM
                                      classification
print("Training output with optimal hyperparameters:",clf)

stop_train = timeit.default_timer()  # Stop timer for the training time of
                                     SVM
print('Training time of SVM:', stop_train - start_train)

################################## Testing ###################
start_test = timeit.default_timer()  # Start timer to count the testing
                                     time
y_pred = clf.predict(x_test) # Predict labels of the test dataset based on
                              the training result
result = clf.score(x_test, y_test)
stop_test = timeit.default_timer()  # Stop timer for the testing time of
                                    SVM
print('Testing time of SVM:', stop_test - start_test)
```