COMP5329
Deep Learning
Assignment 1

University of Sydney
April 24, 2021

Team members
Zuqian Song [480240077]
Chang Zhao [480482013]
Zirong Guo [500417317]

Contents

1 Introduction

With the fast growth deep learning community and improvements of the algorithms , deep learning algorithms are broadly applied to vast areas of research and data analysis in the industry. The goal of the study is to promote the general understanding of how a MLP model works under different settings. The aim is to gain insights on how different optimizer would affect the learning of the model; how different hyper-parameters setting would affect the models and the performance; how different regularization methods impact the model's performance on the testing set. With our work, others who just started with deep learning would have a deeper understanding of a MLP model and how to tune their own models better with reasonable understanding of behind each setting.

There are many reasons for implementing MLP. First of all, after being familiar with and successfully implementing MLP with code, one can have a deep understanding of the structure of the neural network and the algorithms used; in addition, one can lay a solid foundation for the future study of deep learning.

2 Methods

The original shape of the training set is equal to (50000, 128), where 50000 is the number of training data and 128 is the feature dimensions. The shape of training labels is (50000, 1). There are 10 different classes in y, which encoded from 0 to 9.

2.1 Data Pre-processing

2.1.1 Standardisation and Normalisation

We've defined two functions for standardisation and normalisation respectively. Standardization takes the mean and standard deviation of each feature dimension, and transforms the feature data to follow the standard normalization distribution. The same process is applied to both validation set and test set. Below is the mathematical formula

$$X = \frac{X-\mu}{\sigma}$$

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling. Below is the mathematical formula

$$X = \frac{X-X_{min}}{X_{max}-X_{min}}$$

2.1.2 Data set segmentation and transpose

Before training the model, the original training data is arbitrarily split into training set and validation set, where training set is 98% of the original training data set size and the validation set is 2% by default. With validation set, we can obtain an unbiased estimation of the performance of the model while training the model and prevent it from over fitting the training samples.

The training set X is transposed to shape (128, 49000) and the training label transposed and transformed to one hot encoding with shape (10, 49000) before feeding them into the neural network, so do validation set and testing set. The purpose of one hot encoding is for calculating cross entropy loss because the output is a 10 by m vector, here m is the number of samples in each batch, during the training stage.

2.2 The Principle of Different Modules

There are 3 main modules in our models, which are activation module, hidden layer modules and MLP module. In short, the activation module contains the calculation of relu, tanh, logistic and softmax function and their calculation of gradient; the hidden layer modules define the behavior of a single layer in the models; the MLP manipulates the behaviors of a collection of layers. In addition to those 3 main modules above, we also explain the algorithm we used in our neural network such as the regularisation algorithm, the optimisation algorithm and so on.

Next, this section will have an in depth discussion about each module. To make the code easy to understand, we use Z to represent the result of linear forward step in neuron network, $Z = WX + B$. A is used to represent the result of an activation function $A = g(Z)$. $\hat{y}$ is the model's prediction and y is the true label.

2.2.1 Activation Function Module

There are 2 kind of activation functions in this assignment: tanh and ReLU.

tanh

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

below is the code version

```
def __tanh(self, x):
        return np.tanh(x)
```

and the tanh derivative function

```
def __tanh_deriv(self, a):
        # a = np.tanh(x)
        return 1.0 - a**2
```

ReLU

$$Relu(z) = max(0, z)$$

below is the code version:

```
def __relu(self, x):
        return np.maximum(0,x)
```

and the ReLU derivative function

```
def __relu_deriv(self, a):
        a[a <= 0] = 1e-8
        a[a > 0] = 1
        return a
```

In the above code, the formal parameter 'x' is the input parameter of the activation function of forward propagation, and the formal parameter 'a' is the input of the backward propagation activation function (or the output of the forward propagation activation function), because the partial derivative $\frac{\partial a}{\partial x}$ can be represented by the output 'a' of the activation function.

2.2.2 Hidden Layer Module

Hidden layer is implemented in an object-oriented way, the following is the code

```
class Hiddenlayer(object):
  def __init__(self, number_in, number_out, W=None, B=None,
      activation_this_layer='tanh', activation_last_layer='tanh'):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

if define the attributes and functions of each hidden layer, like the weight W, bias B, the dimension of input and output and the activation function of this layer and last layer.

In addition, forward propagation function and backward propagation function are defined and implemented in this module.

```
# define forward propergation function
def forward(self, input, drop_out_rate, isOutputLayer=False, batchNorm=False):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

```
# define backward prop
def backward(self, input, batchSize, lambd, penalty, batchNorm):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

2.2.3 MLP Module

MLP(Multiple layer perceptron) is also implemented in an object-oriented way, the following is the code

```
# define the NN
class MLP(object):
  def __init__(self, unitsNumber_list, typeOfActivation):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

if define the attributes and functions of the whole neural network, like the network structure because it initialize each hidden layer object. More importantly, it defines the sequence and input/output of each layer for backward-prop/forward-prop, the loss function, the gradient update function, training function and the predict function. Below is the code

```
def forward(self, input, drop_out_rate,batchNorm=False):
    # The specific code implementation is omitted here. You can see the details in
       the code.
def backward(self, loss_deriv, batchSize, lambd, penalty,batchNorm):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

```
# calculate cross entropy
def crossEntropyLoss_andDeriv(self, Y_hat, Y, lambd=0.1, penalty='l2'):
    # The specific code implementation is omitted here. You can see the details in
       the code.
```

```
def updatePara(self, learningRate, beta_momentum, momentum, beta_RMS, adam,
    batchNorm):
    # The specific code implementation is omitted here. You can see the details in
        the code.
```

```
def fit(self, X, y, learning_rate=0.1, learning_rate_dacay=0.9,
          epochs=100, print_interval=10,
          batchSize=1024, iter_each_epoch=3,
          beta_momentum=0.9, momentum=False,
          beta_RMS=0.999, adam=False,
          lambd=0.1, drop_out_rate=0.5, penalty='l2', batchNorm=False,
              early_stop=False, calc_val=False):
    # The specific code implementation is omitted here. You can see the details in
        the code.
```

```
def predict(self, X, Y):
    # The specific code implementation is omitted here. You can see the details in
        the code.
```

2.2.4 Mini-batch

In order to prevent the excessive amount of batch gradient calculations from causing memory overflow or the randomness of the SGD, the gradient is too large and it is not easy to converge. It seems that Mini-batch is the best solution for the problems above.

The idea of Mini-batch is that define a batch size, integer value, within (0,samples], then update the gradient after going through each batch. Below is the implementation in code

```
# shuffled the index of training data in each epoch
shuffled_x_indexes = np.random.permutation(X.shape[1])
```

```
for it in range(iter_each_epoch):
    # get the index of the first batch
    indexes =
        shuffled_x_indexes[it*self.batchSize:(it*self.batchSize+self.batchSize)]

    # pick data of one batch by indexes
    miniBatch = X[:,indexes]
```

2.2.5 Learning Rate Decay

Large learning rate might cause that the model can not converge or loss increase and small learning rate might cause that the process of convergence takes a large amount of time.

Therefore, we define our own algorithm for learning rate decay, below is the code

```
learning_rate *= learning_rate_dacay
```

and it performs good if we set the value to 0.9 0.99 while training.

2.2.6 Early Stopping

Since the training process will face the overfitting problem and we hope the process will stop and show the result when the accuracy of validation set does not increase any more. Here, we define our own algorithm for early stopping, if no such one value is greater than the maximum value in the later 50 epochs then return. Below is the implementation

```
# if the val accuracy decrease, descend=True
if accuracy_val_arr[int(times)]-accuracy_val_arr[int(times)-1] < 0:
  descend = True

if descend:
  tmp_max = accuracy_val_arr[int(times)]
  if tmp_max > max_val_acc:
    # means meets a new maximum value for val_acc
    max_val_acc = tmp_max
    early_stop_count=0
    # here save all parameters for current model
    save_current_model()
  else:
    # means the current value is less than the maximum value
    early_stop_count+=1
    # if no value is greater than the max in the later 50 epochs
    # then return
    if early_stop_count>=50:
      current_epoch = times
      return loss_return, accuracy_train, val_loss_return, accuracy_val_arr,
          current_epoch
```

2.2.7 Optimizer Module

Momentum

The basic idea to compute an exponentially weighted average of the gradients, and then use that gradient to update the weights. Below is the mathematical formula

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

and the code version (here the bias correction make the computation of the average more accurately)

```
# mommentum
layer.v_dw = beta_momentum*layer.v_dw + (1-beta_momentum) * layer.dw
layer.v_db = beta_momentum*layer.v_db + (1-beta_momentum) * layer.db
# bias correct
v_dw_correct = layer.v_dw/(1-beta_momentum**i)
v_db_correct = layer.v_db/(1-beta_momentum**i)
# update W and B
layer.W -= learningRate * layer.v_dw
layer.B -= learningRate * layer.v_db
```

Adam (Adaptive Momentum Algorithm)
Adam algorithm is basically taking Momentum and RMSProp and putting them together which is performed better than the Momentum. Below is the mathematical formula

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

here m is the parameter in RMSProp algorithm

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

below is the code version

```
# mommentum
layer.v_dw = beta_momentum*layer.v_dw + (1-beta_momentum) * layer.dw
layer.v_db = beta_momentum*layer.v_db + (1-beta_momentum) * layer.db
# RMSprop
layer.s_dw = beta_RMS*layer.s_dw + (1-beta_RMS) * layer.dw * layer.dw
layer.s_db = beta_RMS*layer.s_db + (1-beta_RMS) * layer.db * layer.db

# bias correct
s_dw_correct = layer.s_dw/(1-beta_RMS**i)
s_db_correct = layer.s_db/(1-beta_RMS**i)

# update W and B
layer.W -= learningRate * v_dw_correct/(np.sqrt(s_dw_correct)+1e-8)
layer.B -= learningRate * v_db_correct/(np.sqrt(s_db_correct)+1e-8)
```

2.2.8 Soft-max Function and Cross-Entropy Loss module

Soft-max function
Since the current problem to be solved is a multi-classification problem, it is necessary to use the Soft-max activation function in the output layer. The Soft-max activation function replaces the results of n units in the output layer with values in the interval [0,1], and the sum of these values is 1. Below is the mathematical formula of Soft-max function

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$$

and the implementation in code

```
# softmax function
def __softmax(self, Z):
    expZ = np.exp(Z - np.max(Z))
    return expZ / expZ.sum(axis=0, keepdims=True)
```

Since the partial derivative of Cross Entropy Loss to the input value Z of Soft-max function is $\hat{y} - y$, there is no need to obtain the partial derivative of Soft-max, thereby reducing the amount of calculation

Cross entropy loss
For the Soft-max multi-classification problem, the cross entropy is expressed as

$$L(\hat{y}, y) = \frac{1}{M} \sum_{c=1}^{M} y_c \log(\hat{y}_c + \epsilon)$$

here M is equal to batch size. Below is the implementation in code

```
loss = -np.mean(Y * np.log(Y_hat + 1e-8),axis=1) + self.l2_penalty(lambd)
```

2.2.9 Regularisation Module

Dropout module
The intuition of Dropout is that can't rely on any one feature, so have to spread out weights to each units so that shrink the weights. The implementation of the Dropout is to go through each of the layer and set some probability of eliminating a node in neural network.

```
# dropout
if isOutputLayer == False and drop_out_rate >= 0:
    # creating mask for dropout and save the mask
    self.mask = np.random.uniform(0,1,size=(self.unitsNumber,1)) >= drop_out_rate
    self.A = self.A * self.mask
    self.A /= (1-drop_out_rate)
```

Weight decay (L2 regularisation) module
As the loss of training set decreased, overfitting problem will occur. The most intuitive solution is to add a penalty term to loss, so that every time the weight W is calculated in back-propagation, W will be reduced to a certain extent, so as not to overfit the data in the training set.

$$Loss = Error(Y - \widehat{Y}) + \frac{\lambda}{2m} \sum_{1}^{n} w_i^2$$

when calculate the W for each layer, there will be a penalty before W

$$W = (1 - \lambda * \frac{\alpha}{m})W - \alpha * dW$$

Below is the implementation in code

```
# calculate l2 penalty
def l2_penalty(self, lambd):
    l2_penalty = 0
    for layer in self.__layers:
      l2_penalty += np.sum(np.square(layer.W))
    l2_penalty = l2_penalty * lambd / 2 / self.batchSize
    return l2_penalty
```

```
# calculate Cross Entropy loss with l2 penalty
if penalty == 'l2':
    loss = -np.mean(Y * np.log(Y_hat + 1e-8),axis=1) + self.l2_penalty(lambd)
```

update the derivative of weights in each layer with l2 penalty

```
# update the derivative of weight with l2 penalty
self.dw = np.dot(dz,self.A_input.T)/batchSize + self.W * lambd / batchSize
```

2.2.10 Batch Normalisation

For a deeper network, subtle changes in the shallow layer will be amplified as they propagate in the network, resulting in a serious shift in the parameters of the deeper layer. This shift is called Internal Covariate Shift. And Batch Normalisation is to reduce this unnecessary offset, thereby speeding up training and generating reliable models.

The idea of Batch Normalisation is to add a Batch Normalisation function between the linear function and the activation function in each layer.

Batch normalisation in forward

Below is the Batch Normalisation function in forward-propagation

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

$y_i^{(k)}$ is the input of activation and below is the formula for getting $\hat{x}_i^{(k)}$

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}$$

$x_i^{(k)}$ is the output of linear function(Z=WX+B) in each layer, and below is the formula for getting $\mu_B^{(k)}$ and $\sigma_B^{(k)^2}$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$$

The implementation of code is showed below

```
# Batch norm (to solve Internal Covariate Shift problem)
self.mean_Z = np.mean(self.Z,axis=1)[:,np.newaxis]
self.sqr_var = np.sqrt(np.var(self.Z,axis=1)+1e-8)[:,np.newaxis]
self.Z_norm = (self.Z-self.mean_Z) / self.sqr_var
self.Z_wave = self.Z_norm * self.gamma_norm + self.beta_norm
```

Batch normalisation in backward

Below is the batch normalisation in backward, first is the derivative of γ and β

$$\frac{\partial l}{\partial \gamma^{(k)}} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i^{(k)}} \hat{x}_i^{(k)}$$

$$\frac{\partial l}{\partial \beta^{(k)}} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i^{(k)}}$$

then the derivative of $\hat{x}_i^{(k)}$, $\sigma_B^{(k)^2}$ and $\mu_B^{(k)}$

$$\frac{\partial l}{\partial \hat{x}_i^{(k)}} = \frac{\partial l}{\partial y_i^{(k)}} \gamma^{(k)}$$

$$\frac{\partial l}{\partial \sigma_B^{(k)^2}} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i^{(k)}} (x_i^{(k)} - \mu_B^{(k)}) \left(-\frac{\gamma^{(k)}}{2} (\sigma_B^{(k)^2} + \epsilon)^{-3/2} \right)$$

$$\frac{\partial l}{\partial \mu_B^{(k)}} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i^{(k)}} \frac{-\gamma^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} + \frac{\partial l}{\partial \sigma_B^{(k)^2}} \frac{1}{m} \sum_{i=1}^{m} (-2) \cdot (x_i^{(k)} - \mu_B^{(k)})$$

at last, the derivative of $x_i^{(k)}$

$$\frac{\partial l}{\partial x_i^{(k)}} = \frac{\partial l}{\partial \hat{x}_i^{(k)}} \frac{1}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} + \frac{\partial l}{\partial \sigma_B^{(k)^2}} \frac{2(x_i^{(k)} - \mu_B^{(k)})}{m} + \frac{\partial l}{\partial \mu_B^{(k)}} \frac{1}{m}$$

below is the implementation

```
dz_wave = dz
# d_gamma
self.d_beta_norm = np.sum(dz_wave,axis=1)[:,np.newaxis]
# d_beta
self.d_gamma_norm = np.sum(dz_wave * self.Z_norm, axis=1)[:,np.newaxis]
# d_Z_norm
self.d_Z_norm = dz_wave * self.gamma_norm
# d_ivar
d_ivar = np.sum((self.Z - self.mean_Z) * self.d_Z_norm, axis=1)[:,np.newaxis]
# d_mu1
d_mu1 = self.d_Z_norm * 1 / self.sqr_var
# d_sqr_var
d_sqr_var = -1 / (self.sqr_var**2) * d_ivar
# d_var
d_var = 0.5 * 1 / np.sqrt(self.sqr_var) * d_sqr_var
# d_sq
d_sq = 1. / dz_wave.shape[1] * np.ones((dz_wave.shape[0],dz_wave.shape[1])) * d_var
# d_mu2
d_mu2 = 2 * (self.Z - self.mean_Z) * d_sq

d_x1 = d_mu1 + d_mu2
d_mu = -1 * np.sum(d_x1, axis=1)[:,np.newaxis]
d_x2 = 1. / dz_wave.shape[1] * np.ones((dz_wave.shape[0],dz_wave.shape[1])) * d_mu
dz = d_x1 + d_x2
```

2.2.11 Scoring Metrics

Below are some metrics used in this assignment. For ease of presentation, we only summarise the formula of the version of binary classification which share the same theory with multiple classification and the concrete result will be showed in experiment part. Below is the mathematical formula

$$\text{Accuracy: } \frac{TP + TN}{TP + FP + TN + FN}$$

$$\text{Precision: } \frac{TP}{TP + FP}$$

$$\text{Recall: } \frac{TP}{TP + FN}$$

$$\text{F1-score: } 2\text{x}\frac{\text{precision x recall}}{\text{precision + recall}}$$

below is the implementation

```
test_label = test_label.flatten()
result_visualization = np.zeros((numberOfClass, numberOfClass))
# best_model_on_test[0] is the prediction of targets of test set
```

```
# best_model_on_test[1] is the targets of test set
for i in range(len(test_label)):
    row = best_model_on_test[1][i]
    column = best_model_on_test[0][i]
    result_visualization[row][column] += 1
result_visualization # row results the true value, column results predicted value

TP = np.zeros((1,numberOfClass)).flatten()
FP = np.zeros((1,numberOfClass)).flatten()
TN = np.zeros((1,numberOfClass)).flatten()
FN = np.zeros((1,numberOfClass)).flatten()
Accuracy = np.zeros((1,numberOfClass)).flatten()
Precision = np.zeros((1,numberOfClass)).flatten()
Recall = np.zeros((1,numberOfClass)).flatten()
F1 = np.zeros((1,numberOfClass)).flatten()

sum_confus = np.sum(result_visualization)
for i in range(numberOfClass):
  TP[i] = result_visualization[i][i]
  FP[i] = np.sum(result_visualization[:,i])-TP[i]
  FN[i] = np.sum(result_visualization[i,:])-TP[i]
  TN[i] = sum_confus - FP[i] - FN[i] + TP[i]

  Accuracy[i] = (TP[i] + TN[i])/(TP[i]+FP[i]+TN[i]+FN[i])
  Precision[i] = TP[i]/(TP[i]+FP[i])
  Recall[i] = TP[i]/(TP[i] + FN[i])
  F1[i] = 2*Precision[i]*Recall[i]/(Precision[i]+Recall[i])
```

2.3 Design of the Best Model

The best model has 4 layers shown as follow:

$$\underset{256\times N}{Z^{[1]}} = \underset{256\times 128}{W^{[1]}} \times \underset{128\times N}{X} + \underset{256\times 1}{B^{[1]}} \tag{1}$$

$$\underset{256\times N}{A^{[1]}} = tanh(Z^{[1]}) \tag{2}$$

$$\underset{64\times N}{Z^{[2]}} = \underset{64\times 256}{W^{[2]}} \times \underset{256\times N}{A^{[1]}} + \underset{64\times 1}{B^{[2]}} \tag{3}$$

$$\underset{64\times N}{A^{[2]}} = tanh(Z^{[2]}) \tag{4}$$

$$\underset{64\times N}{Z^{[3]}} = \underset{64\times 64}{W^{[3]}} \times \underset{64\times N}{A^{[3]}} + \underset{64\times 1}{B^{[3]}} \tag{5}$$

$$\underset{64\times N}{A^{[3]}} = ReLU(Z^{[3]}) \tag{6}$$

$$\underset{10\times N}{Z^{[4]}} = \underset{10\times 64}{W^{[4]}} \times \underset{64\times N}{A^{[4]}} + \underset{10\times 1}{B^{[4]}} \tag{7}$$

$$\underset{10\times N}{A^{[4]}} = softmax(Z^{[4]}) \tag{8}$$

The loss function is cross entropy loss. Batch normalization is apply to each Z values mentioned above before entering the activation functions. Optimizer is Adam. The mini-batch size is set to 500. The learning rate is set to 0.005 with learning rate decay in place. The learning rate decay rate is set to 0.9. Early stop is turn on. L2 regularization is turn on and the penalty lambda is set to 0.3.

3 Experiments and Result

At the beginning of our experiment, we have debugged and tested the number of hidden layer and the number of units in each layer.

The optimal number of hidden layer is 4 and the impact on the current data set did not increase when the number of layers are greater than 4. In addition, for the number of units in each layer, it is found that if the number is too large, it will easily cause overfitting; but if the number is too small, underfitting will occur. However, the number of units in each layer has less impact on result, between 64 and 256, so we choose the value performed best.

Learning rate decay is to decrease the learning rate in training process. We define the most suitable formula for our learning rate decay and we found that 0.9 performed best in our model.

Small batch size can boost the computation process so we choose the size as small as possible, but too small batch size makes batch normalisation performed poorly. We have done several experiments for batch size within [100,500,1000] and we pick 500 which performed best.

Therefore, after our experiment on several values, we found that the network structure below is the best.

Note: The following experiments are conducted with the following parameters fixed:

1st hidden units	2nd hidden units	3rd hidden units	learning rate decay	batch size	early stop
256	64	64	0.9	500	ON

Table 1: Fixed Parameters

3.1 Experimental Equipment Configuration

Below is the information of local environment

Hardware	CPU	Memory	Operating system
2019 Macbook-Pro	Intel i9	32GB	MacOS Big Sur 11.2.3

Table 2: Fixed Parameters

We also use Colab of CPU running time to run our code. The details of environment of Colab are showed on colab.

3.2 Learning Rate Experiment

We begin our experiments by exploring different learning rate and try to find the relatively optimal learning rate. The optimizer is set to Adam. The non-linear functions for each layer is relu, tanh, tanh and softmax. The non-linear functions are arbitrarily chose. The regularization modules are completely turn off. The learning rate will arbitrarily start at 0.0001 and scale up to 0.5. The result shown in Table 1.

Learning Rate	0.0001	0.0005	0.001	0.005	0.01	0.05	0.1	0.5
Train Accuracy	31.05%	43.57%	48.05%	59.35%	52.47%	55.86%	9.98%	9.96%
Validation Accuracy	31.8%	42.6%	45.00%	51.00%	47.19%	50.6%	11.4%	12.00%

Table 3: The different accuracy results of different learning rate

The result shows when setting the learning rate to 0.005 gives the best validation accuracy. Thus, in the following experiment, we will always set the learning rate to 0.005.

3.3 Activation Function Experiments

The learning rate used here is the best learning rate of Section 3.2, which is 0.005. Batch normalization has been turned on. Optimizer is set to Adam. Other parameters and hyperparameters use the same settings mentioned at the begining of section 3. The detailed experiment result is the following:

list of non-linear	Train Accuracy	Validation Accuracy
'relu','relu','relu','softmax'	44.48%	43.60%
'tanh','relu','relu','softmax'	65.16%	55.00%
'relu','tanh','relu','softmax'	61.79%	53.60%
'relu','relu','tanh','softmax'	62.38%	53.20%
'tanh','tanh','relu','softmax'	68.42%	56.39%
'tanh','relu','tanh','softmax'	66.70%	53.00%
'relu','tanh','tanh','softmax'	59.58%	50.20%
'tanh','tanh','tanh','softmax'	64.51%	53.40%

Table 4: Performances of utilizing nolinear functions

The result shows the [tanh, tanh, relu, softmax] combination gives the best validation accuracy. The result also suggests, when setting all the hidden layers to either relu or tanh is not optimal.

list of linear	Train Accuracy	Validation Accuracy
'linear','linear','linear','softmax'	18.0%	19.4%

Table 5: Performances of utilizing linear functions

Additionally, for ablation studies, We removed the activation function of all hidden layers except the output layer and the train accuracy and validation accuracy were very low. Therefore, this result conforms that the multiple layer perceptron can not solve the non-linear problem without activation function.

3.4 Optimizer Comparisons

Use the best result from Section 3.3 and 3.4 by setting activation function combination to [tanh, tanh, relu , softmax] and learning rate to 0.005, the regularization turn off, only varying optimizers. The detailed experiment result is as follows:

Optimizer	Train Accuracy	Validation Accuracy	Running Time
Mini-Batch without optimizer	50.91%	47.60%	1059s
Momentum	46.69%	47.60%	646s
Adam	66.70%	56.0%	299.67s

Table 6: Performances using different optimization methods

Adam algorithm includes the advantages of Rmsprop and momentum. In our case, Adam

optimizer achieves significantly better result comparing to Mini-batch without optimizer and momentum in terms of both running time and the validation accuracy. Short running of Adam suggests it will make the gradient converge much faster.

3.5 Regularization Techniques Comparisons

Lastly, we experiment on different combination of regularization techniques. Similarly, we use the best result from the experiments mentioned above by setting the learning rate to 0.005, optimizer to Adam, activation list set to [tanh, tanh, relu, softmax]. L2 regularization is to limit the growth of weights in the network by penalizes weights that has high values when calculating the loss. Drop out is simply retain the value of each unit with given probability p. Batch normalization is used to reduce covariate shift by normalizing the value z to follow a Gaussian distribution. During experiment of comparing different regularization approach, learning rate is chosen using the best result from Section 3.3, and the result is listed in Table 2, which is 0.005.

One important thing is that we have done several grid search experiments on choosing the optimal value of dropout rate and lambda of L2 regularisation. It seems that setting dropout rate to 0.2 and lambda of L2 to 0.3 perform best. Therefore, the dropout rate when enabled is set to 0.2 and the L2 penalty λ is set to 0.3 when enabled.

Learning Rate	L2	Drop out	Batch Norm	Validation Accuracy
0.005	ON	OFF	OFF	51.4%
0.005	ON	ON	OFF	12.0%
0.005	ON	OFF	ON	56.99%
0.005	ON	ON	ON	54.2%
0.005	OFF	ON	ON	53.8%
0.005	OFF	ON	OFF	12.0%
0.005	OFF	OFF	ON	53.6%
0.005	OFF	OFF	OFF	44.0%

Table 7: Performances using different combinations of regularization methods

The result shows the model works the best with batch normalization turn on and L2 turn on. Batch normalization the most effective on improving the validation accuracy, comparing to only using L2 and only using dropout. When drop out is turn on without batch normalization turn on, the result is significant worse. However, when batch normalization is turn on, it rescues the performance. This may because drop out will introduce noise during the process, which could potentially leads to internal covariate shift problem, and batch normalization can effectively solve this kind of problem.

3.6 Best Result with Justification

The best result is generated using the setting mentioned in 2.3. These setting are the best settings from the experiments that generated the highest validation accuracy mentioned above. The result is shown in Table 8.

	Training set	Validation set	Test set
Accuracy	0.684	0.564	0.459
Loss	0.063	0.200	-
Precision	-	-	0.458
Recall	-	-	0.459
F1 score	-	-	0.457
Running time	-	-	279.211s

Table 8: The results are based on the best performance of the test set accuracy

The confusion matrix of this result is shown below:

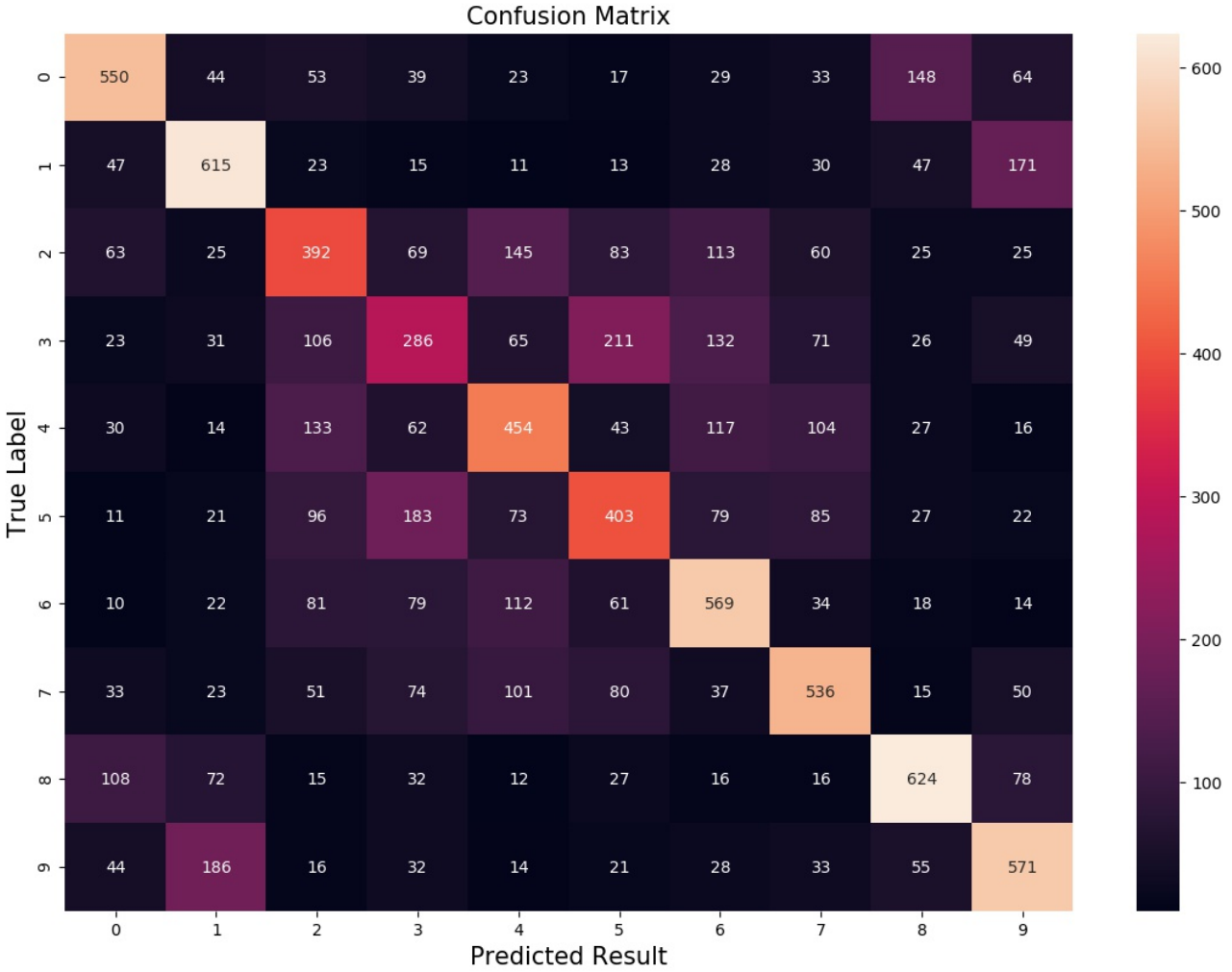

Figure 1: Confusion matrix, rows represents true labels, columns presents predicted labels

There is a obvious and straight diagonal in confusion matrix which means our model performs fairly good, while there are still some defects. For instance, the accuracy of prediction results between 2 and 7 is still low, especially between 2 and 3, and through our qualitative analysis, we get the coupling of these samples may be very high or the characteristics are not obvious. And these prediction errors are probably the main reason for the low accuracy of prediction results. Therefore, we check the precision, recall and F1 score for each class

Metrics \ Targets	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Accuracy	0.920	0.925	0.883	**0.876**	0.880	0.895	0.905	0.909	0.926	0.906
Precision	0.557	0.582	0.365	**0.326**	0.367	0.428	0.480	0.498	0.582	0.482
Recall	0.562	0.578	0.323	**0.288**	0.408	0.382	0.501	0.508	0.602	0.543
F1 score	0.559	0.580	0.342	**0.306**	0.386	0.404	0.490	0.503	0.592	0.511

Table 9: The accuracy, precision, recall and F1 for each class

From the metrics for each class, we can see that In the four kinds of metric, the value of C3 is the minimum, and the value between C2 and C4 is very small. Therefore, the metric of each class conforms our original thoughts, the result shows that the model is struggling with identifying class 2, class 3 and class 4.

Additionally, the model is confused about class 9 with class 1. It miss classified class 9 to class 1 186 times and miss classified class 1 to class 9 171 times. It is also had hard time to distinguish class 3 and class 5. It miss classified class 3 to class 5 211 times and miss classified the class 5 to class 3 183 times.

Below are all hyper-parameter of our best result

```
# dataset size
TRAIN_SIZE = 49000 # traing dataset size
VAL_SIZE = 500 # validation dataset size
TEST_SIZE =10000 # test dataset size
BATCH_SIZE=500 # size of batch
PRINT_INTERVAL=1 # interval for log print
CAL_VAL=True # show/NOT show the validation loss and accuracy in log print

# iteration related para
LEARNING_RATE = 0.005 # the initial learning rate
LEARNING_RATE_DECAY = 1 # the decay rate of the learning
EPOCHS = 300 # number of epochs to run
ITER_EACH_EPOCH=int(TRAIN_SIZE / BATCH_SIZE) # number of batches to iterate in each
    epoch

EARLY_STOP=True # early stopping

# Optomizers
MOMENTUM=False # momentum turned on/off
ADAM=True # adam turned on/off

# Regularisation
DROP_OUT_RATE=-1 # rate of dropout
PENALTY='l2' # use l2 regularisation
LAMBD=0.3 # lambda for l2

# Batch norm
BATCH_NORM=True # turn on/off batch normalisation
```

4 Conclusion and Discussion

Our work shows that when choosing the optimizer, Adam algorithms not only converge much faster comparing to Mini-batch without optimizer and momentum algorithm, but also can achieve a higher result. Thus it is a much preferred optimizer for this given task. We also notices, L2 and batch normalization will work better in our case in terms of validation accuracy improvement comparing to dropout. Adding reasonable gradient decay (L2 regularization) results better performance on validation set, according to the experiment, since it effectively restrict weight. Early stopping makes the training much faster. On the contrary, drop out does not seem to be a very effective regularization method using our model. Batch norm guarantees the data distribution of each layer keeps the same, which accelerates the training process by targeting on the fixed aim, and the experiment results proved this theory.

The result of our work need future improvement, in the next stage, we will trying out Convolution Neuron Network on the given task and have more experiment on numbers of hidden layers and the number of hidden layer's neuron.

A Appendix

A.1 Running code in Jupyter Notebook and Google Colab

The code are devided into 7 parts: Import lib, Load data function , Processing data, Activation function class, Hidden layer class, MLP class and Tuning and Running. The only difference to use Jupyter Notebook and Google Colab is that files are usually saved in Google Drive when using Colab rather than local drive. In default, Google Drive is not mounted to Colab. Methods should be performed to connect Google Drive with Colab. On the contrary, data can be directly accessed through local drive file structure using Jupyter Notebook.

A.1.1 Loading Data using Jupyter Notebook

Put the ".npy" file of data and targets in the folder where the code is located then the code will automatically load the numpy data. If you don't want to change the location of the data, you could just change the path of 'load_npData' function in code, below is what 'load_npData' function looks like.

```
# load data
def load_npData(train_data_path='train_data.npy',
                train_label_path='train_label.npy',
                test_data_path='test_data.npy',
                test_label='test_label.npy'):
```

A.1.2 Loading Data using Google Colab

1. The first method is using the "drive.mount" method of "google.colab", which directly mounts personal Google Drive to the running Colab notebook, then the Google Drive file structure can be used just like using local drive.

```
# directly mounting personal Google Drive file structure to a given path
from google.colab import drive
drive.mount('/content/drive')
```

2. The second method is indirect and requires more codes. However, this method only allows programmers to visit only some specific files without deliberately or accidentally modify other files in that file structure.

   (a) Some codes to call for authentication to Google Drive

```
!pip install PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

   (b) Locating the required files in Google Drive, and finding the file id by generating file links. Use file ids to make directly connection between Colab and those files (Actual File ids are replaced by several * marks):

```
train_data_google_file = drive.CreateFile({'id':"**********"})
train_data_google_file.GetContentFile('train_data.npy')

train_label_google_file = drive.CreateFile({'id':"**********"})
train_label_google_file.GetContentFile('train_label.npy')

test_data_google_file = drive.CreateFile({'id':"**********"})
test_data_google_file.GetContentFile('test_data.npy')

test_label_google_file = drive.CreateFile({'id':"**********"})
test_label_google_file.GetContentFile('test_label.npy')
```

   (c) Using common reading commands to read data files:

```
train_data = np.load('train_data.npy')
train_label = np.load('train_label.npy')
test_data = np.load('test_data.npy')
test_label = np.load('test_label.npy')
```

A.1.3 Hyper-parameters

Hyper-parameters are showed in Tuning and Running part, you can adjust the parameter if you like.
Important: the sum of TRAIN_SIZE and VAL_SIZE cannot be greater than 50000 which is the number of all input data.

```
### Tuning Part ###
# dataset size
TRAIN_SIZE = 49000 # traing dataset size
VAL_SIZE = 500 # validation dataset size
TEST_SIZE = test_raw_data.shape[0] # test dataset size
PRINT_INTERVAL=1 # interval for log print
```

```
CAL_VAL=True # show/NOT show the validation loss and accuracy in log print
# iteration related para
LEARNING_RATE = 0.005 # the initial learning rate
LEARNING_RATE_DECAY = 1 # the decay rate of the learning
EPOCHS = 300 # number of epochs to run
BATCH_SIZE=500 # size of batch
ITER_EACH_EPOCH=int(TRAIN_SIZE / BATCH_SIZE) # number of batches to iterate in
    each epoch

EARLY_STOP=True # early stopping

# Optomizers
MOMENTUM=False # momentum turned on/off
ADAM=True # adam turned on/off

# Regularisation
DROP_OUT_RATE=-1 # rate of dropout
PENALTY='l2' # use l2 regularisation
LAMBD=0.3 # lambda for l2

# Batch norm
BATCH_NORM=True # turn on/off batch normalisation
```

After adjusting the path and hyper-parameters, you just need to run all cells and the result and the log of each epoch will show.