Neural Network Training-Shizhi Chen-10307389

Extend the multi-layer perceptron (MLP) exercise on the training of the MNIST character recognition dataset. Evaluate the role of different hyperparameter values, to try to improve the results achieved in the training. Discuss and justify the choice of your hyperparameters and explain the general pattern of results to optimise the MLP training.

1. A description in your own words of the technique in general

Multi-layer perceptron (MLP) is a common Artificial Neural Network algorithm consisting of an input layer, an output layer and one or more hidden layers. Single layer perceptron can only learn linear functions, while multilayer perceptron can also learn nonlinear functions. The following figure is a simple MLP example with one hidden layer.

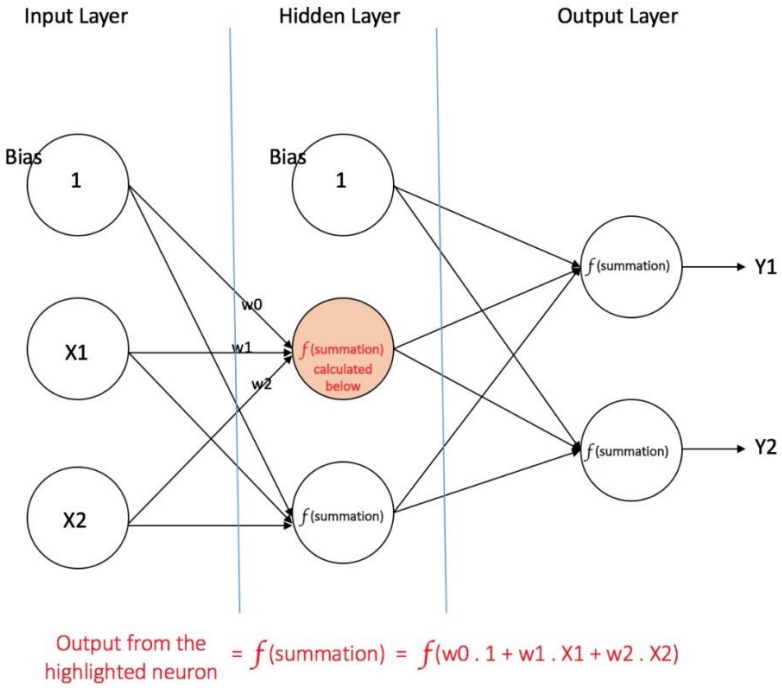


Figure 1 Multi-layer perceptron with one hidden layer

Input layer: The input layer has three nodes. Contains a bias node 1 and two external input nodes X1 and X2. No calculation is done at the input layer, so the output 1, X1 and X2 of the input layer node are passed directly into the hidden layer.

Hidden layer: The hidden layer also has three nodes. The output of the bias node is 1 and the output of the other two nodes of the hidden layer depend on the output of the input layer (X1, X2) and the weight of the connection. Figure 1 shows the calculation of an output in a hidden layer (highlighted, f is the activation function). These outputs are passed to the nodes of the output layer.

Output layer: The output layer has two nodes, receives input from the hidden layer, and performs calculations similar to the hidden layer that is highlighted. The results of these calculations (Y1 and Y2) are the outputs of the multilayer perceptron.

Given a series of features X = (X1, X2, ...) and target Y, a multi-layer perceptron can learn the relationship between features and targets for the purpose of classification or regression.

2. The explanation of the hyperparameters you decide to use and how and why you chose them.

Optimizer: To successfully train an MLP model, it is important to choose an appropriate optimization method. Although the stochastic gradient descent method (SGD) usually works well, more advanced methods like Adam and Adagrad can run faster, especially when training very deep networks. Therefore, we first choose to compare the performance of different optimizers including RMSprop, Adagrad and Adam.

Number of epochs: When a complete dataset passes through the neural network once and returns once, the process is called an epoch. Generally, as the number of epochs increases, the number of updates of weights in the neural network also increases, and the curve becomes good fitting from under-fitting. Due to the multi-layer perceptron can learn faster, so we set the initial epochs to 50, and then try the 200 epochs.

Number of hidden units: Hidden units are also important when training the MLP model. In theory, the more hidden units mean deeper network and can

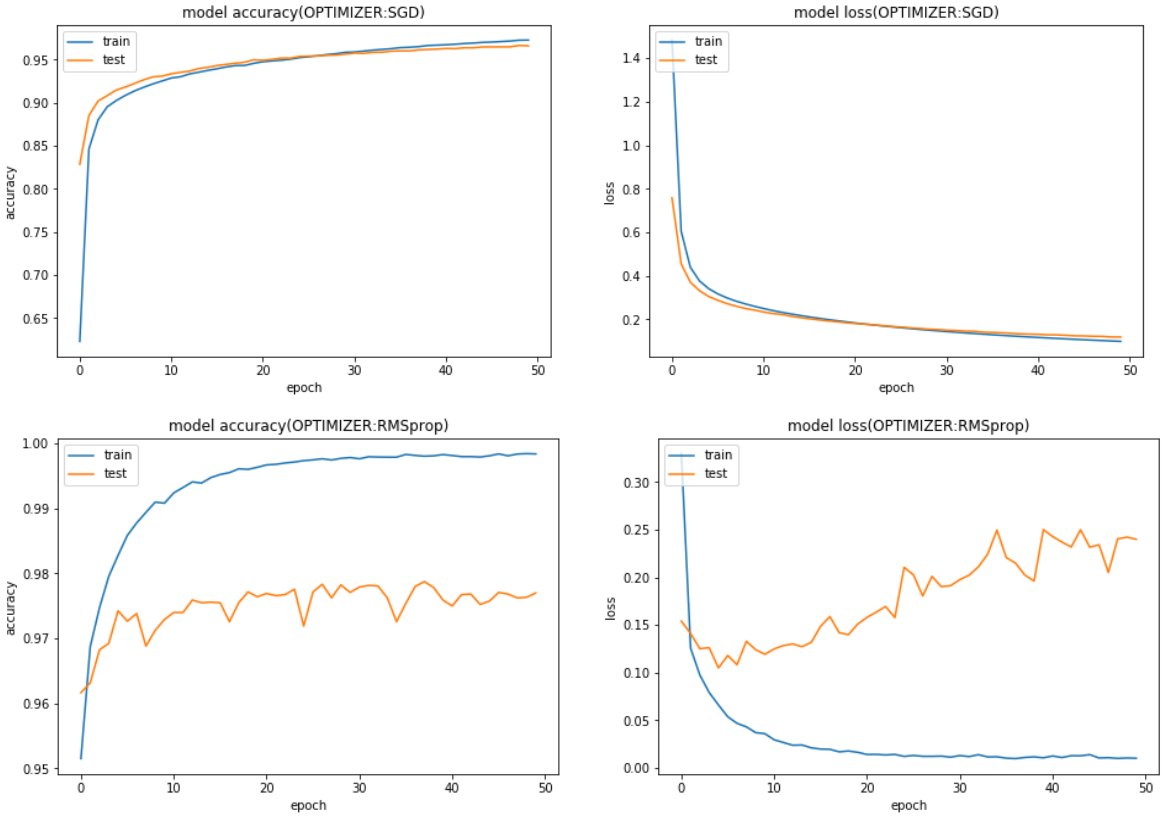
gain a better result. In this report, we initially set the hidden units to 128, and then try the 64 and 256 to see the different effects.

Dropout rates: Dropout refers to the temporary discarding of a part of the neural network unit from the network according to a certain probability during the training of the deep learning network, which is an important hyperparameter to prevent over-fitting and improve the model performance. After finding other optimal parameters, we set the dropout rate to 0.1, 0.2, 0.3, 0.4 to observe the results.

3. Description, interpretation and assessment of success of the results of the hyperparameter testing simulations, including appropriate figures and tables to support the results.

Optimizer: First, use SGD, RMSprop, Adagrad and Adam optimizers to train the model and find the best optimizer. Set other hyperparameters to initial values (N_EPOCH = 50, N_HIDDEN = 128, P_DROPOUT = 0).

The following figures are the performance of these four optimizers:



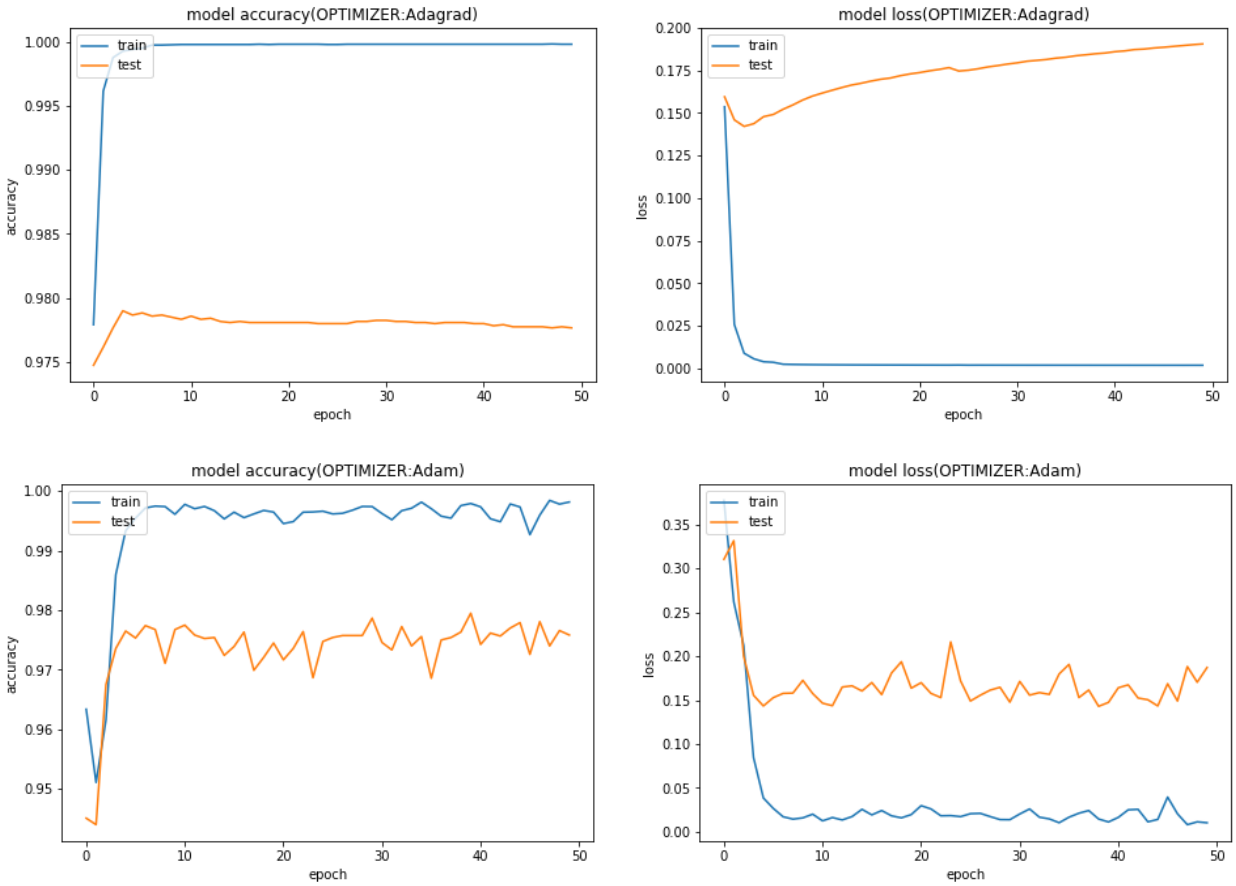


Figure 2 Model performance of SGD, RMSprop, Adagrad and Adam

The test score and accuracy are shown below.

```
OPTIMIZER: SGD
Test score: 0.11557499876283109
Test accuracy: 0.966

OPTIMIZER: RMSprop
Test score: 0.25219955126688853
Test accuracy: 0.9779

OPTIMIZER: Adagrad
Test score: 0.18495396409558598
Test accuracy: 0.9789

OPTIMIZER: Adam
Test score: 0.15868301378935576
Test accuracy: 0.9785
```

From all the results above, we can see that SGD optimizer has the lowest test score which is the most stable optimizer. However, compared with SGD, the other optimizer have higher accuracy. The Adagrad optimizer has the highest accuracy on the test but the test score is much higher than Adam. Adam is the best performing optimizer in terms of overall accuracy and test score. Thus, we use Adam as our optimizer in the next adjustment hyperparameters process.

Epoch: Keep the other parameters unchanged and set the epochs to 200.

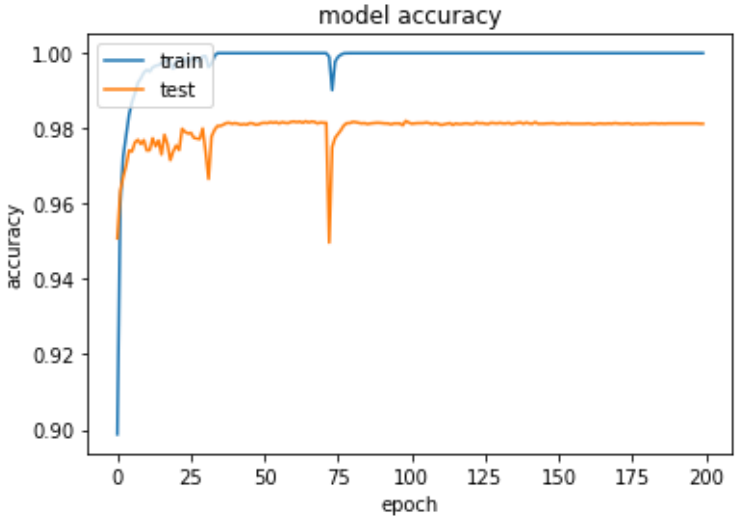


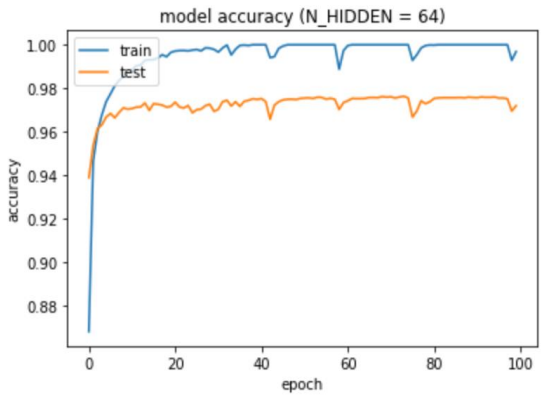
Figure 3 Model accuracy on train and test (200 epochs)

```
Test score: 0.15643983174368747
Test accuracy: 0.9828
```

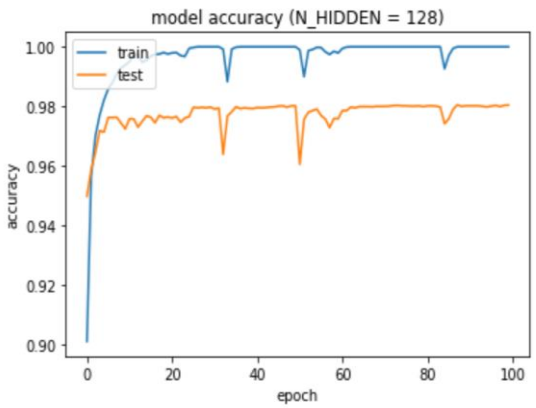
The result shows that with the increase of the epoch, the accuracy on the test and train are both increasing. However, when the number of epochs reaches 100, the whole model has stabilized and continued increasing the number of epochs is just a waste of time. Thus, we set the number of epochs to 100.

Hidden units: Similarly, we are trying to find out the number of hidden units influence on the model performance. Therefore, we set the hyperparameter N_HIDDEN = 64, 128 and 256.

```
Test score: 0.1645955732533675
Test accuracy: 0.9757
```




```
Test score: 0.1275473328612559
Test accuracy: 0.9816
```



```
Test score: 0.14071018937387492
Test accuracy: 0.984
```

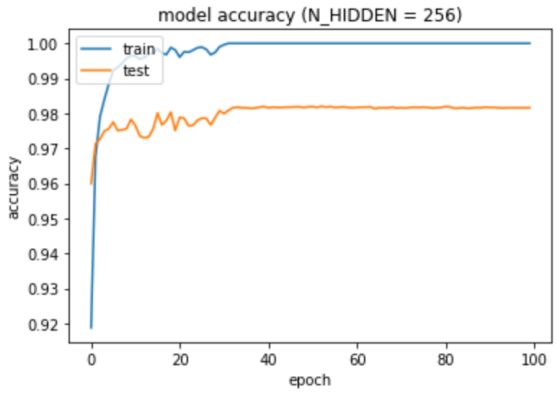


Figure 4 Model accuracy on train and test with hidden units 64,128 and 256

It is obvious that more hidden units have a better performance on the test accuracy. The model with hidden units equal to 256 has the highest test accuracy 0.984. However, the model is a little over-fitting as the test score is increasing when then hidden units increase from 128 to 256. At this point, we already have three optimal hyperparameters, they are optimizer = Adam, the number of epochs = 100 and the number of hidden units = 256.

Dropout rates: Finally, we use the dropout method to solve the problem of partial overfitting. Set the dropout rates from 0.1 to 0.5 (interval is 0.1) and record all the accuracy and loss of train and test.

From the table below we can see that dropout rates have a clear effect on balance the training accuracy and test accuracy, which means that can effectively avoid the overfitting situation.

Table 1 Model performance with dropout rate 0 to 0.5

	Dropout_rate	train_acc	test_acc	train_loss	test_loss
0	0.0	1.000000	0.9834	1.376917e-07	0.128141
1	0.1	0.998313	0.9816	5.595799e-03	0.128096
2	0.2	0.997917	0.9838	6.836420e-03	0.093973
3	0.3	0.996271	0.9846	1.271998e-02	0.090165
4	0.4	0.993146	0.9826	2.228298e-02	0.089981
5	0.5	0.988521	0.9784	3.846304e-02	0.096421

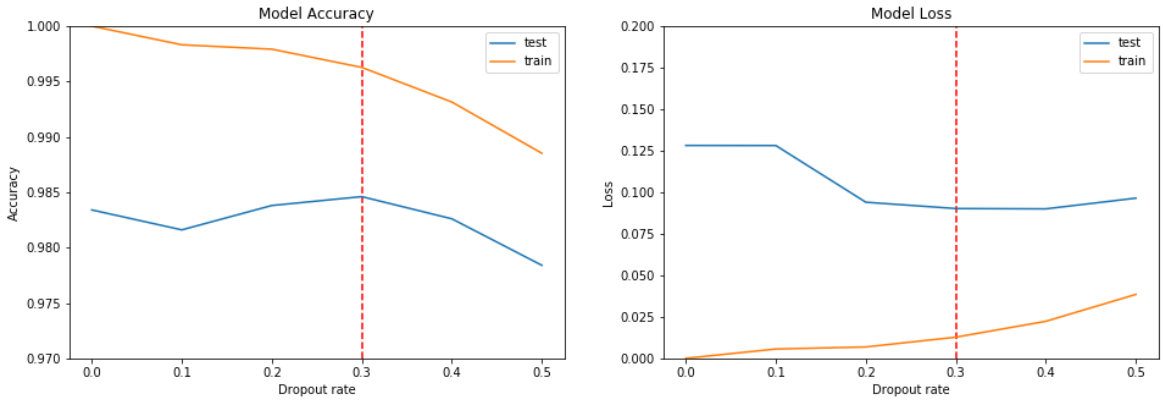


Figure 5 Model accuracy and loss

Figure 5 shows the model accuracy and model loss when the dropout rates increase from 0 to 0.5. It is obvious that when the dropout rate = 0.3 the model obtains the optimal performance on both test and train.

Optimal Hyperparameters

Finally, we find the optimal hyperparameters in this dataset classification task with Optimizer = Adam, Epoch = 100, Hidden units = 256, Dropout rate = 0.3.

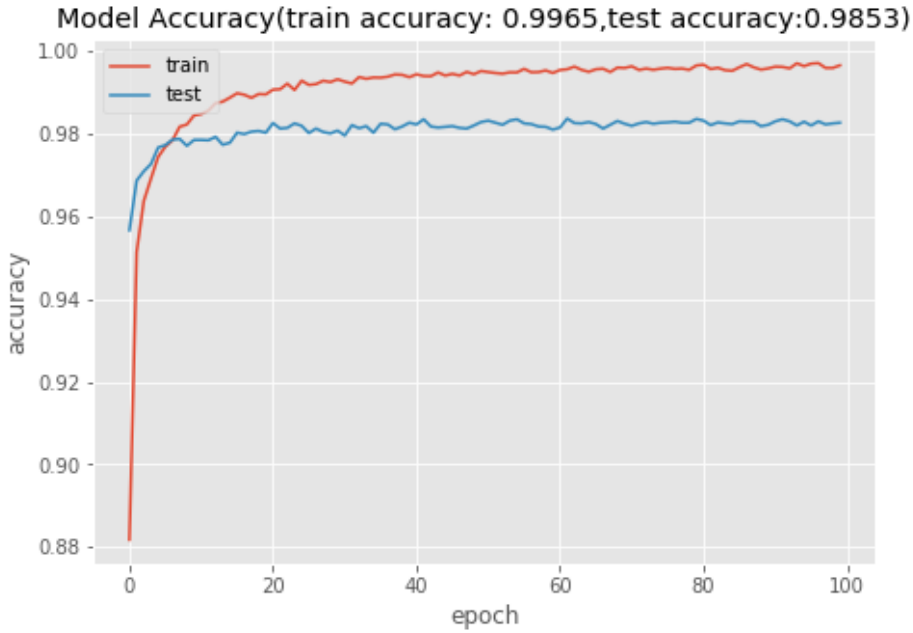


Figure 6 Final Model accuracy on train and test

The final result is shown in Figure 6, this model achieves the 99.65% accuracy on training and 98.53% accuracy on testing with the hyperparameters we chose.

4. Python code used to produce the simulations.

Because the output of the code is very long, the codes following cleared all output, and the main output is already displayed in the previous parts.

Importing the libraries

```
from __future__ import print_function
import numpy as np
import pandas as pd
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD, Adagrad, RMSprop, Adam
from keras.layers.core import Dropout
from keras.utils import np_utils
import matplotlib.pyplot as plt
```

Preparing the MNIST dataset

View the dataset

```
# data: shuffled and split between train and test sets, loading and using the Keras mnist dataset
(input_X_train, output_Y_train), (input_X_test, output_Y_test) = mnist.load_data()

# print the shapes of the input and output data
print("Training data input shape: " , input_X_train.shape)
print("Training data output shape: " , output_Y_train.shape)
print("Test data input shape: " , input_X_test.shape)
print("Test data ouput shape: " , output_Y_test.shape)
```

Visualisation of the numerical vector and plot of a selected image

```
Selected_Image = 2
image = input_X_train[Selected_Image]
#print ("Sample input image: " + str(image))
plt.imshow(image, cmap='gray')
plt.show()
```

prepare the input data

```
#each 2D image consists of 28x28 values/pixels, which need to be reshaped in a vector of 784 pixels
RESHAPED = 784

# use 60000 images for training, 10000 for validation test
input_X_train = input_X_train.reshape(60000, RESHAPED)
input_X_test = input_X_test.reshape(10000, RESHAPED)
input_X_train = input_X_train.astype('float32')
input_X_test = input_X_test.astype('float32')

# normalisation of the pixel values from 0-255 range to 0-1 range
input_X_train /= 255
input_X_test /= 255

print ("Input data ready")
```

Preparing the output labels

Converts the output data into categorical (one-hot encoding) vectors of 0s and 1s

```
N_CLASSES = 10
# convert class vectors to binary class matrices
output_Y_train = np_utils.to_categorical(output_Y_train, N_CLASSES)
output_Y_test = np_utils.to_categorical(output_Y_test, N_CLASSES)

# print the categorical, one-hot output vector for the sample image
label = output_Y_train[Selected_Image]
print ("One-hot-vector: " + str(label))
```

Main training parameters

```
# variables for network and training
VERBOSE = 1
N_CLASSES = 10   # number of classes/categories of digits from 0 to 9, i.e. number of output units
VALIDATION_SPLIT=0.2 # proportion of the dataset used for validation, with remaining .8 for training

#each 2D image consists of 28x28 values/pixels, which need to be reshaped in a vector of 784 pixels
RESHAPED = 784

# random seed number to be used for reproducibility
np.random.seed(1671)
```

```
# Initial hypeparameter
N_EPOCH = 50
BATCH_SIZE = 128
OPTIMIZER = SGD() # Stochastic gradient descent optimiser
N_HIDDEN = 128   # number of hidden units
```

Find the best Optimizer

```
model = Sequential()

OPTIMIZER_list = [SGD(), RMSprop(), Adagrad(), Adam()]
OPTIMIZER_name = ['SGD', 'RMSprop', 'Adagrad', 'Adam']

for i,OPTIMIZER in enumerate(OPTIMIZER_list):

    # Hidden layer 1 with 128 hidden units and ReLu activation function
    model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
    model.add(Activation('relu'))
    # Hidden layer 2 with 128 hidden units and ReLu activation function
    model.add(Dense(N_HIDDEN))
    model.add(Activation('relu'))

    # output layer with 10 units and softmax activation
    model.add(Dense(N_CLASSES))
    model.add(Activation('softmax'))
```

```
    # Summary of the whole model
    #model.summary()

    # model compilation
    model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER, metrics=['accuracy'])

    #train the network
    history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,
                        epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)

    #test the network
    score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
    print("\nOPTIMIZER: %s" % OPTIMIZER_name[i])
    print("Test score:", score[0])
    print("Test accuracy:", score[1])

    #Visualization of the model performance
    fig = plt.figure(figsize=(16,5))
    plt.subplot(121)
    plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
    plt.title('model accuracy(OPTIMIZER:%s)'%OPTIMIZER_name[i])
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')

    plt.subplot(122)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss(OPTIMIZER:%s)'%OPTIMIZER_name[i])
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
```

Find the best number of epochs and hidden units

```
N_EPOCH = 100
N_HIDDEN = 256
OPTIMIZER = Adam()

#N_EPOCH = 200
#N_HIDDEN = 64
#N_HIDDEN = 128

model = Sequential()

# Hidden layer 1 with 128 hidden units and ReLu activation function
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
# Hidden layer 2 with 128 hidden units and ReLu activation function
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))

# output layer with 10 units and softmax activation
model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))

# Summary of the whole model
model.summary()

# model compilation
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER, metrics=['accuracy'])

#train the network
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,
                    epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)

#test the network
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nTest score:", score[0])
print("Test accuracy:", score[1])
```

```
#test the network
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nTest score:", score[0])
print("Test accuracy:", score[1])

# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Adding weight droputs

```
# import the dropout layer type
from keras.layers.core import Dropout

# Probability of weights dropout
P_DROPOUT_list = [0, 0.1, 0.2, 0.3, 0.4, 0.5]

train_acc = []
test_acc = []
train_loss = []
test_loss = []
```

```
for P_DROPOUT in P_DROPOUT_list:

    model = Sequential()
    model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
    model.add(Activation('relu'))
    model.add(Dropout(P_DROPOUT))
    model.add(Dense(N_HIDDEN))
    model.add(Activation('relu'))
    model.add(Dropout(P_DROPOUT))
    model.add(Dense(N_CLASSES))
    model.add(Activation('softmax'))

    # model compilation
    model.summary()

    # model compilation
    model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER, metrics=['accuracy'])

    #train the network
    history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,
                        epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)

    #test the network
    score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
    print("Dropout rate: %.2f" % P_DROPOUT)
    print("Test score:", score[0])
    print("Test accuracy:", score[1])

    #record the train accuracy/loss and test accuracy/loss
    train_acc.append(history.history['acc'][-1])
    test_acc.append(score[1])

    train_loss.append(history.history['loss'][-1])
    test_loss.append(score[0])
```

```
df_result = pd.DataFrame({'Dropout_rate':P_DROPOUT_list,
                          'train_acc':train_acc, 'test_acc':test_acc,
                          'train_loss':train_loss,'test_loss':test_loss})
df_result
```

```
fig = plt.figure(figsize=(16,5))
plt.subplot(121)
plt.ylim(0.97,1)
plt.plot(df_result['Dropout_rate'],df_result['test_acc'],label = 'test')
plt.plot(df_result['Dropout_rate'],df_result['train_acc'],label = 'train')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Dropout rate')
plt.vlines(0.3, 0.97, 1, colors = "r", linestyles = "dashed")
plt.legend()

plt.subplot(122)
plt.ylim(0,0.2)
plt.plot(df_result['Dropout_rate'],df_result['test_loss'],label = 'test')
plt.plot(df_result['Dropout_rate'],df_result['train_loss'],label = 'train')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Dropout rate')
plt.vlines(0.3, 0, 1, colors = "r", linestyles = "dashed")
plt.legend()
plt.show()
```

Final optimal hyperparameters model

N_EPOCH = 100 BATCH_SIZE = 128 OPTIMIZER = Adam() N_HIDDEN = 256

```
# hypeparameter
N_EPOCH = 100
BATCH_SIZE = 128
OPTIMIZER = Adam()
N_HIDDEN = 256
P_DROPOUT = 0.3

model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dropout(P_DROPOUT))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(P_DROPOUT))
model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))

# model compilation
model.summary()
# model compilation
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER, metrics=['accuracy'])

#train the network
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,
                    epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)

#test the network
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nFinal Model")
print("Test score:", score[0])
print("Test accuracy:", score[1])
```

```
#Visualization of the model performance
fig = plt.figure(figsize=(16,5))
plt.subplot(121)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model Accuracy(train accuracy: %.4f,test accuracy:%.4f)'%(history.history['acc'][-1],score[1]))
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')

plt.subplot(122)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
```

An interesting multi-layer perceptron 3D visualization created by Adam Harley (http://scs.ryerson.ca/~aharley/vis/fc/).