Deep Learning Midterm Number 7

Experiments on notMNIST Dataset

Indra Imanuel Gunawan - 20195118

This report will explain the result of the CNN experiment for notMNIST data set, primarily using TensorFlow and Keras. The code for this experiment can be found in the github link that is attached within the email. This report will be divided into three sections: (1) Preparation, (2) Building and Training the Model, (3) Evaluation.

I. Preparation

In this step, we prepare the necessary libraries, dataset, normalize the input data, and split the dataset into test, training, and validation data set. The process of importing and loading the necessary libraries and the notMNIST data set can be seen in figure 1.

```
#importing necessary libraries
import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, LeakyReLU, ReLU
from keras.optimizers import Adam, Adagrad

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from os.path import dirname, join as pjoin
import scipy.io as sio

#Import the notMNIST dataset
mat_fname = "C:/Users/USER/Documents/PythonFiles/DL_Midterm_No7/data/notmnist/notMNIST_small.mat"
mat_contents = sio.loadmat(mat_fname)

X = mat_contents["images"].transpose()
Y = mat_contents["labels"]

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=42)
```

```
Using TensorFlow backend.
```

Figure 1 Importing necessary libraries and notMNIST dataset

We are using Keras and tensorflow primarily to run this experiment, but we also use numpy to calculate mean and standard deviation, sklearn to split training and validation set, and matplot lib to plot our result graph. The notMNIST dataset is provided in matlab file format, so we read it using scipy.io.loadmat() function.

After we have loaded the data, we need to normalize the value of x_test and x_train. This can be done by using **Mean Subtraction** and **Normalization**, as can be seen in Figure 2.

```
#Mean subtraction and normalization on the input data

mean_train = np.mean(x_train)
std_train = np.std(x_train)

mean_test = np.mean(x_test)
std_test = np.std(x_test)

x_train_norm = (x_train - mean_train)/std_train
x_test_norm = (x_test - mean_test)/std_test
```

Figure 2 Mean subtraction and normalization on the input data

We searched the mean and standard deviation of both the x_train and x_test input data. After that, we subtract the x_train with the mean of the x_train then divide it by the standard deviation of the x_train. The same thing goes with the x_test. This will ensure the data to range generally from -1 to 1 instead of 0 to 255. If we didn't normalize it and leave the value from 0 to 255, it will cause some problem because the range of the value is too big.

We can also check the images in the dataset visually. The visualization of the images can be seen in figure 3.

```
#Checking the first 25 images
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train_norm[i], cmap=plt.cm.binary)
plt.show()
```

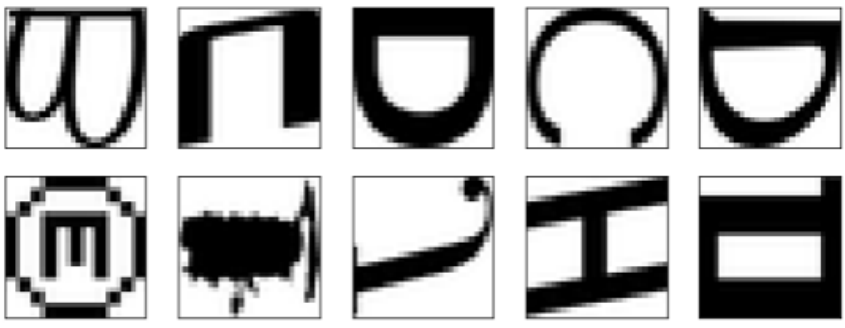

Figure 3 Visualtization of the notMNIST dataset

After we normalize our data, we need to split the training data into training and validation set. This is done with the help of "train_test_split" from sklearn. The ratio is 80% training data and 20% validation data. We can actually just use the test data instead of validation data. After that, we need to reshape our data to a shape of (28, 28, 1). The code for this can be seen in figure 4

```
#Splitting the training data to train and verification data
x_train_norm, x_validate, y_train, y_validate = train_test_split(
    x_train_norm, y_train, test_size=0.2, random_state=12345,
)

#Reshaping the data
img_rows = 28
img_cols = 28
batch_size = 512
img_shape = (img_rows, img_cols, 1)

x_train_norm = x_train_norm.reshape(x_train_norm.shape[0], *img_shape)
x_test_norm = x_test_norm.reshape(x_test_norm.shape[0], *img_shape)
x_validate = x_validate.reshape(x_validate.shape[0], *img_shape)
```

Figure 4 Splitting the training data to training and verification data

II. Building and Training the Model

For this experiment, we use five models, each with their own characteristics. The code for creating these models can be seen in Figure 5. Here's a bit rundown about each of the model:

1. **Xavier 2 Layer Model**
   This model is initialized using the Xavier initializer. It has 2 convolutional layer, followed with a fully connected layer at the end, with each convolutional layer is followed by MaxPooling and Dropout layer. The activation function is using ReLU.
2. **He 2 Layer Model**
   This model is initialized using the He initializer. Its layer structure is the same with the Xavier 2 Layer Model.
3. **He 3 Layer Model**
   This model is initialized using the He initializer. Its layer structure is the same with He 2 Layer Mode, except there is one more convolutional layer, making the structure has 3 convolutional layer instead of 2.
4. **Xavier 2 Layer with LeakyReLU model**
   This model is initialized using the Xavier initializer. Its structure is the same with the Xavier 2 Layer Model, except for the activation function, instead of ReLU, it uses LeakyReLU.

5. **He 2 Layer with L2**

This model is initialized using the he initializer. Its structure is the same with he 2 Layer model, except that instead of dropout layer it uses L2 as the regularizer.

```
#Building the model

name = 'Xavier_2Layer'
modelXavier2L = Sequential([
    Conv2D(32, kernel_size=3, activation='relu', input_shape=img_shape,
           kernel_initializer='glorot_normal', name='Conv2D_Layer1'),
    MaxPooling2D(pool_size=2, name='MaxPool'),
    Dropout(0.2, name='Dropout-1'),
    Conv2D(64, kernel_size=3, activation='relu', name='Conv2D_Layer2'),
    Dropout(0.3, name='Dropout-2'),
    Flatten(name='flatten'),
    Dense(64, activation='relu', name='Dense'),
    Dense(10, activation='softmax', name='Output')
], name=name)

name = 'He_2Layer'
modelHe2L = Sequential([
    Conv2D(32, kernel_size=3, activation='relu', input_shape=img_shape,
           kernel_initializer='he_normal', name='Conv2D_Layer1'),
    MaxPooling2D(pool_size=2, name='MaxPool'),
    Dropout(0.2, name='Dropout-1'),
    Conv2D(64, kernel_size=3, activation='relu', name='Conv2D_Layer2'),
    Dropout(0.3, name='Dropout-2'),
    Flatten(name='flatten'),
    Dense(64, activation='relu', name='Dense'),
    Dense(10, activation='softmax', name='Output')
], name=name)

name='He_3Layer'
modelHe3L = Sequential([
    Conv2D(32, kernel_size=3, activation='relu', input_shape=img_shape,
           kernel_initializer='he_normal', name='Conv2D_Layer1'),
    MaxPooling2D(pool_size=2, name='MaxPool'),
    Dropout(0.25, name='Dropout-1'),
    Conv2D(64, kernel_size=3, activation='relu', name='Conv2D_Layer2'),
    Dropout(0.25, name='Dropout-2'),
    Conv2D(128, kernel_size=3, activation='relu', name='Conv2D_Layer3'),
    Dropout(0.4, name='Dropout-3'),
    Flatten(name='flatten'),
    Dense(128, activation='relu', name='Dense'),
    Dropout(0.4, name='Dropout'),
    Dense(10, activation='softmax', name='Output')
], name=name)
```

Figure 5 Creating the model

```
name = 'Xavier_2Layer_LeakyRELU'
modelXavier2LLeakyRELU = Sequential([
    Conv2D(32, kernel_size=3, input_shape=img_shape, kernel_initializer='glorot_normal', name='Conv2D_Layer1'),
    LeakyReLU(),
    MaxPooling2D(pool_size=2, name='MaxPool'),
    Dropout(0.2, name='Dropout-1'),
    Conv2D(64, kernel_size=3, name='Conv2D_Layer2'),
    LeakyReLU(),
    Dropout(0.3, name='Dropout-2'),
    Flatten(name='flatten'),
    Dense(64, name='Dense'),
    LeakyReLU(),
    Dense(10, activation='softmax', name='Output')
], name=name)

name = 'He_2Layer_L2'
modelHe2LL2 = Sequential([
    Conv2D(32, kernel_size=3, input_shape=img_shape, kernel_initializer='he_normal',
           activity_regularizer=tf.keras.regularizers.l2(10e-3), name='Conv2D_Layer1'),
    ReLU(),
    MaxPooling2D(pool_size=2, name='MaxPool'),
    Conv2D(64, kernel_size=3, activity_regularizer=tf.keras.regularizers.l2(10e-3), name='Conv2D_Layer2'),
    ReLU(),
    Flatten(name='flatten'),
    Dense(64, activation='relu', name='Dense'),
    Dense(10, activation='softmax', name='Output')
], name=name)

cnn_models = [modelXavier2L, modelHe2L, modelHe3L, modelXavier2LLeakyRELU, modelHe2LL2]
```

Figure 6 Creating the model (2)

Then, we train each of our models for 10 epochs, using the sparse categorical cross entropy as the loss function and Adam optimizer. We will also run a second experiment in which we run it using the Adagrad optimizer. The process of this training can be seen in figure 6, while the result of the training can be seen in figure 7 in the form of graph.

```
# train the models and save results to a dict

history_dict = {}

for model in cnn_models:
    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=Adam(),
        metrics=['accuracy']
    )

    history = model.fit(
        x_train_norm, y_train,
        batch_size=batch_size,
        epochs=10, verbose=1,
        validation_data=(x_validate, y_validate)
    )

    history_dict[model.name] = history
```

Figure 6 Training the model

```
# plot the accuracy and loss

fig, (ax1, ax2) = plt.subplots(2, figsize=(8, 6))

for history in history_dict:
    val_acc = history_dict[history].history['val_accuracy']
    val_loss = history_dict[history].history['val_loss']
    ax1.plot(val_acc, label=history)
    ax2.plot(val_loss, label=history)

ax1.set_ylabel('validation accuracy')
ax2.set_ylabel('validation loss')
ax2.set_xlabel('epochs')
ax1.legend()
ax2.legend()
plt.show()
```

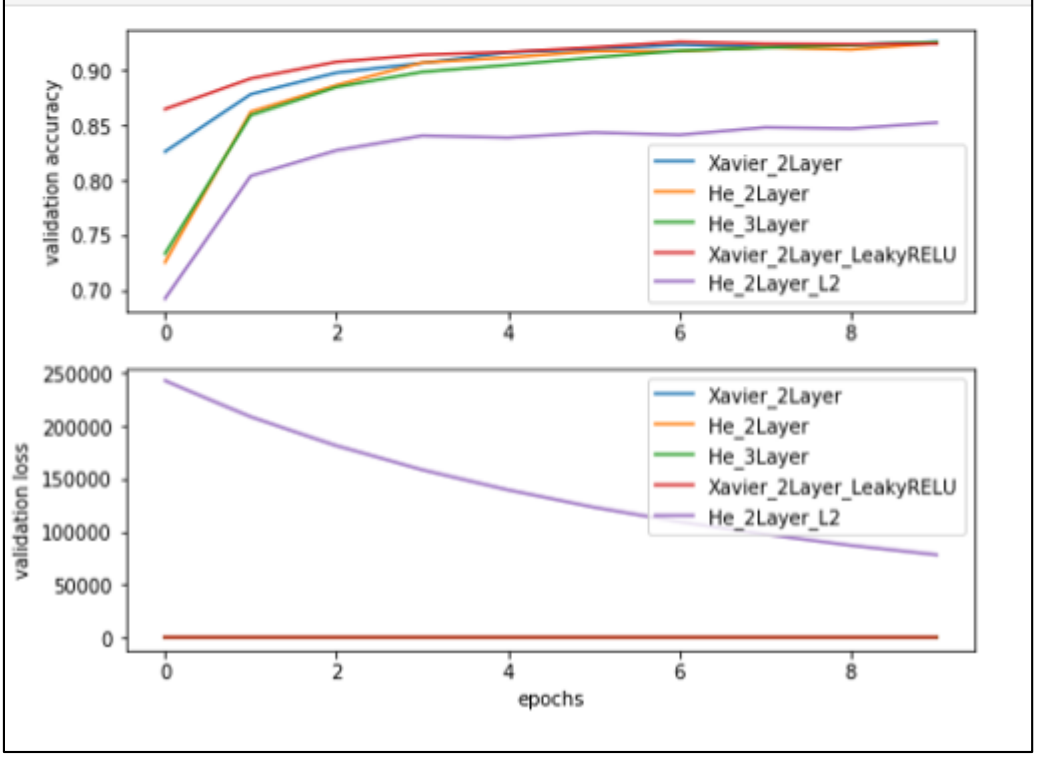

Figure 7 Model training result (Adam Optimizer)

As we can see from figure 7 graph, the model actually performs more or less the same way with exception to the He 2 Layer with L2 model which performs worse compared with the other models. Above are the result graphs if we use the Adam optimizer. For Adagrad optimizer, the graph results can be seen in figure 8. We will explore this in more detail in the evaluation section.

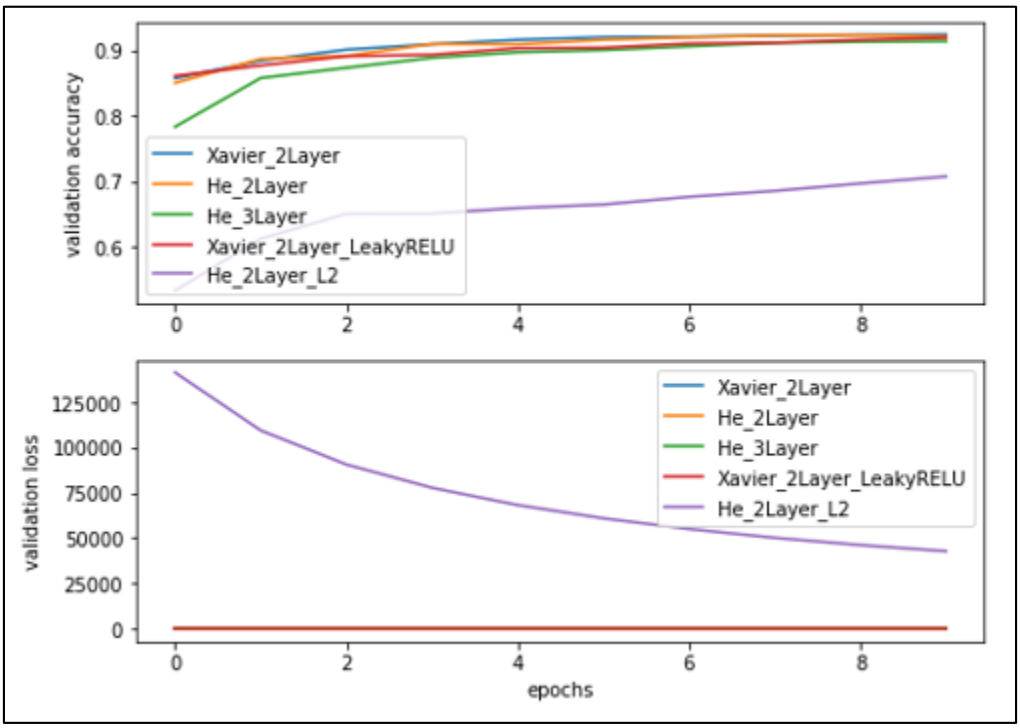

Figure 8 Model training result (Adagrad Optimizer)

III. Evaluation

In this section we will evaluate and compare each model performance using table for better visualization. The accuracy and loss are from the validation set.

A. Xavier vs He Initialization

First, we will compare between the Xavier 2 Layer Model with the He 2 Layer Model. Both of this model have the same configuration, the difference is only in the initializer, one is with Xavier and the other one is with He. The comparison table can be seen in Table 1.

Table 1: Xavier 2 Layer vs He 2 Layer

	Xavier 2 Layer	He 2 Layer
Accuracy	0.9260	0.9249
Loss	0.2588	0.2788

From Table 1, we can see that the performance from both models is actually almost the same. But the He 2 Layer has a slightly better performance. This is due to the network using ReLU activation. He initialization works better with ReLU activation, while Xavier works better with sigmoid activation.

B. He 2 Layer vs He 3 Layer

This time, we want to see whether the number of layer in the network will affects performance or not. We tried comparing between the He model with 2 layers with the He model with 3 layers. The result of this experiment can be seen in table 2.

Table 2: He 2 Layer vs He 3 Layer

	He 2 Layer	He 3 Layer
Accuracy	0.9249	0.9256
Loss	0.2788	0.2576

From Table 2, it can be seen that the model with 2 layers works better rather than the one having 3 layers. This happen because adding too much layer may result in overfitting the data. That's why the performance for the 3 layered model is worse than the 2 layered model.

C. Xavier 2 Layer with ReLU vs Xavier 2 Layer with Leaky ReLU

We compare two of the same model, just with different activation function. We'll see the difference in performance between Xavier 2 Layer with ReLU activation, with Xavier 2 Layer with Leaky ReLU activation. The result can be seen in the Table 3.

Table 3: Xavier 2 Layer with ReLU vs Xavier 2 Layer with Leaky ReLU

	Xavier 2 Layer with ReLU	Xavier 2 Layer with Leaky ReLU
Accuracy	0.9260	0.9636
Loss	0.2588	0.1266

As we can see from Table 3, the model with Leaky ReLU performs slightly better than the one with regular ReLU. This may happens due to negative number existing in the input data (x_train). Our x_train range from -1 to 1, so some of the numbers are negative numbers. With ReLU, the derivative of the positive part is 1, and 0 in the negative part. This can cause in some dead ReLU, while in Leaky ReLU, the derivative of the positive part is 1, and a small fraction in the negative part. This nature of Leaky ReLU and the fact that we have some negative numbers in our input supports the fact that the performance of the model with Leaky ReLU better.

D. He 2 Layer with Dropout vs He 2 Layer with L2

We now compare He 2 layer with dropout with He 2 layer with L3. The result can be seen in Table 4

Table 4: He 2 Layer with Dropout vs He 2 Layer with L2

	He 2 Layer with Dropout	He 2 Layer with L2
Accuracy	0.9249	0.8523
Loss	0.2788	78153.2798

As we can see from Table 4, the result of the model with dropout regularization is much better compared with the model with L2.

E. Adam Optimizer vs Adagrad Optimizer

We also run all the model with Adagrad optimizer this time. Table 5 shows all the model running on the Adam optimizer, while Table 6 shows all the models running on the Adagrad optimizer.

Table 5 : All the models running on Adam Optimizer

	Xavier 2 Layer	He 2 Layer	He 3 Layer	Xavier 2 Layer with Leaky ReLU	He 2 Layer with L2
Accuracy	0.9260	0.9249	0.9256	0.9242	0.8523
Loss	0.2588	0.2788	0.2576	0.2704	78153.2798

Table 6 : All the models running on Adagrad Optimizer

	Xavier 2 Layer	He 2 Layer	He 3 Layer	Xavier 2 Layer with Leaky ReLU	He 2 Layer with L2
Accuracy	0.9242	0.9213	0.9138	0.9192	0.7074
Loss	0.2592	0.2735	0.2883	0.2880	42723.0249

As we can see from Table 5 and 6, Adam optimizer performs better than Adagrad optimizer.