Homework 5: Neural Networks for Classifying Fashion MNIST

Minghu Zhou

March 10, 2020

Abstract

In this report we use Neural Network to classify clothes in Fashion MNIST dataset. We will explore a fully-connected neural network and a convolutional neutral network to see their performances.

1. Introduction and Overview

Human brains are consisted of billions of neurons, each of which connected to others through dendrites. One neuron takes in information and transfers it to other neurons. This network can be complicated due to the large number of neurons inside human brain. Human can thus make reactions thanks to these neutrons.

Neural Network's development is based on the concept above. It takes in a big training dataset and learns the pattern of the data through layers and then it will be able to classify new data. In this report we have a dataset called Fashion MNIST, it has images of 10 different classes of fashion items. Each of these images are labeled from 0 to 9. We want to build a fully-connected neural network and a convolutional neutral network that we think perform well. After that, we check the accuracy of our final model on the test data.

2. Theoretical Background

2.1 Fully-connected Neural Network

A fully-connected Neural Network include layers of neurons. Each neuron in a layer connects to all the neurons in the next layer . We have an input layer that takes in the data and an output layer that gives the classification results. The middle layers are mapped from the previous layer by some weight matrices and activation functions.

In particular, if we have an input layer $\overrightarrow{x_1}$, a middle layer $\overrightarrow{x_2}$, and an output layer $\vec{y}$, then the relationship between them are:

$$\overrightarrow{x_2} = \sigma(A_1\overrightarrow{x_1} + \overrightarrow{b_1}) \quad (2.1)$$

$$\vec{y} = \sigma(A_2\overrightarrow{x_2} + \overrightarrow{b_2}) \quad (2.2)$$

where A_1 , A_2 are weight matrices, $\overrightarrow{b_1}$, $\overrightarrow{b_2}$ correspond to bias neurons and σ represents activation function.

2.2 Convolutional Neutral Network(CNN)

Convolutional Layers use a small window that slides across the entire layer and transfer the data into a new node through a given activation function.

A CNN performs well on classifying images. It contains convolutional layers of feature maps. The general idea is that we use a small window to slide across the image, then we apply a filter kernel to the pixel values in that window and render a new node in our feature map.

We also use the pooling layers along with convolutional layers to subsample our image or feature maps so computation is reduced. Pooling finds the max or the average of nodes in the window and makes it a new node in the feature map in the next layer.

2.3 Activation function (model)

An activation function takes in an input to a neuron and produce an output going to the next layer. We will use *ReLU* and *tanh* as our activation function in the middle layers. We use a $softmax$ activation function in the final layer which gives the probability of the input belonging to all the groups.

2.4 Loss function

A loss function calculates the difference between the actual value and the predicted value. It represents the accuracy of the model during training and we want to minimize this function. In our experiments, we chose the *sparse_categorical_crossentropy* loss function. This function has good performance on multi-class classification.

The cross entropy loss function is the following:

$$L = -\frac{1}{N}\sum_{j=1}^{N} ln(p_j) + \lambda\|\vec{w}\|_2^2 \qquad (2.3)$$

where N is the sample size, p_j is the probability of the actual group in the output. $\lambda\|\vec{w}\|_2^2$ is a regularization term to avoid overfitting where λ is a constant and $\vec{w}$ is the weight matrix.

2.4 Gradient Descent

Now we want to find the minimum of our loss function L with respect to $\vec{w}$. We will use an iterative method called gradient descent to approach the solution. We start from a guess point on the surface, then descend a distance along the gradient vector of that point to reach a new point. We repeat this process until the point is close enough to the bottom of the surface. The relationship between one point $\overrightarrow{w_n}$ and the next point $\overrightarrow{w_{n+1}}$ is

the following:

$$\overrightarrow{w_{n+1}} = \overrightarrow{w_n} - \delta \nabla L(\overrightarrow{w_n}) \quad (2.4)$$

Where δ represents the learning rate, which decides how much distance the point will travel in one iteration. Once we found that $\nabla L(\overrightarrow{w_n})$ is very close to 0, this $\overrightarrow{w_n}$ is an ideal parameter in our model.

3. Algorithms Implementation and Development

First, we need to load in the data. Use the following command we load in our data:

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

There are 60,000 training images in the array X_train_full and 10,000 test images in the array X_test, each of which is 28 by 28. The labels are contained in the vectors y_train_full and y_test. The values in these vectors are numbers from 0 to 9 which correspond to the 10 classes.

Then we need to preprocess the data. We remove 5,000 images from our training data to use as validation data. So we end up with 55,000 training examples in an array X_train and 55,000 labels in a vector y_train; 5,000 validation examples in an array X_valid and 5,000 labels in a vector y_valid. The numbers in the arrays X_train, X_valid, and X_test are integers from 0 to 255. We convert them to floating point numbers between 0 and 1 by dividing each by 255.0.

Part 1

We first design the NN architecture. I decide to use a 4 layer Neural Network because it has good performance. We flatten the data in the first layer, so one image data which is 28 by 28 will flatten to 784. The second layer is a dense layer with 500 neurons and a $Relu$ activation function. The third layer is the same as the second except for it has 100 neurons. The final layer is a dense layer with 10 neurons because there are 10 categories; it uses a $softmax$ activation function. Note that we add a regularization factor to the dense layers to avoid possible overfitting caused by too many parameters.

Next, we need to compile our model. There are three factors. The first is the loss function. We will use $sparse\_categorical\_crossentropy$ loss function which we mentioned in section 2. The second is the optimizer, it determines how our model will be updated. We use $Adam$ with a learning rate of 0.001 as our optimizer. The last is an accuracy metric which measures the percent of correctly classified images in training and validating process.

Then we can train our model. We give our model the training data and the validation data. We want our network to be trained 5 times so the epochs is set to be 5. We assign the whole training process to a parameter $history$ so we can see the improvement of our model through the epochs. We can also plot how accuracy and loss of both training and validation change afterwards.

Last, we can evaluate our model. We give our model the test data and get the loss and the accuracy. We can visualize the performance by looking at the confusion matrix. Each column of the confusion matrix represents the actual label and each row represents the classified label so the elements along the diagonal are numbers correctly classified and elements off the diagonal are numbers incorrectly classified.

Part 2

The algorithm is almost the same as the first part. There are mainly two differences.

First, we need to add a new dimension to our data. We do this by using *np.newaxis.* Second, the NN architecture is different. In my design, the first layer is a convolutional layer, with 16 filters each of which has a size 5 by 5. In this layer, we want the input image and output feature map have the same size so we add a padding of zeros to periphery to the image. The second layer is a max pooling layer which has a window size 2 by 2. The third layer is a convolutional layer with 32 5 by 5 filters. The next max pooling layer has the same size as the second. The next convolutional layer has 120 5 by 5 filters. The following layers are fully connected dense layers. These layers all have a *tanh* activation function except the final output layer uses a *softmax* activation function. Meanwhile, I only uses 3 epochs in my training and the accuracy is already good.

4. Computational Results

Part 1:

To find a good model, I mainly change the number of layers and the width of layers. When there are more than 4 layers, I find the training accuracy even drops so I think 4 layers might be a good choice. When there are fewer than 50 neurons in a layer, the accuracy can be very low. On the other hand, when there are over 1000 neurons in a layer, the accuracy of test data barely changes but it causes the problem of overfitting, which means the validation accuracy and the test accuracy is much lower than the training accuracy. Thus, I decide to use two middle layers, one with 500 neurons and another with 100 neurons. The training history is shown below:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/5
55000/55000 [==============================] - 35s 633us/sample - loss:
 0.4898 - accuracy: 0.8274 - val_loss: 0.3698 - val_accuracy: 0.8708
Epoch 2/5
55000/55000 [==============================] - 33s 606us/sample - loss:
 0.3786 - accuracy: 0.8655 - val_loss: 0.3659 - val_accuracy: 0.8726
Epoch 3/5
55000/55000 [==============================] - 34s 626us/sample - loss:
 0.3431 - accuracy: 0.8787 - val_loss: 0.3274 - val_accuracy: 0.8880
Epoch 4/5
55000/55000 [==============================] - 35s 627us/sample - loss:
 0.3250 - accuracy: 0.8848 - val_loss: 0.3467 - val_accuracy: 0.8794
Epoch 5/5
55000/55000 [==============================] - 34s 615us/sample - loss:
 0.3058 - accuracy: 0.8934 - val_loss: 0.3103 - val_accuracy: 0.8952
```

We can see both the training accuracy and the validation accuracy increases through epochs and they are almost the same in the end. Thus, I think this is a good model.

The accuracy of this model on the test data is 0.8796. The confusion matrix for the test data is shown below:

```
[[772   1  16  14   6   1 178   0  12   0]
 [  2 970   0  18   4   0   5   0   1   0]
 [ 11   0 752  13 123   0 101   0   0   0]
 [ 22   5  10 875  39   0  44   0   5   0]
 [  1   0  76  28 822   0  72   0   1   0]
 [  0   0   0   1   0 965   0  22   0  12]
 [ 76   0  70  25  70   0 750   0   9   0]
 [  0   0   0   0   0  19   0 969   0  12]
 [  3   0   2   3   4   3   3   3 979   0]
 [  0   0   0   0   0   4   1  53   0 942]]
```

Part 2:

In the CNN model I mainly change the number of filters in convolutional layers and the kernel sizes. When we have more filters in each convolutional layer, we see higher accuracy. Meanwhile, when each filter has a smaller kernel size, there is also a higher accuracy. However, while increasing the number of filters and decreasing the kernel size of them, we spend more time in training in each epoch. Thus, there is a tradeoff between accuracy and training time in a CNN model.

Here is the training history:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/3
55000/55000 [==============================] - 73s 1ms/sample - loss:
0.4506 - accuracy: 0.8415 - val_loss: 0.3220 - val_accuracy: 0.8888
Epoch 2/3
55000/55000 [==============================] - 71s 1ms/sample - loss:
0.3128 - accuracy: 0.8889 - val_loss: 0.2991 - val_accuracy: 0.8916
Epoch 3/3
55000/55000 [==============================] - 71s 1ms/sample - loss:
0.2693 - accuracy: 0.9054 - val_loss: 0.2693 - val_accuracy: 0.9080
```

Still, both the training accuracy and the validation accuracy increases through epochs and they are almost the same in the end. Thus, I think it is a good model.

The accuracy of this model on the test data is 0.8988. The confusion matrix for the test data is shown below:

```
[[804   1  12  19   3   1 152   0   8   0]
 [  0 973   1  17   3   0   4   0   2   0]
 [ 15   1 855  10  50   0  68   0   1   0]
 [  7   3   7 919  21   0  40   0   3   0]
 [  0   0  79  32 792   0  96   0   1   0]
 [  0   0   0   0   0 967   1  26   0   6]
 [ 76   0  60  25  48   0 783   0   8   0]
 [  0   0   0   0   0   9   0 974   0  17]
 [  2   1   1   6   3   1   2   5 978   1]
 [  0   0   0   0   0  14   0  43   0 943]]
```

The column of the confusion matrix means the actual label while the row means the predicted label. The element in ith row and jth row represents the number of images labeled j and classified as i.

Compare these two matrices, we found many images labeled 0 are easily misclassified as 6 and vice versa. Look up the table we find that 0 corresponds to T-shirt while 6 corresponds to shirt. These images can be very similar so they will be easily misclassified. Overall, the CNN model has a better performance in test accuracy than the fully connected NN.

5. Summary and Conclusions

In this report I explore the performance of a fully-connected neural network and a convolutional neural network on classifying fashion clothes. After customizing our models, we see an accuracy of around 90% on test data for both models. Like human, our models make mistakes on images which are similar to those in other groups. The CNN model generally has better performance, but it takes more time to train in each epoch. Thus, we see there is a tradeoff between training time and test accuracy.

Appendix A.
Python functions and brief implementation explanation

keras.model.compile - Configures the model for training.
keras.model.fit - Trains the model for a fixed number of epochs (iterations on a dataset).
keras.model.evaluate - Returns the loss value & metrics values for the model in test mode.
keras.model.predict - Generates output predictions for the input samples.

Appendix B.
Python codes

Part 1

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix

fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

plt.figure()
for k in range(9):
    plt.subplot(3,3,k+1)
    plt.imshow(X_train_full[k], cmap="gray")
    plt.axis('off')
plt.show()
```

```
X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0
y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(0.00001))

model = tf.keras.models.Sequential([
   tf.keras.layers.Flatten(input_shape=[28, 28]),tra
   my_dense_layer(500),
   my_dense_layer(100),
   my_dense_layer(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
            optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
            metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=5, validation_data=(X_valid,y_valid))

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)
```

Part 1

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix

fashion_mnist = tf.keras.datasets.fashion_mnist
```

```
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
]

X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0

y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]

from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="tanh",
kernel_regularizer=tf.keras.regularizers.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="tanh", padding="valid")

model = tf.keras.models.Sequential([
   my_conv_layer(16,5,padding="same",input_shape=[28,28,1]),
   tf.keras.layers.MaxPooling2D(2),
   my_conv_layer(32,5),
   tf.keras.layers.MaxPooling2D(2),

   my_conv_layer(120,5),
   tf.keras.layers.Flatten(),
   my_dense_layer(84),
   my_dense_layer(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
           optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
           metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=3, validation_data=(X_valid,y_valid))

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)
```