Hw5 : Neural Networks

Author : Richy Yu

Abstract

In this paper, I train two neural networks to classify the images in the Fashion-MNIST data set. The programming language used is python.

Introduction and Overview:

The purpose of this paper is to train a neural network that classifies the images in the Fashion-MINIST data set. There are 60000 training images in the array x_train_full and 10000 test images in the array x_test each of which is 28x28. The labels are contained in the vectors y_train_full and y_test. The values in the vectors are numbers from 0 to 9, and they correspond to the 10 classes in the following way. T-shirt/top corresponds to 0, trouser to 1, pullover to 2, dress to 3, coat to 4, sandal to 5, shirt to 6, sneaker to 7, bag to 8, ankle boot to 9. Before I begin, I remove 5000 images from the training data to use as validation data. As we learnt from lecture, the validation data should be randomly selected, but we select the first 5000 images for simplicity. I also convert the integers in x_train, x_valid, and x_test to floating point numbers between 0 and 1 by dividing each by 255.0.

There are two parts of the homework. The first part is to train a fully-connected neural network. Two kinds of neural networks will be used. The first one is a fully-connected neural network. In this part I will try several different neural network architecture with different hyperparameters to try to achieve the best accuracy I can on the validation data. I might try the depth of the network, the width of the layers, the learning rate, the regularization parameters, the activation functions, the optimizer. I would try to make the accuracy on the validation data as high as possible then use that architecture to train one final model.

The second part is to train a convolutional neural network. For this part, I can also try adjusting the number of filters for my convolutional layers, the kernel sizes, the strides, the padding options, the pool sizes for my pooling layers.

Theoretical Background

From our course notes, it says that for linear and logistic regression, there were three ingredients: the data, the model, and the loss function. Recall that the model was just a

function that gives the relationship between the independent variable (x) and the dependent variable (y), which could be scalars or vectors. For linear regression, the model was a linear function. For logistic regression, it was a linear function that was then plugged into a nonlinear function (the logistic or softmax function). A neural network is just a different model. It is a different (and more complicated) function that we can use to relate all our x and y variable. However, we are going to see that sometimes the function is so complicated that we don't write down the formula. Instead, we draw pictures.

In the model of neural nets, input vector is modeled as a column of nodes, and each edge represents weights. Each neuron inputs and outputs a number and each connection has a weight associated with it. The layer of neurons on the left are the input layer and they just output the same number they input. We will call these numbers x_1, x_2 and x_3. We can denote the weights of the connections by w_1, w_2 and w_3. For the output of the neuron on the right, we first calculate a weighted sum of the inputs:

$$z = w_1x_1 + w_2x_2 + w_3x_3$$

and we can write this sum using vectors (it is a dot product)

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix},\ w = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix},\ z = w^T x = w_1x_1 + w_2x_2 + w_3x_3.$$

Then we will say that the output neuron is active if this sum is greater than some threshold and inactive if the sum is less than some threshold. To do this mathematically, we can just plug the weighted sum into a step function.

$$y = \sigma(z) = \begin{cases} 0, & z < threshold \\ 1, & z \geq threshold \end{cases}$$

Then 1 represents an active neuron and 0 an inactive neuron. Therefore, the function σ is called an *activation function*. This is exactly what we did for linear discriminant analysis for dogs and cats which I explain in my last project's report. For linear discriminant analysis, we had a procedure for finding an optimal w vector in order to best separate our data into two classes. With this linear threshold unit, the calculation is exactly the same so these can be used for binary classification. For the activation function, we know that we always typical threshold which is 0. To account for shifting the threshold, we can add an extra neuron to the input layer called a bias neuron. It always outputs a 1. In terms of the formula this amounts to adding a constant term b.

$$y = \sigma(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

This is a linear threshold unit. A *perceptron* is composed of a layer of linear threshold units. Each one is connected to the input layer and has a different set of weights. If we define the vectors and matrix

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad A = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

then the calculation of the outputs y_1, y_2, y_3 can be written as $y = \sigma(Ax + b)$. In other words, you do the calculation $Ax + b$ and then you plug each component of that vector into the (step) *activation function*. Later, people found that we can solve much more complicated problems by stringing together multiple perceptrons. This gives what is called a *multilayer perceptron (MLP)*. The leftmost layer is the *input layer* and the rightmost layer is the *output layer*. Any layers in between are called *hidden layers*. Written as an equation, this neural network is

$$y = \sigma(A_2\sigma(A_1x_1 + b_1) + b_2)$$

where x_1 is the input layer, x_2 is the hidden layer, y_1 is the output layer, A_1 is the weight matrix between the input and hidden layer, b_1 are the biases between the input and hidden layer, A_2 is the weight matrix between the hidden and output layer, and b_2 are the biases between the hidden and output layer. It is much easier to think of this formula as two separate steps, one to compute the hidden layer and then another to compute the output layer.

$$x_2 = \sigma(A_1x_1 + b_1), \qquad y = \sigma(A_2x_2 + b_2)$$

There are now many different types of neural networks, some of which are quite different from the *multilayer perceptron*. The multilayer perceptron that we referred to earlier, typically paired with a sigmoid or ReLU activation function, is what we would refer to as a *feed-forward* neural network because everything flows in one direction from the inputs to the outputs. For a given layer, every neuron is connected to every neuron in the previous layer. We call those types of layers *fully-connected* or *dense*. When all of the layers are of that type, we say that it is a *fully-connected network*. The number of neurons in each layer is called the *width* of the layer. The number of layers is called the *depth* of the network. You may hear the terminology *deep learning* or *deep neural networks*. This just refers to neural networks with many layers.

The process of finding the weights and biases that minimize the loss function is known as training the network. One of the biggest problems with neural networks is overfitting. Overfitting tends to occur when you have a lot of parameters in comparison to the number of data points you have. For neural networks, every weight and bias is a parameter. So if we see that our neural network is performing much better on training data than test data, we can try to make some adjustments to your network (such as changing the depth of the network or the width of your layers). Essentially, we are making adjustments to the model. The standard procedure is to split our data into three parts: training data, validation data, and test data. Validation data plays the role of test data while we are trying to adjust aspects of your network and choose the right model. It can also help us figure out when to stop training our network. The test data is used at the very end to determine the final performance of our model.

The basic structure of a convolutional neural network (CNN), is built up by convolutional and pooling layers, fully connected layers, and output layers. For a single convolutional layer we have multiple feature maps. Each one has its own filter. But we consider all of them to be one convolutional layer. Stride is the distance to slide the receptive field. If we increase stride then the size of the feature map is decreased. Pooling layers are referred to as a downsampling

layer. It is also referred to as a down sampling, or subsampling layer, with max pooling and average pooling options. One common advantage of convolutional layers is that it can capture the same features even if the image is shifted (in other word, the pixels are changed a lot but the features as a whole do not change so much).

Algorithm Implementation and Development

I pretty much use the starter code that is provided, so I will briefly explain the starter code. Firstly, 60000 images of fashion items are loaded into the training set, and 10000 images are loaded into the testing set. First 5000 images in training set are selected as validation set. Then the integer values are converted to floating numbers by dividing them by 255.0. The standard ReLU activation function is used in fully-connected layers. A regularizer is the sum of coefficient square and applies L2 regularization. Then I use the built in function to create a model with a sequential pipeline.

For part one, I add flatten layer with input shape = 28x28, dense layer with 300 nodes, dense layer with 100 nodes, and output layer with 100 nodes. The learning rate is 0.0001. I experimented different number of nodes for the layers, and different number of layers. I also changed learning rate. However, I do not get optimal result. After discussion with other people, I found out that such configuration gives pretty good result.

For part two, I add convolutional layer with 256 filters, kernel size is 3x3, and zero padding. I have maxpooling layer with pool size = 2x2, convolutional layer with 128 filters, kernel size = 3x3, and no padding, maxpooling layer with pool size 2x2, flatten layer, dense layer with 128 nodes, and output layer with 10 classes. The learning rate is 0.001. I experimented with different number of filters for convolutional layers, and different kernel sizes and padding options.

Computational Results:

For part one, after I run the training process, the training accuracy is around 0.91, the loss is 0.3562, and the val_accuracy is 0.8854. The accuracy for the test is 0.8771. To answer the questions raised by the specification, I am not able to beat 90% accuracy. I believe this result is fine, because the confusion matrix tells that number 6 is problematic because many off diagonal entries occur on row 6 and column 6. In general, the model is reliable.

For part two, after I run the training process, the accuracy for the test is around 0.91, and the loss is 0.41, which is slightly better than the first case. However, in this case the accuracy for the training set is pretty high, around 0.98, but the accuracy for the validation data is not that high. This is an indication that the model is memorizing the examples in the training

set and not able to capture the creatures very well. In other words, the model lacks generality. Again, we can see from the figure 4 that the problem occurs on row 6 and column 6, so the problem category is number 6. In general, the model is doing well in other categories.

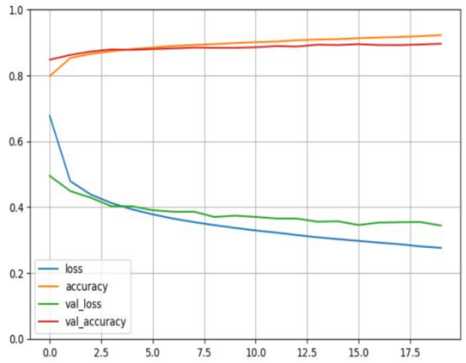

Figure 1 ：accuracy of training in part one

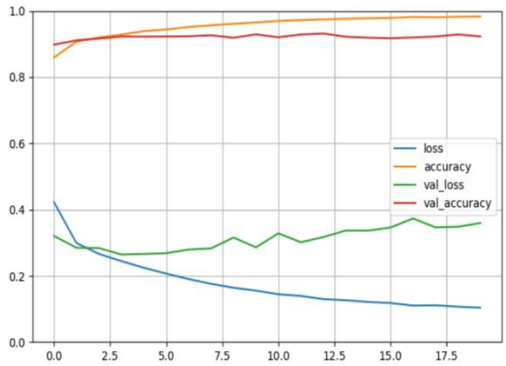

Figure 2: accuracy of training in part two

	0	1	2	3	4	5	6	7	8	9
0	862	1	15	34	4	0	80	0	4	0
1	3	963	0	29	2	0	2	0	1	0
2	14	0	834	19	68	1	64	0	0	0
3	20	2	11	934	14	0	17	0	2	0
4	1	0	95	58	757	0	88	0	1	0
5	0	0	0	1	0	964	0	19	1	15
6	126	0	86	41	43	0	696	0	8	0
7	0	0	0	0	0	16	0	957	0	27
8	6	1	5	9	3	3	10	4	959	0
9	0	0	0	0	0	5	1	35	0	959

Figure 3: confusion matrix for part one

	0	1	2	3	4	5	6	7	8	9
0	808	0	10	21	8	1	146	0	6	0
1	2	977	2	12	4	0	1	0	2	0
2	14	0	843	10	60	0	72	0	1	0
3	8	3	6	929	34	0	17	0	3	0
4	0	0	33	12	930	0	25	0	0	0
5	0	0	0	0	0	992	0	6	0	2
6	69	0	38	24	81	0	779	0	9	0
7	0	0	0	0	0	14	0	957	0	29
8	3	1	0	2	3	2	5	0	983	1
9	0	0	0	0	0	7	0	18	0	975

Figure 4 : confusion matrix for part two

Summary and Conclusion :

In this paper, I use neural nets to classify the given data, Fashion-MNIST. I use two kinds of neural nets. The first one is fully-connected neural network, and the second one is convolutional neural network. I experimented with different parameters for the model. Finally I believe I created a model that is doing well in general, except that it confuses category 6 with other categories.

Appendix B :

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


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:]

# Part 1
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
   tf.keras.layers.Flatten(input_shape=[28, 28]),
   my_dense_layer(300),
   my_dense_layer(100),
   my_dense_layer(10, activation="softmax")  # 10 classes to be classified
])

model.compile(loss="sparse_categorical_crossentropy",
          optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
          metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=50, 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)


fig, ax = plt.subplots()
# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')
```

```
# create a table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='center',
cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')


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

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]


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",
kernel_regularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    # do zero padding, so that the size of the feature map is the same as the size of the input
    my_conv_layer(32, 3, padding="same", input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(64, 3),
    tf.keras.layers.MaxPooling2D(2),
```

```
   tf.keras.layers.Flatten(),  # Always remember to flatten before processing into fully connected
layers
   my_dense_layer(256),
   my_dense_layer(10, activation="softmax")  # 10 classes to be classified
])

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)


fig, ax = plt.subplots()
# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')
# create a table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='center',
cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```