Classifying Fashion with Neural Networks

Alyssa Mell

Abstract

Using both a fully-connected neural network, and a convolutional neural network, we train a classifier in Python to differentiate between several clothing items provided in the data set Fashion-MNIST. We adjust multiple hyperparameters for each of these neural networks and study how they affect the accuracy of our classifier on the validity data. We attempt to maximize this accuracy in order to find the best model. Finally, we use this model on test data and determine how well our classifier performs.

Sec. I. Introduction and Overview

The Fashion-MNIST data used for this paper includes 10 different clothing items. These are classified as: 0 - T-shirt/top, 1 - Trouser, 2 - Pullover, 3 - Dress, 4 - Coat, 5 - Sandal, 6 - Shirt, 7 - Sneaker, 8 - Bag, and 9 - Ankle boot. We attempt to classify images of these items into their correct categories by training a neural network. However, when building a neural network, there are several different options to consider. We use this paper to present many of these options and how they affect the success of our classifier when they are changed.

There are two overarching parts to our analysis. We begin by testing a fully-connected neural network and, secondly, study how our accuracy changes when we use a convolutional neural network instead. For our fully-connected neural network we change the values of the hyperparameters such as the depth of the network, the width of the layers, the learning rate, the regularization parameters, the activation functions, and the optimizer. For our convolutional neural network, in addition to the hyperparameters listed above, we can also adjust the number of filters per layer, the kernel sizes, the strides, the padding options, and the pool sizes for pooling layers.

Sec. II. Theoretical Background

Neural networks have many important applications as a machine learning tool, but, in our case, they are used to aid in classification. Neural networks are created with an input layer, hidden layers, and an output layer. Weightings and various other hyperparameters are used to train the neural network for the specific problem.

In order to understand the main purpose of this paper, its important to understand the difference between a fully-connected and a convolutional neural network. In a fully-connected neural network, every neuron in a given layer is connected to every neuron in the previous layer. In contrast, in a convolutional neural network, each neuron is connected to only a few

spatially proximate neurons from the previous layer. We study how these differences affect each type of neural network's ability to classify images.

In our use of neural networks as a classification tool, we will use both an activation and loss function. The loss function used in this paper is the cross-entropy loss function described in equation 1.

$$-\frac{1}{N}\Sigma_{j=1}^{N}[y_j \ln(p_j) + (1-y_j)\ln(1-p_j)] \tag{1}$$

In equation 1, N is the number of data points, j is the current data point, y_j is a vector containing the probabilitites of data point j belonging to each possible category, and p_j is the probability of the correct category being classified for j. This function is negative so that it can be minimized it in order to reduce our loss. The logarithm within the expression makes it so that the optimization problem is convex, therefore having only a single local minimum.

We also use two activation functions in our neural network: Softmax and ReLu (rectified linear unit). These are defined respectively in equations 2 and 3. Equation 2 is the Softmax function which is commonly used for the output layer of a neural network because it gives us a vector of values that provide the probability distribution for all m values of y.

$$P = \sigma(y) = \frac{1}{\Sigma_{j=1}^{m} e^{y_j}} \begin{pmatrix} e^{y_1} \\ e^{y_2} \\ \vdots \\ e^{y_m} \end{pmatrix} \tag{2}$$

Equation 3 is the ReLu function which zeros out all negative entries in a neural network layer and leaves positive entries alone, it is commonly used in central layers of a neural network.

$$\sigma(x) = \max(0, x) \tag{3}$$

Sec. III. Algorithm Implementation and Development

In this section there will often be references to the Python code which is located in Appendix B. The specific line of the code that is being referenced will be stated at the end of the sentence. The code implementation and development for our fully-connected neural network was very similar to that for our convolutional neural network, with a few additional factors that will be mentioned below.

For both cases, we begin by separating our initial training data into two parts. We remove the first 5,000 images to use as validation data and leave the remaining 55,000 images as the training set (lines 22-24). Next, we divide the integer values for each data point by 255.0 in order to convert them to floating points between 0 and 1 (lines 22-24). With the convolutional neural network there is one additional step of adding a third dimension to the data using `np.newaxis`. This is necessary in order to use the convolutional layers. We then begin to build the neural networks for each part. Beginning with the fully-connected neural network, we test the following hyperparameters listed below. Line numbers reference the Fully-Connected Neural Network section of Appendix B.

- Depth of network - the depth of the network was determined by the number of layers inside the model (lines 37 to 45). The optimal number of layers was found through experimentation. The first layer was necessary in order to flatten the input data into a workable form (line 37). The final layer was also necessary because it output the final probability values for the correct classification of each fashion item (line 45). This was done by setting the number of neurons to 10 and using the activation function "Softmax" which converts all values to numbers between 0 and 1. All layers in between these two were experimented with to find the optimal solution. Using too few layers causes the accuracy of the model to decrease, and using too many causes the model to take too long to run and risk overfitting. Therefore we used five central layers with different widths, along with two dropout layers that are discussed below.

- Width of layers - The width of each layer is set by the number within `my_dense_layer`(). This determined the number of neurons in the layer. Again, having too many neurons per layer caused the model to take too long to run and tend towards overfitting. However, having too few neurons per layer decreased its accuracy. The width of the layers were determined to be 300, 100, 100, 50, and 20 respectively.

- Learning rate - the learning rate of a neural network controls how much to change the model in response to an estimated error rate after each update to the model weights. Choosing a very small value for the learning rate greatly increases the time required to run the program, however choosing a value too big could cause a suboptimal model to be chosen. Through experimentation and trial and error it was found that a learning rate of 0.1 best fit our model (line 50).

- Regularization parameters - these parameters help reduce the likelihood of the model overfitting. We used both L2 regularization and Dropout layers to help with overfitting in our model (lines 32 and 39). We test different regularization methods such as L1 instead of L2, but found that L2 worked best for our model. Additionally, we experimented with the value inside the regularizer and eventually settled on 0.00001 as the optimal value to ensure no weights get too large (line 33). After experimentation, we set the Dropout layers to have a dropout rate of 0.1, values much higher than this tended to reduce the accuracy of our model.

- Activation functions - the activation function determines many components of the neural network. It affects the output, accuracy, and efficiency of the model. Again, through simple experimentation with different activation functions such as ReLu, tanh, sigmoid, and Softmax, we determined that ReLu worked best for our model (line 32).

- Optimizer - we tested two main optimizers for our model, Adam and SGD (stochastic gradient descent). With Adam, the learning rate needed to be changed from 0.1 to 0.0001 in order to get similar results. However, it did not reach an accuracy rate quite as high as SGD, so we decided on SGD for our final model (line 50).

- Epoch - the epoch determines the number of times we loop through the training data while forming our classifier. Too many loops led to overfitting and too few led to

underfitting. Through trial and error methods, we determined that an epoch of 12 worked best with our other hyperparameters (line 54).

Next, for the convolutional neural network, we tested all of the hyperparameters above along with a few additional ones. For the hyperparameters above, the best hyperparameters were found to be: 2 convolutional layers, 1 fully-connected layer (width = 128), 1 max pooling layer, 1 dropout layer (rate = 0.3), learning rate of 0.0001, L2 regularizer with value 0.0001, ReLu activation function everywhere except Softmax for output, Adam optimizer, and an epoch of 20. For the new hyperparameters we describe our process of optimizing the model below. Line numbers reference the Convolutional Neural Network section of Appendix B.

- Number of filters per convolutional layer - this hyperparameter determines the dimensionality of the output space. After some research and experimentation it was found that two convolutional layers with 6 and 32 filters respectively provided the best results (lines 36-37).
- Kernel sizes - this hyperparameter specifies the height and width of the convolutional layer. It was found that values of 4 and 3 respectively for each layer provided the best results (lines 36-37).
- Strides - the stride of the convolutional layer determines how far over each sample shifts across the input layer. Increasing the stride tended to decrease the accuracy of the model, so the stride was left at 1 (lines 36-37).
- Padding options - the padding options allowed us to add zero values around a layer in order to control the size of the following layer, the value "same" pads the layer with zeros and the value "valid" does not. This was set to "same" for the first convolutional layer but was left as "valid" for the second (line 36).
- Pool sizes of pooling layers - only one pooling layer was used in our model. This was the `tf.keras.layer.MaxPooling2D()` layer. We used a value of 2 for this layer meaning that the maximum of every 2x2 grid was taken for the output layer (line 38). Using larger values reduced the accuracy of our model. We tested with average pooling layers as well but these led to lower accuracy models.

After running each model with the conditions specified above, we were able to create confusion matrices for both the training and test data using the command `confusion_matrix()`. Finally, we evaluated the accuracy of our models by using the command `model.evaluate()`.

Sec. IV. Computational Results

Fully-Connected Neural Network

For the fully-connected neural network, we were able to reach the following accuracies with our model: training accuracy - 89.61%, validation accuracy - 89.56%, test accuracy - 88.29%. The loss and accuracy for the training and validation data at each epoch is shown in figure 1a. The blue and green lines in this figure show the training and validation loss respectively.

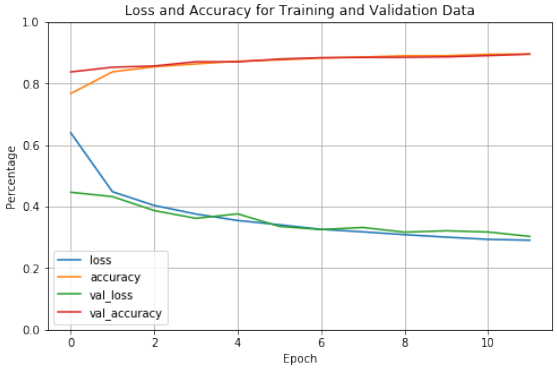

Figure 1a : Fully-connected neural network model.

	0	1	2	3	4	5	6	7	8	9
0	882	1	28	16	5	2	59	0	7	0
1	5	966	2	22	3	0	1	0	1	0
2	19	2	830	8	96	0	42	1	2	0
3	23	5	18	872	49	0	28	0	5	0
4	1	1	120	20	802	0	56	0	0	0
5	0	0	0	0	0	973	0	15	0	12
6	170	0	140	24	63	0	594	0	9	0
7	0	0	0	0	0	55	0	912	0	33
8	3	0	6	4	5	4	4	5	969	0
9	0	0	0	0	0	5	0	34	1	960

Figure 1b : Fully-connected neural network confusion matrix for test data. Column (true) and row (test) labels 0-9 represent each fashion item as defined in Sec. I.

Since the training loss is not significantly lower than the validation loss, we can conclude that the model is not over fitting. Using a confusion matrix, we found that the most common mistake in this classification was 0 - t-shirts - classified as 6 - shirts - as seen in figure 1b.

Convolutional Neural Network

For the convolutional neural network, we were able to reach the following accuracies with our model: training accuracy - 93.12%, validation accuracy - 92.24%, test accuracy - 91.34%. In comparison to the fully-connected neural network, these accuracies are quite a bit higher. However, as shown in figure 2a, it also seems as though our model is overfitting more than before because the training loss is somewhat lower than the validation loss. Using a confusion matrix, we found that the most common mistake in this classification was again 0 - t-shirts - classified as 6 - shirts - as seen in figure 2b.

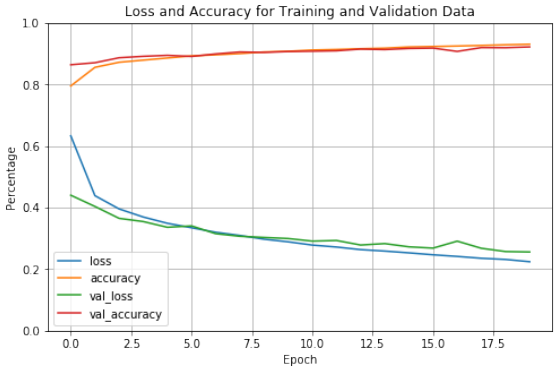

Figure 2a : Convolutional neural network model.

	0	1	2	3	4	5	6	7	8	9
0	887	0	17	14	3	1	71	0	7	0
1	1	979	0	14	2	0	2	0	2	0
2	19	2	884	7	39	0	49	0	0	0
3	17	1	15	924	16	0	23	0	4	0
4	1	1	62	35	853	0	48	0	0	0
5	0	0	0	0	0	975	1	14	0	10
6	130	0	66	25	58	0	712	0	9	0
7	0	0	0	0	0	8	0	967	1	24
8	3	1	3	2	1	1	4	3	982	0
9	1	0	0	0	0	4	0	24	0	971

(a) Figure 2b : Convolutional neural network confusion matrix for test data. Column (true) and row (test) labels 0-9 represent each fashion item as defined in Sec. I.

Sec. V. Summary and Conclusions

In this paper, we used two different types of neural networks - fully-connected and convolutional - to create a classifier for Fashion-MNIST. This allowed us to compare the effectiveness of each type of neural network in correctly classifying images of clothing. From our experimentation, we determined that convolutional neural networks seem to be more effective in building classifiers for the images in our study since they allow for a higher classification accuracy to be reached.

Appendix A. Python Functions

Listed below are the important Python functions used in this paper, along with a brief explanation of how they are implemented.

- `tf.keras.layers.Dense`(units, activation = "function") : creates a fully-connected neural network layer with width given by "units" and a specified activation function.
- `tf.keras.layers.Conv2D`(filters, kernel_size, activation = "function") : creates a convolutional neural network layer with output size given by "filters", size of the convolutional layer given by "kernel_size", and a specified activation function.
- `tf.keras.layers.Dropout`(rate) : helps prevent overfitting by randomly setting a fraction rate of input units to set to 0 during training.
- `tf.keras.layers.Flatten`(input_shape = [N,N]) : reshapes the data into a single column, must be provided with initial NxN data size.
- `tf.keras.layers.AveragePooling2D`(pool_size) : downscales by factor "pool_size" by averaging input each time.
- `tf.keras.layers.MaxPooling2D`(pool_size) : downscales by factor "pool_size" by taking the maximum input each time.
- `model.compile`(loss = "function", optimizer = tf.keras.optimizers.SGD(learning_rate = rate), metrics = ["accuracy"])) : sets initial values for neural network such as loss function, optimization function, learning rate, and accuracy metrics.
- conf = `confusion_matrix`(y_test,y_pred) : created a matrix showing the number of correctly classified photos for each object.
- `model.evaluate`(X_test,y_test) : determines the accuracy of the model on the test data.

Appendix B. Python Code

Fully-Connected Neural Network

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

# Load fashion data
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = \
    fashion_mnist.load_data()

# Plot first 9 images
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()

# Separate data into training and validation
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

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

# Creates all fully-connected layers
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(300),
    tf.keras.layers.Dropout(0.1),
    my_dense_layer(100),
    my_dense_layer(100),
    tf.keras.layers.Dropout(0.1),
    my_dense_layer(50),
```

```
    my_dense_layer(20),
    my_dense_layer(10, activation="softmax")
])

# Adds opimizer to neural network
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
              metrics=["accuracy"])

# Trains neural network
history = model.fit(X_train, y_train, epochs=12,
                    validation_data=(X_valid,y_valid))

# Plots loss and accuracy for training and validation data
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epoch')
plt.ylabel('Percentage')
plt.title('Loss and Accuracy for Training and Validation Data')
plt.show()

# Plots confusion matrix for training data
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

# Plots confusion matrix for test data
y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)

# Evaluates model on test data
model.evaluate(X_test,y_test)
```

Convolutional Neural Network

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

# Load fashion data
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = \
    fashion_mnist.load_data()

# Separate data into training and validation
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:]

# Add third dimension
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]

from functools import partial

# Creates baseline dense and convolutional layer
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu",
                         kernel_regularizer=tf.keras.regularizers.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D,
                        activation="relu", padding="valid")

# Creates all layers
model = tf.keras.models.Sequential([
    my_conv_layer(6,4,padding="same",input_shape=[28,28,1]),
    my_conv_layer(32,3),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Flatten(),
    my_dense_layer(128),
    my_dense_layer(10, activation="softmax")
])

# Gives dimensions of model
model.summary()
```

```
47
48 # Adds opimizer to neural network
49 model.compile(loss="sparse_categorical_crossentropy",
50               optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
51               metrics=["accuracy"])
52
53 # Trains neural network
54 history = model.fit(X_train, y_train, epochs=5,
55                     validation_data=(X_valid,y_valid))
56
57 # Plots loss and accuracy for training and validation data
58 pd.DataFrame(history.history).plot(figsize=(8,5))
59 plt.grid(True)
60 plt.gca().set_ylim(0,1)
61 plt.show()
62
63 # Plots confusion matrix for training data
64 y_pred = model.predict_classes(X_train)
65 conf_train = confusion_matrix(y_train, y_pred)
66 print(conf_train)
67
68 # Evaluates model
69 model.evaluate(X_test,y_test)
70
71 # Plots confusion matrix for test data
72 y_pred = model.predict_classes(X_test)
73 conf_test = confusion_matrix(y_test, y_pred)
74 print(conf_test)
75
76 fig, ax = plt.subplots()
77
78 # Hide axes
79 fig.patch.set_visible(False)
80 ax.axis('off')
81 ax.axis('tight')
82
83 # Create table and save confusion matrix to file
84 df = pd.DataFrame(conf_test)
85 ax.table(cellText=df.values, rowLabels=np.arange(10),
86          colLabels=np.arange(10), loc='center', cellLoc='center')
87 fig.tight_layout()
88 plt.savefig('conf_mat.pdf')
```