Neural Networks for Classifying Fashion MNIST

By: Priyanshi Rastogi

1. **Abstract:** In this report we look at how we can train neural networks (in Python) to classify images, in this case it'll be images of 10 different classes of fashion items, using the Fashion MNIST dataset. We use a fully connected neural network and a convolutional neural network and adjust the hyperparameters to make it so that our validation data's accuracy is above 90% and try to make it above 95% as well. We see how increasing/decreasing the various hyperparameters of the built neural network can affect our accuracy rate.
2. **Introduction/Overview:** We are given 60,000 images to train with and 10,000 images to test with, from which we are supposed to classify each image into one of the 10 categories. The labels 0-9 each correspond to a specific fashion item. The images themselves each contain an image of a fashion item. In this report, we will explore how different hyperparameters such as the width of a layer, the depth of the neural network, the activation functions, the learning rate etc. affect the accuracy of our model to be able to correctly classify the testing images.
3. **Theoretical Background:** For any machine learning classification problem we have to split our data into 3 different batches. One is the training data that we will have our model train with, the second is our validation data to make sure the model isn't overfitting (being really accurate on the training data but not on the testing data) and then the actual testing data. To classify our images, we train our fully connected and convolutional neural network. A neural network consists of an input layer, a (or multiple) hidden layers (a multi-layer perceptron), and an output layer. It takes in a picture (pixel values) as an *input* and *outputs* a probability which tells us, in this case, 10 probabilities, each one corresponding to what the probability of being in each (of the 10) classes is. Each layer has a width, the number of neurons it is being trained with, and the output layer's width is the number of classes there are that can be determined. In a fully connected neural network, each layer is connected to the next with some weight and bias which is then put through the **activation function**. Therefore let's say $x_1, x_2, \ldots x_n$ represents each neuron and that each weight connected to the next neurons (ex. In a fully connected neural network) has weights $w_{11}, w_{12}, \ldots w_{nn}$. Thus, we can write the process of this *multi-layered perceptron* as

$$x_2 = \sigma(A_1 x_1 + b_1), \text{ and } y_1 = \sigma(A_2 x_2 + b_2) \text{ and so on} \ldots (1)$$

Where x, y, b, are all vectors and A_n was the corresponding matrix with the coefficients as each weight's value. The more layers we have, the more functions of function computations we'll be doing as we can see from eq. 1. Here σ(x) is our chosen activation function. The most common activation function is, *Rectified Linear Unit (ReLu),* which is one of the most popular activation functions in machine learning. This function zeros out all of the negative values and keeps the positive values as is.

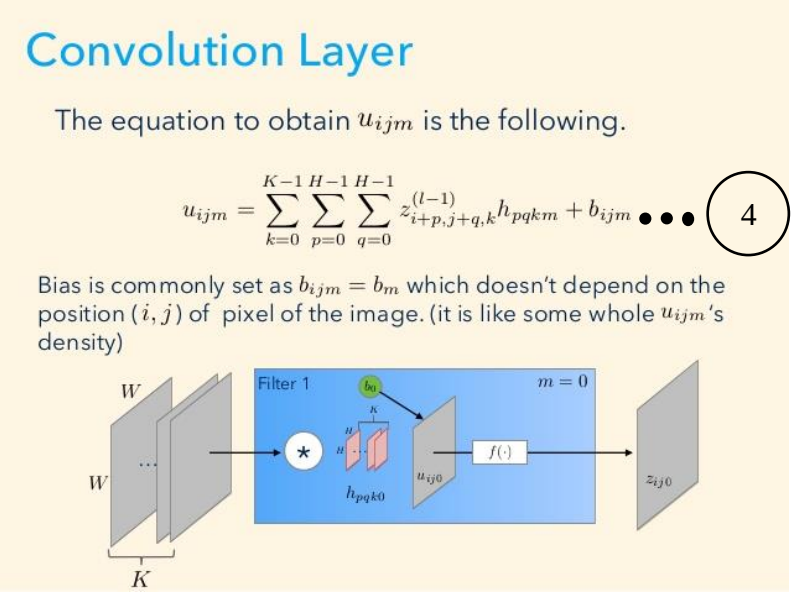
$$f(x) = x^+ = \max(0, x), \quad \ldots (2)$$

So, we will end up needing data, whose input and output layers' widths will be the same, the model that we will train, and the loss function which in our case will be *SoftMax* because it returns probabilities for each class. The *softmax functio*n being:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \quad \ldots (3)$$

After we have trained our model, we can explore how different values for the hyperparameters make our validation data's accuracy better and its loss as small as possible.

For CNN's, each layer works to detect a simple feature about the image and as you go deeper, the layers start to detect/look for more complicated features. Each layer has some receptive field of the neurons (connected to one before that it is attached to). Each receptive field contains values and so does the corresponding filter that we apply (both of the same size). We multiply out each value in the receptive field with its respective value in the filter and sum them up to get the value of that respective neuron and do this for each neuron, keeping the filter the same (but with different receptive fields). If we want to cover all of the edges in the receptive fields (have them be the center when sliding over the filter) we won't be able to do that unless we add *paddings* of values with all 0 around the edges so that the edges themselves can be centers too. In the end we want our feature maps to be the size of our image, with each convolutional layer having multiple different feature maps each with different filters. Since if we apply many convolutional layers, it might end up taking a lot of our memory space then we can subsample our layers by a method called Average or Max pooling. Average pooling takes the average of all of the numbers in the receptive field and max pooling just takes the max of the numbers. Again, we can pick the size of the pool, the stride, and if we want zero padding just like convolutional layers. Here u_{ijm} is the value of each neuron for each image and how we obtain that



4. **Algorithm Implementation and Development:** We chose to use Python because Python has many more advanced machine learning libraries than MATLAB. Python is also used a lot in artificial intelligence in the industry. In this report we used the following libraries: NumPy, TensorFlow, Matplotlib, Pandas, and Scikit-Learn. For our fully connected neural network, we had multiple steps.

1. We first imported all of the necessary libraries that we were going to use to train, build, and findings we were going to display for our network and load in the Fashion MNIST dataset. (see lines 1-7)
2. We split the loaded data set into 2 parts, the x part of the training data and the y part of the training data. We then take the first 5000 images (and their corresponding labels) to be our validation data and the rest 55,000 images (and their labels) to be our training data. (see lines 14-18). The X part of the training data consists of the actual image into pixels, and then y part of the training data consists of the images' corresponding labels. We can plot a few of the images to see for ourselves. (see lines 8-13)
3. Then we start building our model. We create an input and output layer of width 10 and then play around with how having various depths (number of hidden layers) each of various widths (number of neurons) can make our model worse or better. We can also make the activation function what we think best suits our data, some choices including sigmoid, tanh, and ReLu. We also apply a regularizer (with a parameter value) which exists to avoid overfitting. We then apply the softmax (eq.3 function to our outputs to produce the results into probabilities. (see lines 20-29)
4. After we have created each layer with various widths, since we want to have the least loss possible, we add an optimizer that will account for our loss (using the sparse categorical cross entropy loss). Here we also have various choices, we can use the Stochastic Gradient Descent, Adam, or other optimizers. (see lines 30-31)
5. We then fit our model with our trained x,y's and with a certain number of epochs (that we can also change and see how that affects our model). (see line 33)
6. We then plot how our overall loss and accuracy did for both the validation data and the training data. We also print out our confusion Matrix, for both the training and testing data, which will tell us how many of each label it was able to guess and how many times our model guessed some other label instead. (see lines 34-44)

For part 2: Using the Convolutional Neural Network

1. For our CNN, our import statements are the same since we use the same libraries and since we are using the same dataset, the loading statements are the same as well.
2. For CNN's since we are using TensorFlow, it wants our data to be of an added dimension 1 at the end, we add a new axis at the end. (see lines 13-15)
3. In general, we are doing the same things as we did for the fully connected model, but just changing the model and the layers/hyperparameters themselves.
4. For CNNs we don't want to start by flattening the input, so we start by adding a convolutional layer the first number corresponding to the *number of filters* that we want and the second number corresponding the *size of the filter (kernel size)*. Here we use zero padding, which is why we say "same", and then add an activation function with the function of our choice (these are just a few of the several parameters that we can explore) (see lines 17-20)
5. We can also add the Max or Average pooling layers as well as more dense layers. After researching there were a lot of different layers that are listed in Keras that we can explore. A key thing that we want to remember is that we need to flatten our layers before we want to use a dense layer (for fully connected layers). (see lines 21-31)
6. We can then test our model and again plot our accuracies and losses and print our confusion matrix for training and testing data as we did before. (see lines 32-end)

5. **Computational Results: Part 1:** After changing various hyperparameters and plotting our accuracies and losses for our validation and training data we were able to get a validation data's accuracy of about 90.13 %. We were able to get this by setting the learning rate to .01 using the SGD optimizer, the width of the first hidden layer to be 350 and the second hidden layer's width to be 80. We also set the Regularizer rate to be 0. 0000001. After some researching online, we also added in a batch normalization layer right after adding the input layer, which normalizes and scales the inputs or activations and it made the accuracy to go up just a bit. If we made the learning rate too small then it ended up taking longer to converge but it may not over-step on where the minimum loss is and if we made it too high there was a chance it could never converge, so we had to find the right balance. We also found out that after adding the first hidden layer adding a dropout layer with rate 0.1, randomly sets the fraction, rate, of input to 0 at each update during training time and helps prevent overfitting.

We also added a 2nd hidden layer after the second hidden layer again to make sure it was preventing any overfitting and really picking up the minimum loss values wherever they occurred. If we had too many neurons, there would be too many parameters (weights) for it to learn, but it seemed that (from the final model) if you gradually decreased the width of each layer, i.e. smoothly trying to transition into the 10 classes, it seemed to be slightly more accurate. After plugging in different functions, it seemed that ReLu was the best and most popular activation function and other functions, such as sigmoid and tanh, didn't really seem to improve the accuracy. We also saw that increasing the regularizer rate seemed to make the accuracy worse. After increasing the depth of the neural network (to 3/4), we saw that the accuracy of both the validation and training data went down slightly as well as if we just kept one layer then the accuracy wasn't doing too well either, so we ended up having 2 hidden layers with their corresponding widths in decreasing order. The number of epochs stayed between 10-25 and it seemed that after a certain number in each run the accuracy seemed to converge to a number. As we can see from Figure 1, the loss of the validation data and the training data seem to be very close to each other, and that the validation data's accuracy and the training data's accuracy are almost identical. Thus, we can see that there is no overfitting happening. From the confusion matrices (figure 2 & 3) we can most clearly see that our model was having a hard time distinguishing between a t-shirt and a shirt and a some other not as obvious confusions elsewhere.

Part 2: For CNN's after exploring the various hyperparameters we were able to obtain a validation accuracy of 92.15%. The CNN ended up doing much better in terms of accuracy than compared to the fully connected neural network. We tried combinations of different activation functions for both the dense layers and the convolutional layers and it still seemed that ReLu ended up giving us the highest accuracy. Increasing the number of epochs, ended up just taking way too long to converge which is why we set it to a lower number like 10. When using CNN's it seemed Adam as the optimizer gave us a slightly better accuracy vs. using SGD. Increasing the number of filters of the convolutional layer seemed to make the accuracy worse and it seemed that if we started off with a higher number of filters for one of the convolutional layers and then decreased it for the next convolutional layer later, the accuracy improved. Not including padding in our convolutional layers seemed to make things worse. Having a convolutional layer followed by a max pooling (instead of average pooling) also seemed to increase accuracy. We played around with not having dense layers at the end, but it seemed to make the accuracy much worse. In general, it also seemed that our dropout rate was best when it was kept below 0.5. From our confusion matrices (figure 5&6)

we can see that our testing data did better than our training data overall and that the only clear confusions look like it was between distinguishing what is a t-shirt and what is a top.

Figure 1: Plot of validation data's loss and accuracy with training data's loss and accuracy.

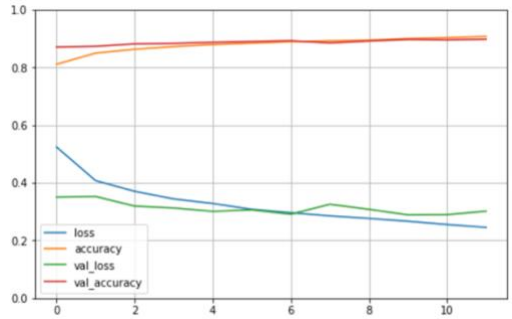


Figure 2: Confusion Matrix for Training Data

```
[[4979    5   59  107    8    1  379    2    3    0]
 [   1 5399    0   41    2    0    1    0    0    0]
 [  49    2 4810   29  349    0  255    0    2    0]
 [  40    8    9 5313   96    1   32    0    0    0]
 [   5    4  246  168 4923    0  163    0    3    0]
 [   0    0    0    0    0 5377    0  125    2    3]
 [ 487    6  256  101  235    0 4415    0    7    0]
 [   0    0    0    0    0    6    0 5459    1   22]
 [   5    1    4   11   11    2   33    3 5440    0]
 [   0    0    0    0    0   15    0  279    0 5200]]
```

Figure 3: Confusion Matrix for Testing Data

```
[[849   2  18  25   2   3  94   3   4   0]
 [  4 977   0  15   2   0   1   0   1   0]
 [ 15   2 796  13  97   0  75   1   1   0]
 [ 20   4  11 917  25   0  19   1   3   0]
 [  0   1  77  41 838   0  42   0   1   0]
 [  0   0   0   0   0 942   0  47   0  11]
 [119   1  79  33  70   0 690   0   8   0]
 [  0   0   0   0   0  10   0 983   0   7]
 [  3   0   4   5   6   4   7   6 965   0]
 [  0   0   0   0   0   9   1  62   0 928]]
```

Figure 4: Plot of accuracies/losses for training and validation Data for CNN Part 2

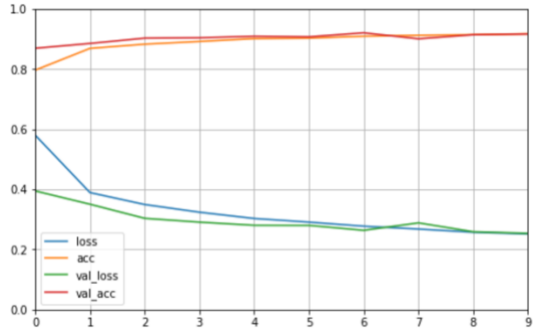


Figure 5: Confusion Matrix for CNN for training data

```
[[5058    1   79  123   11    1  258    0   12    0]
 [   0 5421    0   16    3    0    2    0    2    0]
 [  33    1 5056   45  258    0  102    0    1    0]
 [  25    8    9 5323  101    0   33    0    0    0]
 [   3    3  273  117 5028    0   86    0    2    0]
 [   1    0    0    0    0 5481    0   17    1    7]
 [ 566    2  359  141  484    1 3949    0    5    0]
 [   0    0    0    0    0   10    0 5426    2   50]
 [   6    0    4    5   12    2   11    1 5469    0]
 [   0    0    0    0    0   13    0  170    0 5311]]
```

Figure 6: Confusion Matrix for CNN for testing data

	0	1	2	3	4	5	6	7	8	9
0	874	0	20	26	5	2	68	0	5	0
1	1	986	0	10	1	0	0	0	2	0
2	13	0	898	13	49	0	27	0	0	0
3	11	3	8	936	21	0	20	0	1	0
4	1	1	67	28	875	0	27	0	1	0
5	0	0	0	0	0	987	0	11	0	2
6	122	1	78	35	118	0	639	0	7	0
7	0	0	0	0	0	8	0	984	0	8
8	5	2	1	0	3	2	4	2	981	0
9	1	0	0	0	0	7	0	45	0	947

6. **Summary and Conclusions:** From part 1, after many trials and errors we were able to see that it was pretty hard to get above a 90% (90.13%) accuracy for the validation data. After trying various hyperparameters, we saw that in the end we had the SGD optimizer (with learning rate 0.1), 2 hidden layers one first one with width of 350 and the second with width of 80. After some researching, we also added a dropout layer after each hidden layer which prevented any further overfitting and slightly improved our accuracy. Adding the batch normalization after the input layer also improved our accuracy, ending with 13 epochs. In part 2, it was much easier to get over a 90% on the validation data accuracy. Our CNN ended up giving us a 92.15% accuracy for validation data and didn't seem to be doing any overfitting. This is expected because fully connected layers tend to do worse since (because each neuron is connected to the previous) there are many more weights that need to be learned. Instead, CNN's let us learn a lot of more different feature maps and are great for when the images are shifted since the filters just get shifted which as well isn't the case in the fully connected neural networks.
7. **Appendix A-Function Descriptions: (all of these functions have several more parameters if wanted)**
    - tf.keras.layers.Dense(units, activation, regularizer_rate): creates a dense neural network layer with the given width (units), activation function and regularizer rate.
    - tf.keras.layers.Conv2D(num_filters, kernel_size, padding): creates a convolutional layer with the passed in number of filters, each with size kernel size and can specify padding=”same” for zero padding.
    - tf.keras.layers.Dropout(rate): Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.
    - tf.keras.layers.Flatten(shape): Flattens input into one column, necessary before adding a dense layer.
    - model.compile(loss, optimizer, metrics): Configures the model for training, with the specified loss function, chosen optimizer with learning rate, and list of metrics to be evaluated by the model during training and testing.
    - model.fit(x_train, y_train, epochs, validation_data): Trains the model for a fixed number of epochs (iterations on a dataset).
    - model.evaluate(x_test, y_test): Returns the loss value & metrics values for the model in test mode.
    - confusion_matrix(y_true,y_predict): Compute confusion matrix to evaluate the accuracy of a classification.
    - tf.keras.layers.BatchNormalization(): Normalize and scale inputs or activations.

8. 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
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 #first 5000 is for validation
X_train = X_train_full[5000:] / 255.0 #5000-55000 is for training
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(0.0000001))

model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=[28, 28]),
tf.keras.layers.BatchNormalization(),
my_dense_layer(350),
tf.keras.layers.Dropout(0.1),
my_dense_layer(80),
tf.keras.layers.Dropout(0.2),
my_dense_layer(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy",
optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=13, 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 2:

```
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()
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="relu", kernel_regularizer=tf.k
eras.regularizers.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="valid")

model = tf.keras.models.Sequential([
my_conv_layer(64,(4,4),padding="same",input_shape=[28,28,1]),
tf.keras.layers.MaxPooling2D(4,4),
my_conv_layer(64,(4,4)),
tf.keras.layers.MaxPooling2D(2,2),
tf.keras.layers.Dropout(0.6),
tf.keras.layers.Flatten(),
my_dense_layer(256),
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=10, 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 table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='ce
nter', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```

References:

https://www.slideshare.net/matsukenbook/deep-learning-chap6-convolutional-neural-net