Neural Network Classification of Fashion MNIST Data

Maxwell Weil

Abstract

In this paper, we will be exploring neural network (NN) classification methods, and how they can be used to categorize real datasets. In order to accomplish this, a background of NNs will be provided, along with the specific algorithms used within this assignment. We will then give an overview of the network architectures used, and the results achieved by these models. Through this, we hope to demystify the concepts used in NNs and present a simple application of these analytical models.

I. Introduction

Over time, researchers have developed a number of techniques for classifying and processing data to make informed, automated decisions. However, many of these models have consistently struggled to reproduce the precision and power one of the most computationally complex systems in the world; the human brain. In the mid 1900s, scientists drew inspiration from biological NNs to create new machine learning algorithms in the form of artificial neural networks (ANNs). These algorithms use nodes with weighted links to represent neurons and synapses, respectively. Nodes can then be grouped together to form layers of varying sizes, with each sequential layer being connected to the next, creating a network of artificial cells.

Using the general architecture above, researchers have been able to generate a variety of complex networks that are capable of effectively processing and classifying many different kinds of data. However, small customizations can be made to these networks to create entirely new kinds of processing methods. In this work, we will be illustrating different styles of NNs, and how they can each be used to process a large dataset. Both a fully-connected NN and a convolutional neural network (CNN) will be used to demonstrate a few of the features of NN classification.

Before discussing the design and results of the NNs used in this assignment, a brief overview of the mathematical concepts used with be given. This will hopefully elucidate some of the methods and techniques used throughout this paper.

II. Theoretical Background

As previously mentioned, a NN consists of several interconnected layers, each composed of nodes and links. The value of given node within a layer is based on the sum of all the inputs to that node multiplied by their respective weights, as shown in equation 1.

$$y = w_1 x_1 + w_2 x_2 \dots = \sum_{j=1}^{N} w_j x_j \qquad \text{EQ. 1}$$

In this equation, y represents the value at a node, x_j is the value of a node in the previous layer, w_j is the strength of the connection between x_j and y, and N is the number of nodes in the previous layer. With this, we imitate the structure of biological neurons, where a given neuron, y, receives input for a variety of others, x_j, each with varying weights, w_j. However, in biology the output of y would not be directly equal to the sum of the inputs, instead it is either on or off, depending on whether the summed inputs were above a certain threshold. In ANNs, this is known as an activation function. The activation function is applied to y in order to determine the value for inputting into the next layer of neurons. While a variety of activation functions can be used, we will be utilizing the rectified linear unit function, or ReLU, which is shown in equation 2.

$$z = ReLU(y) = \begin{cases} 0, & y \leq 0 \\ 1, & y > 0 \end{cases} \qquad \text{EQ. 2}$$

This equation essentially thresholds the value at y, giving us either 0 or y as the value of z, which will serve as an input to the next layer. The threshold can also be shifted by using what is called a bias neuron. This bias neuron always outputs a constant value of 1 and would be added into the sum used in equation 1. Combining all of these concepts together, we can use equation 3 to find the values in a layer given inputs from a previous layer.

$$\boldsymbol{z} = ReLU(\boldsymbol{Ax} + \boldsymbol{b}) \qquad \boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad \boldsymbol{A} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix} \qquad \boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \qquad \text{EQ. 3}$$

This equation illustrates how inputs of x_n nodes and b_m biases will weigh into an output of z_m nodes in the next layer. In generally, this equation can apply to any number of layers within a NN, but a final classification layer is always necessary. Our output classification layer should tell us what the predicted class is for a given input. Most sensibly, this can be done by outputting the probability of the input belonging to any one class. To get this information, a softmax function, as shown in equation 4, can be used.

$$\boldsymbol{z} = \sigma(\boldsymbol{y}) = \frac{1}{\sum_{j=1}^{m} e^{y_j}} \begin{pmatrix} e^{y_1} \\ \vdots \\ e^{y_m} \end{pmatrix} \qquad \text{EQ. 4}$$

This will serve as the activation function for the final output layer of a NN, with z representing the probability vector for a given number of classes and y being the sum of the inputs (as in equation 1). These principles have so far explained how a NN computes an output from a given input, but not how to find the correct output. To achieve this, we must use supervised training, where the predicted class is compared to the true class, and the network's model is adjusted accordingly. To find the different between the predicted and true classes, a variety of algorithm can be used. In this work, the cross-entropy loss, as shown in equation 5, will be used to find how greatly our predictions differ from the true values.

$$L = -\frac{1}{N}\sum_{j=1}^{N} y_j * \ln(z_j) + (1 - y_j) * \ln(1 - z_j) \qquad \text{EQ. 5}$$

This equation computes the loss (or error) of our network, L, by comparing the true classes, y, to the predicted classes, z. Once we know the loss of our network for a given data point, the weights between nodes have to be adjusted to minimize this loss. To account for the many weights used, an optimization method known as stochastic gradient descent, shown in equation 6, is used.

$$\boldsymbol{A}_1 = \boldsymbol{A}_0 - \delta \nabla L(\boldsymbol{A}_0) \qquad \text{EQ. 6}$$

This equation uses the negative gradient of the loss function, L, to adjust the weights of the network, **A**, with a step size of δ. The negative gradient can be used because the loss function should be convex at most points, due to the use of the log function. Essentially, this causes the loss function to be minimized by moving down the gradient of the function in steps towards a minimum. However, due to the complexity of NNs the loss is often not completely convex, leading to some issues with getting caught in local minima.

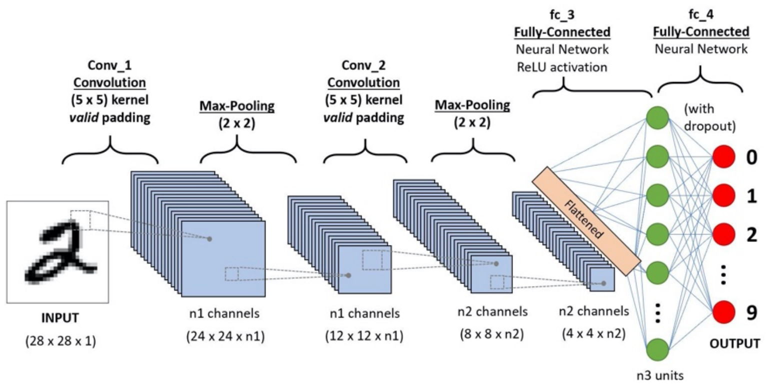

Figure 1. Example architecture of a CNN for use on the original MNIST dataset. Note the use of pooling layers, full-connected layers, and convolutional layers. Image provided from [1].

All of the previously mentioned mathematics apply broadly to ANNs but are slightly different in CNNs. These types of networks are often used in image classification, as shown later in this paper. Rather than summing a flat layer of nodes, they aim to use spatial information to detect features from 2D images. Each layer is then not a set of nodes, but a set of images known feature maps. Feature maps are calculated by convolving a filter (similar to weights) of a certain size (known as the kernel) with the previous layer to obtain a new image. These resulting images are then pooled to reduce dimensionality, by averaging (average pooling) or calculating the maximum (max pooling) over a certain region of the image. Eventually, the images are reduced down and fed into a fully connected NN, where classification can occur as before. An example of CNN can be seen in Figure 1, for additional clarity.

III. Algorithm Implementation and Development

In this work, the fashion dataset from the Modified National Institute of Standards and Technology (MNIST) will be used to train and test two different styles of NNs. This dataset contains 70,000 28x28 pixel images of 10 different clothing types, as shown in Figure 2. 55,000 of these images were used for training, 5,000 for validation, and 10,000 for testing of each network. This data was first loaded in, normalized, and split into these sets before training the NNs (lines 1-30).

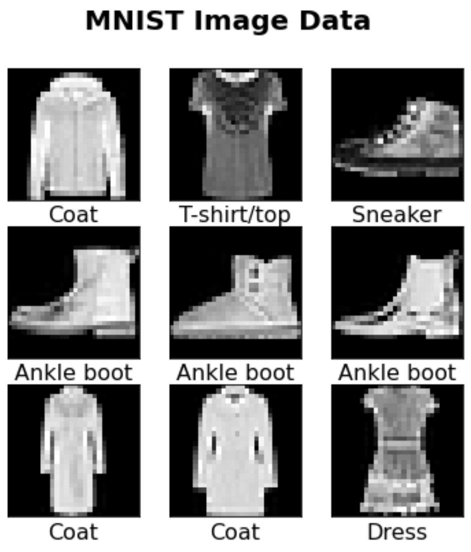

Figure 2. Example images from the fashion MNIST dataset. Each image is 28x28 pixels and represents a different article of clothing. Some images have clear similarities (shoes, shirts, etc.) while others are fairly easy to differentiate (bags, not shown).

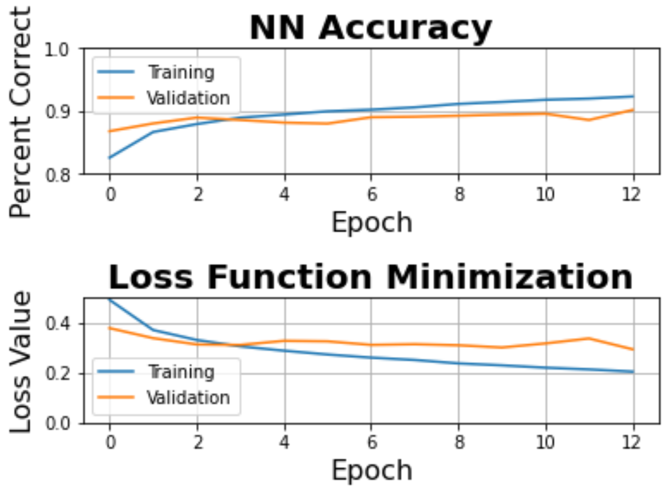

Figure 3. Accuracy and loss values over training epochs for fully connected NN. Note that overfitting usually begins when the training accuracy is greater than the validation accuracy (or the loss is less). Learning rate may also be to blame for the quick leveling off and relative instability of the loss and accuracy in the validation data.

The first NN to be tested was a fully connected NN with only 3 layers, a flattened input layer (28x28), a layer of 256 nodes and a ReLU activation function, and an output layer with a softmax activation (lines 44-52). Note that random seeds were set for both NNs to provide consistency in the presented results. This model was then compiled using cross-entropy loss and the Adam optimizer, which uses similar elements to those used in stochastic gradient descent (lines 54-57). The network was then trained and validated by running through the corresponding data 13 times (lines 59-61). Each run though the data is also known as an epoch. After the model rose above 90% accuracy on the validation data, it was then analyzed using the test data, and training history were examined (lines 63-100).

The second NN tested was a CNN with 2 convolutional layers, 2 pooling layers, and 2 fully connected layers. Before training, the data was reorganized slightly for proper implementation, by adding a third empty axis (lines 102-105). Them, the following network architecture was assembled: a convolutional layer of 32 feature maps with a 3x3 kernel, an average pooling layer, a convolutional layer of 16 feature maps with a 3x3 kernel, another average pooling layer, a flattening of the data into 128 fully connected neurons with ReLU activation, and an output layer with softmax activation (lines 110-124). As before, this network was then trained, tested, and had its training information printed for further analysis (lines 126-175).

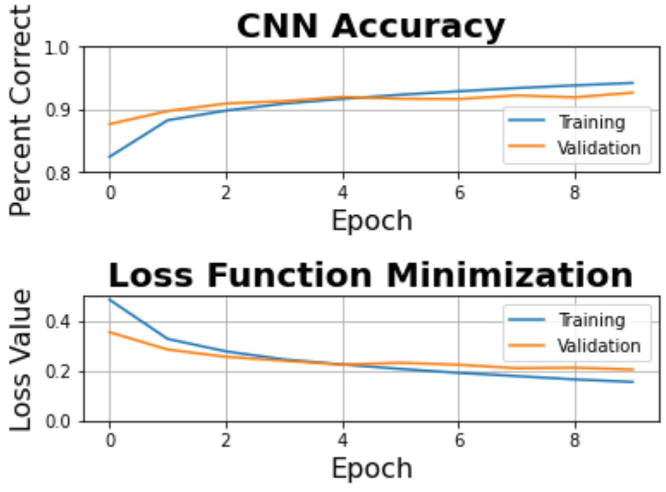

Figure 4. Accuracy and loss values over training epochs for CNN. While overfitting appears in this network too, less instability of the validation accuracy and loss occurs. Overall, stronger performance is seen in fewer epochs, highlighting the strength of CNNs in image classification.

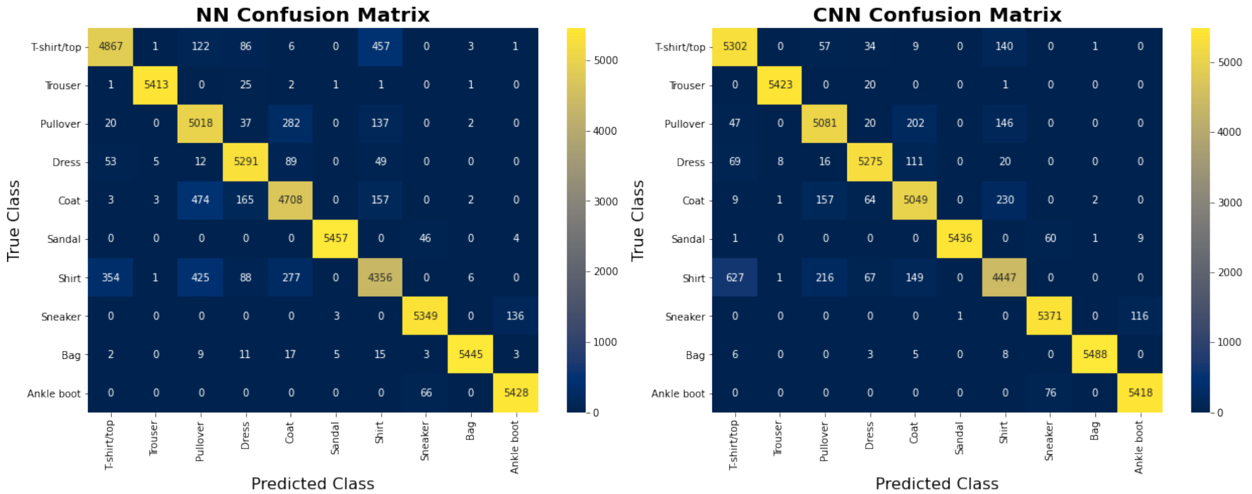

Figure 5. Side-by-side confusion matrices for the fully connected NN (left) and CNN (right) on training data. The diagonal (yellow) shows correctly classified data, where some differences between the models are apparent. Both appeared to struggle with correctly differentiating shirts, coats, pullovers, and t-shirts, though the CNN clearly had a better performance overall.

IV. Computational Results

Following the training and testing of these NNs, we can see several interesting components in our data. In Figure 3 we can see the general performance of the fully-connected NN through each epoch, as shown by the loss and accuracy. While the accuracy rose to 90.10% for the validation data, when tested after this training it only achieved 88.97% accuracy on the test data. This is most likely due to minor overfitting of the model, which can be seen when the training data accuracy and loss crosses the validation data values. This indicates that our model is beginning to sacrifice performance on new data points, by increasing its fit to the training data. Despite this, we can see that the accuracy of the validation data still continues to rise, meaning that the overfit is not too dramatic.

For the CNN, Figure 4 illustrates the performance of the model over time. As before, we see some signs of overfitting, but will still increasing validation data performance. The final performance on the testing data was found to be 92.00% (with validation at 92.62%), clearly outperforming the fully-connected NN in fewer epochs. This is primarily due to the whole design behind CNNs, which are often used in image recognition models. Because CNNs maintain some spatial information between pixels, they are often better able to detect important features within the kernel region than other techniques.

The generally low performance of both NNs through this task was somewhat interesting. This may be due in part to the challenging data they were both given. The fashion MNIST dataset was actually created to challenge neural networks that were performing too well on the original MNIST dataset. We can see in Figure 5 that both networks tended to misclassify shirts as coats, pullovers, or t-shirts, most likely due to their similar appearances. Besides this, neither of these networks are particularly deep or complex, due to computational limitations of the device used. Likewise, there are a number of optimizations that could be made to the hyperparameters of each network but would require additional testing and time.

V. Summary and Conclusions

In this assignment, we were able to explore how NNs learn and perform on large datasets, and how they can be optimized and analyzed afterward. With this, we were able to see how different styles of NNs are optimized for image classification, and how limitations like overfitting and can decrease performance. Ultimately, we demonstrated how to leverage the mathematical background behind NNs, and how they can be used in meaningful ways for data analysis.

References

[1] Sumit Saha, *Towards Data Science.* https://towardsdatascience.com/.

Appendix A. Functions Used

`tf.keras.models.Sequential` – Uses the Keras Sequential model API for creating a neural network.

`tf.keras.layers.Flatten` – Converts an input layer into a single dimension vector.

`tf.keras.layers.Dense` – Creates a fully-connected layer with a specified number of nodes and additional parameters (activation, bias, etc.)

`model.compile` – Configures a given model using a given optimizer and additional parameters (loss function, metrics to observe).

`model.fit` – Trains a model on given data over a specified number of epochs.

`model.evaluate` – Runs given data through a trained model and evaluates metrics.

`model.predict_classes` – Outputs predicted classes from a trained model given input data.

`tf.keras.layers.Conv2D` – Creates a 2D convolutional layer with a specified number of filters and kernel sizes and additional parameters (strides, padding, etc.)

`tf.keras.layers.AveragePooling2D` – Creates an average pooling layer with specified dimensions and additional parameters.

`confusion_matrix` – Given true and predicted labels, computes a confusion matrix of correctly and incorrectly categorize classes.

`sn.heatmap` – Plots given data as a colored matrix (heatmap) with values in each cell.

Appendix B. Python Code

```python
# Importing necessary packages
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sn
from sklearn.metrics import confusion_matrix
from functools import partial

# Loading in fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

# Setting font sizes for plotting
axis_font = {'size':'16'}
title_font = {'weight':'bold','size':'20'}

# Creating and normalizing validation set
y_valid = y_train_full[:5000]
X_valid = X_train_full[:5000]/255.0

# Normalizing training and testing datasets
y_train = y_train_full[5000:]
X_train = X_train_full[5000:]/255.0

X_test = X_test/255.0

# Setting list of class names (instead of numbers)
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Plotting the first 9 images and classes
# Of the fashion MNIST dataset
plt.figure(figsize=(6,6))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(X_train[i], cmap="gray")
    plt.xlabel(class_names[y_train[i]], **axis_font)
plt.suptitle('MNIST Image Data', **title_font)
plt.show()

# Setting seed for RNG
tf.random.set_seed(1)

# Creating fully-connected neural network model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28,28]),
    tf.keras.layers.Dense(256, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax"),
])

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

# Training model and reporting/recording accuracy information
history = model.fit(X_train, y_train,
          epochs=13, validation_data=(X_valid, y_valid))

```

```python
# Finding predicted classes for training data based on model
y_pred = model.predict_classes(X_train)

# Creating confusion matrix and dataframe
conf_mat_train = confusion_matrix(y_train, y_pred)
df_conf_mat = pd.DataFrame(conf_mat_train, index=[i for i in class_names],
                  columns=[i for i in class_names])

# Plotting confusion matrix
plt.figure(figsize=(10,7))
sn.heatmap(df_conf_mat, annot=True, fmt='d', cmap='cividis')
plt.xticks(rotation=90)
plt.ylabel('True Class', **axis_font)
plt.xlabel('Predicted Class', **axis_font)
plt.title('NN Confusion Matrix', **title_font)
plt.show()

# Plotting training and validation accuracies
fig, (ax1, ax2) = plt.subplots(2)

ax1.plot(history.history['accuracy'], label='Training')
ax1.plot(history.history['val_accuracy'], label='Validation')
ax1.grid(True)
ax1.set_ylabel('Percent Correct', **axis_font)
ax1.set_xlabel('Epoch', **axis_font)
ax1.set_title('NN Accuracy', **title_font)
ax1.legend()
ax1.set_ylim(0.8,1)

ax2.plot(history.history['loss'], label='Training')
ax2.plot(history.history['val_loss'], label='Validation')
ax2.grid(True)
ax2.set_ylabel('Loss Value', **axis_font)
ax2.set_xlabel('Epoch', **axis_font)
ax2.set_title('Loss Function Minimization', **title_font)
ax2.legend()
ax2.set_ylim(0,0.5)
plt.subplots_adjust(hspace=1)

# Adding additional axis to data for CNN processing
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]

# Setting a seed for RNG
tf.random.set_seed(1)

# Creating default convolutional and pooling layers
conv_layer = partial(tf.keras.layers.Conv2D, padding="valid",
                     activation="relu", kernel_initializer='he_uniform')

# Creating CNN model
model = tf.keras.models.Sequential([
    conv_layer(32,3,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2, strides=2),
    conv_layer(16,3),
    tf.keras.layers.AveragePooling2D(2, strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu",
                          kernel_initializer='he_uniform'),
    tf.keras.layers.Dense(10, activation="softmax"),
])

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

# Training model and reporting/recording accuracy information
history = model.fit(X_train, y_train,
          epochs=10, validation_data=(X_valid, y_valid))

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

# Finding predicted classes for training data based on model
y_pred = model.predict_classes(X_train)

# Creating confusion matrix and dataframe
conf_mat_train = confusion_matrix(y_train, y_pred)
df_conf_mat = pd.DataFrame(conf_mat_train, index=[i for i in class_names],
                  columns=[i for i in class_names])

# Plotting confusion matrix
plt.figure(figsize=(10,7))
sn.heatmap(df_conf_mat, annot=True, fmt='d', cmap='cividis')
plt.xticks(rotation=90)
plt.ylabel('True Class', **axis_font)
plt.xlabel('Predicted Class', **axis_font)
plt.title('CNN Confusion Matrix', **title_font)
plt.show()

# Plotting training and validation accuracies
fig, (ax1, ax2) = plt.subplots(2)

ax1.plot(history.history['accuracy'], label='Training')
ax1.plot(history.history['val_accuracy'], label='Validation')
ax1.grid(True)
ax1.set_ylabel('Percent Correct', **axis_font)
ax1.set_xlabel('Epoch', **axis_font)
ax1.set_title('CNN Accuracy', **title_font)
ax1.legend()
ax1.set_ylim(0.8,1)

ax2.plot(history.history['loss'], label='Training')
ax2.plot(history.history['val_loss'], label='Validation')
ax2.grid(True)
ax2.set_ylabel('Loss Value', **axis_font)
ax2.set_xlabel('Epoch', **axis_font)
ax2.set_title('Loss Function Minimization', **title_font)
ax2.legend()
ax2.set_ylim(0,0.5)
plt.subplots_adjust(hspace=1)
```