AMATH 482 Winter 2020
Homework 5: Neural Networks for Classifying Fashion MNIST

Ozan Erdal

March 13, 2020

Introduction and Overview

Machine Learning models can be trained with and subsequently tested to determine the accuracy of the model. By building up established data sets for standard testing of *ML* by everyone in the *ML* community. One data set that became an early benchmark for use with *Neural Networks* (*NN*s), a type of *ML* system. Two types of Neural Networks are used in this experiment, and the benefits of using them in conjunction are discussed. These are the *Fully Connected Neural Network* and the *Convolutional Neural Network*. Additionally a famous existing model, **LeNet-5**, is used as a starting point from which modifications are made to refine the model. Several python libraries are used to aid with the learning process and the data set ships with the machine learning library. These libraries and the data set are explained in detail at various points below.

Theoretical Background

The data used for this experiment is a harder version of the old industry standard, which used to be 70,000 28×28 grayscale images of handwritten numbers, called *MNIST*. As *NN*s became more advanced, this data started to become "too easy" for the models, necessitating the use of a more complicated data set. The new set, called *Fashion MNIST* was the same dimensions, but rather than being of numbers, they were images of 10 different clothing types, each corresponding to a number similar to the original *MNIST* format. A table containing the labels as well as an example of each clothing category is shown below.

Label	**Description**
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

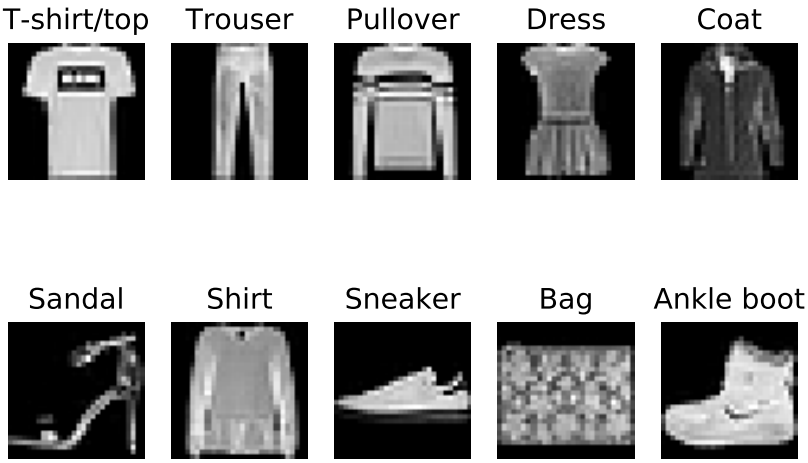

Figure 1: Sample images from Fashion MNIST.

Several times throughout this experiment, an API with support for python called **tensorflow** is used to construct and work with the *NN* model. A library called **Keras** is native to the current version of tensorflow and the API comes with the *MNIST* data set already installed. To use this data with a *NN*, it must be split into 3 groups, a *training set*, a *validation set*, and a *testing set*. This varies from our previous application of *ML* in the addition of a validation set to compare the training set against as it learns, effectively an intermediary testing set, used for refining the model. **It is essential that this set is different from the testing set.** The last ingredient for this model is a list of labels corresponding to which each image represents. From this, a *NN* can be constructed.
Based off of an actual human neuron, a neural net is composed of inputs and outputs, the latter of which will "fire" according to a given threshold. In actual neurons, this is based on an electrical impulse that propagates through a neuron and will often trigger other neurons in the process. In a neural net, a neuron will trigger according to an *activation function* which similarly serves as a threshold. *Sigmoids* are often chosen as the class of functions for thresholds since step-functions result in problems when differentiating. Some examples of sigmoids are shown in the figure below.

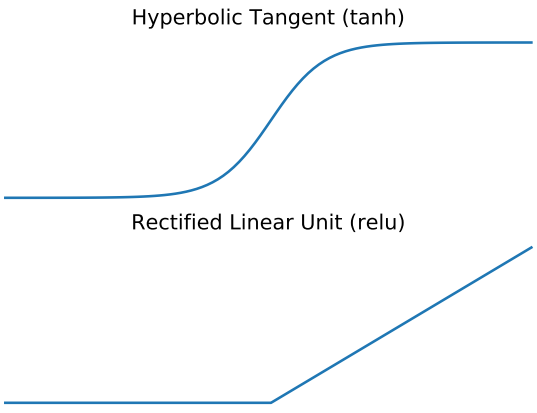

Figure 2: Two commonly used activation functions.

After this process is complete, the output values are converted to a probability distribution using a **softmax** function. This function is also called a **normalized exponential** function. The formula for it is shown below, and a graph of it is shown at the bottom of this page.

$$\sigma(x_i) = \frac{e^{x_i}}{\Sigma_k e^{x_k}} \tag{1}$$

Part 1: Fully Connected Neural Network

This type of *NN* is characterized by dense hidden layers in the model. Each neuron is connected to each every neuron in the adjacent layers, with each connection represented by an edge weight. Before the process can begin, the data must also be flattened.

Part 2: Convolutional Neural Network - *CNN*

This type of *NN* is characterized by a series of *convolutional* layers, usually separated by *subsampling* layers, sometimes in an alternating pattern. After these convolutions and subsamples, a *CNN* will also have several *fully connected* layers, the structure of which was just mentioned.
The convolutional layers work by sliding a windowed view of the data that is then multiplied by a *feature map*. The feature map is determined by the features deemed important to the model, and the kernel of it stores the factors for the convolution. The feature layers or feature maps are also called filters. As the window slides across the data, it performs a mathematical operation that is stored and passed to the next layer. This process should theoretically reduce the size of the data set since the center of the window would go beyond the bounds of the data array unless the data is "padded" with 0 valued entries.
The subsampling layers work by attempting to reduce the complexity of the data by performing an operation on a cluster of points similar to the convolution layer. Unlike the former, the subsampling layers do not have feature maps, and they perform a well defined operation such as calculating the average or the maximum value of the subset of data.
For both of these types of layers, the size of the window can be specified and directly or indirectly, the sliding step or *stride* can be controlled. The *learning rate* and the number of *epochs* can also be adjusted to control the speed with which the model learns as well as the number of iterations of the learning process, respectively.

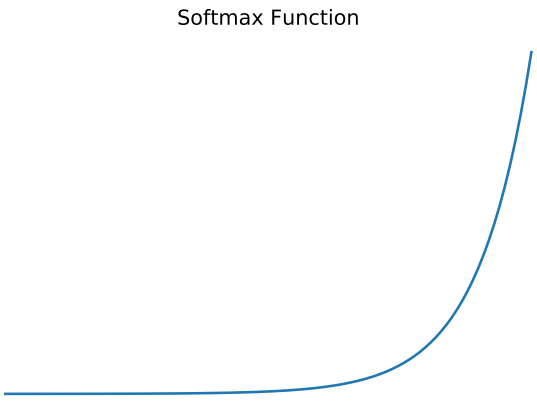

Figure 3: Softmax function from 0 to 10.

Algorithm Implementation and Development

The data was split into the three groups. The training set had 55,000 images, the validation set had 5,000 images, and the testing set had 10,000 images. Since they were not ordered, the first 55,000 were selected as training images and the following 5,000 as validation images.
The general structure of the two parts were described in theory already, but the specifics on the implementation as well as the chosen techniques and changes are described at length below.
Both of these models were then improved, and the techniques for doing so as well as the final technique that provided relatively accurate results on the validation set were used to predict the data in the testing set.

Part 1

For the fully connected *NN*, the initial structure was relatively simple: 2 hidden layers that use the *relu* activation function going from 300 neurons to 100. *Softmax* was then applied, reducing the net to the 10 output neurons. The learning rate is relatively low at 0.0001. In all experiments, 5 epochs were used since the validation set accuracy would level out consistently by this point.

Part 2

For the *CNN*, the initial structure was based off of the LeNet-5 model due to its existing success with the *MNIST* data set. This is a multi-layer model with alternating convolution and subsampling layers. The subsampling layers use an *average pooling* scheme, which works as the name implies. It also uses a *tanh* activation function. The process is as follows:

1. Convolution layer with 6 filters of size 5 with padding.
2. Pooling layer of size 2.
3. Convolution layer with 16 filters of size 5 without padding.
4. Pooling layer of size 2.
5. Convolution layer with 120 filters of size 5 without padding.
6. Fully connected layer with 84 neurons using tanh.
7. Fully connected layer with 10 neurons using softmax.

The learning rate is also set at 0.0001.

Results and Improvements

For the remainder of this paper, the models for part 1 and part 2 will be referred to as model 1 and model 2, respectively.
Using the starting models, the accuracy of each on the validation set were as follows:

Model 1	**Model 2**
87.6%	85.26%

There were many available values to adjust that would often change the success of the model. Sometimes a change would not make a significant different, but when combined with other changes or techniques would provide a boost in the success of the model.
Complicating the model would increase the runtime of the training, so the complexity was important to consider as repeated testing was required to refine the model. Using an overly complicated model would detract from testing time and limit the iterations of the model as a whole.

Depth of Network

By adding more layers, complexity was added and the success was improved. Going from $2 \rightarrow 5$ *relu* layers, the success rate increases marginally at a slight increase in computational time.

Width of Layers

By taking this improved model and doubling the width of each layer, the number of neurons at each layer is doubled, resulting in a significant increase in the computation time with, one again, a slight increase in success.

Learning Rate

The learning rate was adjusted last as it seemed to be the most independent aspect of the model, being set after the model itself was constructed. The best resulting model was run at a learning rates of 0.0005, 0.0001, 0.00005, and 0.00001.

Regularization

This is an optional parameter that is used to combat overfitting, and uses the *L2 norm* to reduce the absolute value of the weights. By using a regularizer on each layer and increasing the parameter in the softmax layer, the success was increased to 89.22% for model 1.

Activation Function

Comparing the success of a *tanh* activation function to that of a *relu* function was used to determine the better model. Additionally, an alternating pattern was also tested for these activation functions, swapping at each layer.

Optimizer

Keras has several optimizers to choose from. The optimizer selected was not done so for a particular reason, so by choosing other optimizers, it is likely that the success will be different for a different optimizer. The initial optimizer was *Adam*, but the *Adagrad*, *Adadelta*, *Adamax*, and *Nadam* optimizers were also experimented with. Out of these, only the *Nadam* optimizer yielded better results, the rest of which resulted in very poor success rates while posting significantly higher runtimes.

Number of Filters

The more or less filters used in the convolution layers is analogous to adjusting the number of features in non-*NN ML*. Adjusting this is entirely trial and error based since these filters are computed behind-the-scenes.

Filter Size

Changing the size of the filter is a technique that has been employed in other contexts like time-frequency analysis to adjust the size of the features being searched for. This number was played around with, although there was not too much room for adjusting this value.

Stride

The step size serves to speed up the computation by lowering the overlap between subsequent convolutions. For the pooling layers, the stride and size are controlled simultaneously. The stride for each dimension can be specified, and unequal strides were experimented with.

Stopping Early

An additional technique that will increase the success of the model is stopping early to prevent overfitting. This was too complicated to implement for the scale of this experiment, but it is hypothesized that this is a useful technique.

Summary and Conclusions

The results of the optimized models are shown in the table below, and the final confusion matrices are also shown below. The off-diagonal elements represent mistakes. A graph of the accuracy through epochs is also shown.

Model 1	Model 2
87.6%	85.26%

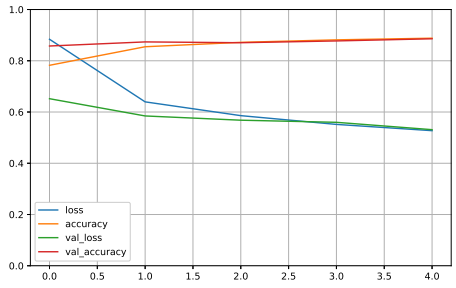

Figure 4: Rates across epochs.

	0	1	2	3	4	5	6	7	8	9
0	794	4	38	45	9	1	92	0	17	0
1	1	959	10	21	7	0	0	0	2	0
2	7	1	847	8	119	0	15	0	3	0
3	38	22	29	838	54	0	15	0	4	0
4	0	0	184	23	786	0	4	0	3	0
5	0	0	0	1	0	850	0	88	4	57
6	170	3	270	38	256	0	242	0	21	0
7	0	0	0	0	0	8	0	889	1	102
8	1	1	33	7	9	1	3	5	939	1
9	0	0	0	0	0	2	0	25	1	972

Figure 5: Best model confusion matrix

Appendix A

load_data()

Splits data set into training and testing sets and makes array of the labels containing the true values of each image.

model.compile()

Builds model in preparation for training. Takes in options for the loss function and optimizer as well as output information.

model.fit()

Trains the model to the training set and compares it to the validation set. Runs for a specified number of epochs.

Appendix B

Imports

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

Plotting Functions

```
t = np.arange(-5.0, 5.0, 0.1)
z = np.zeros(t.size)

plt.figure()
plt.subplot(2,1,1)
plt.plot(t, np.tanh(t))
plt.title('Hyperbolic Tangent (tanh)')
plt.axis('off')

plt.subplot(2,1,2)
plt.plot(t, np.amax([t,z], axis=0))
plt.title('Rectified Linear Unit (relu)')
plt.axis('off')
plt.savefig('activation_funs.eps')

t = np.arange(0.0, 10.0, 0.1)

plt.figure()
plt.plot(t, np.exp(t)/sum(np.exp(t)))
plt.axis('off')
plt.title('Softmax Function')
plt.savefig('softmax_fun.eps')
```

Preparing the 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:]

# only for CNN
X_valid = X_valid[..., np.newaxis]
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

Fully Connected NN

```
from functools import partial

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]),
    dense_layer(525),
    dense_layer(500),
    dense_layer(475),
    dense_layer(450),
    dense_layer(425),
    dense_layer(400),
    dense_layer(375),
    dense_layer(350),
    dense_layer(325),
    dense_layer(300),
    dense_layer(275),
    dense_layer(250),
    dense_layer(225),
    dense_layer(200),
    dense_layer(175),
    dense_layer(150),
    dense_layer(125),
    dense_layer(100),
    dense_layer(10,activation='softmax', kernel_regularizer=tf.keras.regularizers
    .l2(0.001))
])
```

CNN

```
from functools import partial

conv_layer = partial(tf.keras.layers.Conv2D, padding='valid', activation='tanh')
dense_layer = partial(tf.keras.layers.Dense, activation='tanh',
    kernel_regularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    conv_layer(6, 2, padding='same', input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    conv_layer(16, 2),
    tf.keras.layers.AveragePooling2D(2),
    conv_layer(120, 2),
    tf.keras.layers.Flatten(),
    dense_layer(84),
    dense_layer(42),
```

```
    dense_layer(21),
    dense_layer(10, activation='softmax', kernel_regularizer=tf.keras.
   regularizers.l2(0.001))
])
```

Predict

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