Homework 5: Neural Networks for Classifying Fashion MNIST

AMATH 482: Computational Methods for Data Analysis

Rosemichelle Marzan

March 13, 2020

Abstract – **A fully connected neural network and a convolutional neural network were trained to classify items from the Fashion-MNIST dataset. The fully connected neural network correctly identified 88.06% of the items and the convolutional neural network correctly classified 91.68% of the items in the test set. The goal of this study is to experimentally adjust each network's hyperparameters to gain a basic understanding of their effect on the network's validation performance and to find the optimum parameters for training the final model.**

1 Introduction and Overview

In this study, we built and trained two neural network classifiers, a fully connected neural network (FCNN) and a convolutional neural network (CNN), for classifying a Modified National Institute of Standards and Technology database (MNIST) data set of 10 different classes of fashion items. The data set contains 60,000 28x28 training images in the array `X_train_full`, 5,000 of which were used for validation data, and 10,000 test images in the array `X_test`. The labels for both data sets are contained in the vectors `y_train_full` and `y_test`. The values in the vectors are numbers from 0 to 9 which correspond to one of the 10 classes of fashion items: 0 for t-shirts/tops, 1 for trousers, 2 for pullovers, 3 for dresses, 4 for coats, 5 for sandals, 6 for shirts, 7 for sneakers, 8 for bags, and 9 for ankle boots.

The neural networks were built and trained in Python using the Keras application programming interface (API) within the TensorFlow open source machine learning library. The goal is to build a neural network by experimenting with different hyperparameters to yield the highest validation data accuracy without overfitting the training data. Some of these parameters for part I included: the depth of the network, the width of each layer, the optimizer and its learning rate, the regularization constant, and the activation functions. In part II, the same experiment was repeated but using a convolutional neural network, where in addition to the parameters enumerated above, the number of filters in the convolutional layers, the kernel sizes, the strides, the padding options, and the pool layer sizes were adjusted. Both networks were tested with the test data and evaluated on its accuracy. The confusion matrix was also included in the results to indicate which classes had the most mistakes and what the network mistook them for.

2 Theoretical Background

By definition, a fully connected neural network (FCNN) is a network in which each layer receives an input from all the neurons (units) from the previous layers; thus, its hidden layers are described as *dense*. A convolutional neural network (CNN) is generally more complex than a FCNN and is performs much better than FCNN at classifying images of objects due to its ability to assign different weights to different areas, or features, within the image. A CNN can also have dense layers which come after the layer-flattening step.

When building and training neural networks, we must be conscious of overfitting the training data and minimize it accordingly. As one can guess, overfitting occurs when the model models the training data too well, such that it fails to be generalizable enough to classify new data, such as those in the test set. Overfitting is identified by a training accuracy that is much larger than the validation accuracy, and/or training loss that is much lower than the validation loss. Both indicate that the model performs better with the training data than it does with the validation data. Because of this, it is important to ensure that the neural network is given a validation data set to work with. Validation data is similar to the test data in that it gives the network engineer an unbiased estimate of how each model performs as a result of adjusting certain hyperparameters, or the parameters that describe the model.

3 Algorithm Implementation and Development

3.1 Pre-processing the data

The Fashion-MNIST data was loaded into the Jupyter notebook by calling the Keras Fashion-MNIST data set. This returns 2 arrays containing the training and testing images, and 2 vectors containing the arrays' respective labels. The first 5,000 images within the training data set were allocated towards the validation data set. The remaining 55,000 images were used for training. The label vectors were also adjusted accordingly. To convert the images from their original unit8 format, the training, validation, and test data were divided by 255.0.

3.2 Part I: Fully Connected Neural Network

The best FCNN model discovered in this study is designed as follows:

- Flattened input layer; input shape: 28x28
- Hidden layer 1: 784 units
- Hidden layer 2: 32 units
- Output layer: 10 units (for 10 classes)

The `partial` function is used to generalize layer functions. All layers use a ReLU (rectified linear unit) activation function, except the output which must always use a Softmax activation function.

The `compile` method defines the options for training. The loss function is the `sparse_categorical_crossentropy`; the optimizer is the `Adam` with a learning rate of 0.001; the metric used for evaluating the model is *accuracy*. The model is then trained for 10 epochs.

3.3 Part II: Convolutional Neural Network

To create a CNN, TensorFlow requires the input shape to have at least 3 dimensions: image height, image width, # of color channels. Thus, a new axis is added to the training, validation, and test data via the `numpy` function, `np.newaxis`.

The CNN shares a few similarities to the FCNN, namely the use of `partial` layers, a Softmax activation function for the output layer, and the same settings for the `compile` method. `partial` was used to generalize the convolution layers by setting the kernel initializer to He uniform variance scaling and a filter size of 3x3. All of the dense layers after layer-flattening also uses a kernel L2 regularizer with penalty constant 0.001.

The best CNN model discovered in this study is designed as follows:

- 2D convolution layer 1: 32 filters, input shape = 28x28x1
- Average pooling layer 1: pool size = 2x2
- 2D convolution layer 2: 64 filters
- Maximum pooling layer: pool size = 2x2
- 2D convolution layer 3: 128 filters

- (Flatten layer)
- Hidden layer: 32 units
- Output layer: 10 units

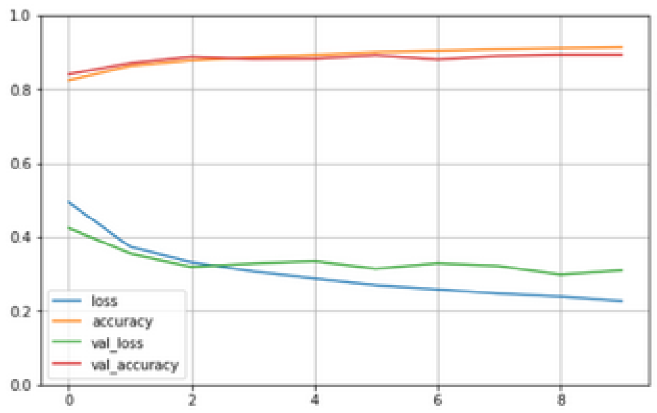
4 Computational Results

The final training models were chosen on the highest validation accuracy with minimal overfitting. Both models show a small degree of overfitting as evidenced by the continued decrease of the training loss and divergence from the validation loss, which remained stable after epoch 2 (Fig. 1 and Fig. 2). **The final FCNN model achieved a validation accuracy of 89.32%, and a testing accuracy of 88.06%. The final CNN training model achieved a validation accuracy of 92.06%, and a testing accuracy of 91.68%.**

4.1 Part I: Fully Connected Neural Network

In tuning the parameters for the FCNN model, I noticed that creating a deeper network ($>$ 3 hidden layers) worsened the validation accuracy. As for layer depth, it appears that using 784 (size of the image) $\geq N \geq 30$ units in a network with a single hidden layer all eventually converged to a validation accuracy of approximately 88-89%. Most of the model versions created by adjusting the parameters had a validation accuracy that converged to 88-89% accuracy within 10 epochs, typically around as early as the 4th epoch. Running more epochs, such 20 and 30, also confirmed the idea that a convergence was reached early on in training since the validation accuracy is stable at 88-89%.

This fast convergence led me to investigate if the network optimizer may have gotten stuck in a local minimum. Considering that the Adam optimizer is regarded as one of the "best" available Keras optimizers in that it converges quickly and uses little RAM compared to other options, I first adjusted the learning rate by increasing and then decreasing the learning rate by one order of magnitude. Increasing the learning rate sped up the training time, while decreasing slowed down the training time, yet both still converged to a validation accuracy value of approximately 86-87%. I then tested the SGD, or Stochastic Gradient Descent, optimizer, another commonly used optimizer that is still outperformed by Adam but offers the option to define a momentum constant in addition to the learning rate. In defining a momentum constant, perhaps



```
[[4855    1   36   45    8    1  584    0   13    0]
 [   2 5406    0   27    3    0    5    0    1    0]
 [  42    0 4102   31  862    0  450    0    9    0]
 [ 145   10   12 5088  114    0  123    0    7    0]
 [   6    1  150  188 4907    1  234    0   25    0]
 [   0    0    0    0    0 5424    0   73    4    6]
 [ 424    4  166   54  397    0 4444    0   18    0]
 [   0    0    0    0    0   16    0 5415    4   53]
 [   0    0    2    5    4    1   16    0 5482    0]
 [   0    0    0    0    0   17    0  200    0 5277]]
```

Figure 1: Fully connected neural network: accuracies and losses (left) and training set confusion matrix (right). Plot x-axis is the epoch #. Confusion matrix columns are each fashion item and the rows are what the model predicted them as. The values indicate how many times the model made each prediction. Correct predictions are on the diagonal.

the network could avoid the local minimum I thought existed. Nevertheless, at the same learning rate and a momentum rate of 0.9, the SGD yielded the same results as the Adam.

Since there is minimal overfitting based only on the results of the losses, I chose to exclude a regularization parameter in my model. Adding one, even with a minimum penalty constant such as 0.0001 causes the validation accuracy to dip slightly.

Fig. 1, right shows the confusion matrix for the FCNN. Most of the errors the model made were in mistaking a coat for a pullover (column 4, row 2 is read as "actual: coat; predicted: pullover") 862 times, a shirt mistaken for a t-shirt/top 584 times, and a shirt mistaken for a pullover 450 times.

4.2 Part II: Convolutional Neural Network

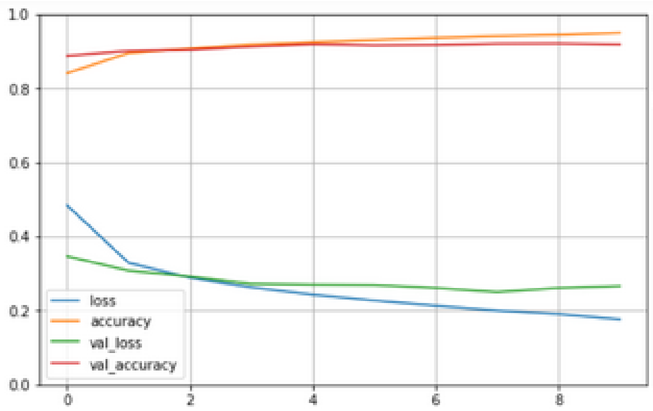
The final CNN training model performs slightly better than the FCNN which is expected because CNN are inherently structured to be better at feature processing of images. In addition, it expands on the capabilities of the FCNN by having a few fully connected layers of its own.

The CNN structure is a bit more complex than the FCNN. The number of layers, filter sizes, strides, and pool sizes were primarily informed by whether or not I encountered an error after adjusting the parameters. There appears to be a mathematical relationship between the number of filters and the filter size and such parameters between the layers that would require further research to determine ideal values.

I also adapted several techniques from other CNN templates online, such as the use of a kernel initializer which inform the initial random weights of each Keras layer to help the model converge faster. Among the available initializers, I tried the `he_uniform`, `he_normal`, and `random_uniform` and the He uniform initalizer produced the best validation accuracies.

Most other adjustments to the hyperparameters were a mix of random trial-and-error and researching ways to optimize the performance of a neural network as well as other examples provided online. For most of the hyperparameters adjusted, there is not much of a pattern to be gleaned by increasing and decreasing values, which further complicated the optimization process. If I were to perform this experiment again in a longer time frame, I would systematically test each hyperparameter with different sets of values. The consensus from internet results for ways to optimize neural networks suggest that there does not appear to be a systematic order for which parameters adjust the network the most and how, and their effects are also largely dependent on the size of the network and the training data set.

Fig. 2 depicts the accuracies and losses of the CNN model, as well as its confusion matrix. Similar to the FCNN, the CNN appears to mistake shirts as t-shirts/tops but now almost half as many times as the CNN



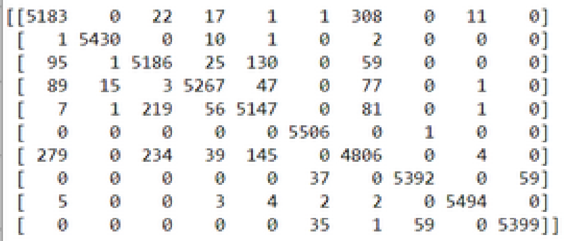

```
[[5183    0   22   17    1    1  308    0   11    0]
 [   1 5430    0   10    1    0    2    0    0    0]
 [  95    1 5186   25  130    0   59    0    0    0]
 [  89   15    3 5267   47    0   77    0    1    0]
 [   7    1  219   56 5147    0   81    0    1    0]
 [   0    0    0    0    0 5506    0    1    0    0]
 [ 279    0  234   39  145    0 4806    0    4    0]
 [   0    0    0    0    0   37    0 5392    0   59]
 [   5    0    0    3    4    2    2    0 5494    0]
 [   0    0    0    0    0   35    1   59    0 5399]]
```

Figure 2: Convolutional neural network: accuracies and losses (left) and training set confusion matrix (right). Plot x-axis is the epoch #. Confusion matrix columns are each fashion item and the rows are what the model predicted them as. The values indicate how many times the model made each prediction. Correct predictions are on the diagonal.

(308 times). The second most common error is the reverse, in which t-shirs/tops are mistaken for shirts (279 times).

5 Summary and Conclusions

In this study, two kinds of neural network architectures were built, trained, tested, and compared: a fully connected neural network and a convolutional neural network. After experimenting with several hyperparameters in both networks, including but not limited to the depth of the network, layer width, activation function, optimizer type, regularization, and initialization, the optimum models found achieved testing accuracies of 88.06% and 91.68% for the FCNN and CNN, respectively. These results match how we expect a CNN to perform compared to a FCNN considering that it is better designed for image feature processing by convolving a filter across the the image. Both networks appear to converge early in the training process and also demonstrate a minimal amount of overfitting, wherein the model performs better on the training set that in the validation set. Confusion matrices for both indicate similar errors in how the fashion items are classified, but also demonstrate that such errors are significantly reduced with the CNN model.

The primary challenge of this study was understanding how each parameter affects the network's validation performance, given that the information available online that for improving network performance is limited and not straightforward, and that it is difficult to identify patterns from results from trial-and-error. More research into the literature, an understanding of successful networks, and more practice is necessary to achieve substantial gains in validation accuracy and, in turn, test accuracy.

6 Appendix A. Python functions used and brief implementation explanation

- **partial**: Reduces redundancy by creating layers that share similar properties.
- **Sequential**: Creates a stack of layers with linear neurons.
- **activation**: Activation function. Defines what output the layer produces according to a given input.
- **regularizers.l2**: L2 regularization. Defines how much penalty is applied to the layer parameter during each iteration.
- **loss**: Loss function. Determines how the model calculates the error.
- **optimizers**: The algorithm that dictates how the model converges to the solution (minimum of the loss function).
- **learning_rate**: A parameter of the optimizer that determines the step size at each iteration.
- **padding**: Indicates if input is zero padded ("same") or not ("valid").
- **kernel_initializer**: Initializes the random weights of each layer according to the given distribution function.
- **AveragePooling2D**: Outputs the average of the values within the pooling area.
- **MaxPooling2D**: Outputs the maximum value of the values within the pooling area.

7 Appendix B. Python code

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


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


# In[2]:


fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()


# ## Preprocessing the data

# In[3]:


# use 5,000 training images for validation data
# convert uint8 to floating point numbers by dividing by 255.0
# create validation set
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:]


# # Fully Connected Neural Network

# ### Building the model

# In[4]:


from functools import partial
# (use 'partial' to eliminate redundant lines)
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu")

model = tf.keras.models.Sequential([
    # flattened input layer
    tf.keras.layers.Flatten(input_shape=[28,28]),
    # hidden layer; dense = fully connected
    my_dense_layer(784),
    my_dense_layer(32),
    # output layer; 10 probabilities for 10 classes of items
    my_dense_layer(10, activation="softmax")
])

# learning rate = determines how much to change weights/biases in each iteration; very
    important
model.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
             metrics=["accuracy"])

```

```python

# ### Training the model

# In[5]:


history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid,y_valid))


# ### RESULTS

# In[6]:


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)


# # Convolutional Neural Network

# ### Preprocessing the data

# In[7]:


# Training: 55000x28x28 (TensorFlow wants 55000x28x28x1)
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]


# ### Building the model

# In[8]:


from functools import partial

my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="valid",
    kernel_initializer="he_uniform")
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regularizer=tf.
    keras.regularizers.l2(0.001))

model = tf.keras.models.Sequential([
    # no need to flatten layer for CNN
    # Conv2D args = # of filters, filter size, zero padding = same (as input size)
    my_conv_layer(32,3,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(64,3),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(128,3),

    # must flatten before going into fully connected
    tf.keras.layers.Flatten(),
    # hidden layer: how many neurons; dense = fully connected
    my_dense_layer(32),
    # output layer; 10 probabilities for 10 classes
    my_dense_layer(10, activation="softmax")
])
```

```python


# decide on options for training
# learning rate = determines how much to change weights/biases in each iteration; very
    important
model.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
             metrics=["accuracy"])


# ### Training the model

# In[9]:


history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid,y_valid))


# ### RESULTS

# In[11]:


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)


# In[ ]:
```