AMATH 482- Computational Methods in Data Analysis: Optimizing Neural Networks

Gargi Kher

March 13th, 2020

Abstract: There are many types of Machine Learning techniques, each of which is beneficial for solving a certain type of problem. Neural Networks are a subset of Machine Learning that work well for analyzing large sets of data. Inspired by the connections between neurons in our brain, these models allow for large sets of information to be processed and classified. Neural networks are often trained by information from the MNIST database, which consists of large numbered datasets. I will be discussing how I optimized a neural network using the Fashion-MNIST dataset.

Sec. I. Introduction and Overview

In this assignment, our goal was to train a neural network such that it correctly classified data from the Fashion-MNIST set. While we were given most of the code to produce this assignment, our goal was to determine how changing certain parameters influenced the accuracy of the model for a fully-connected dense neural network (Part 1) and a convolutional neural network (Part 2). These parameters included the learning rate, regularization function, activation function, width and depth of the neural network, and more. By adjusting these values, I was able to see which conditions gave me the best accuracy for my test data.

Sec. II. Theoretical Background

Neural networks are modeled after how neurons are connected to each other. An input layer of these “neurons” consisting of the dataset gets sent to an output layer of “neurons”, resulting in classification of the data. The data in the input layer can be sent to the output layer through a series of hidden layers, which can consist of more or less neurons than the input layer, which is known as having a different width. The number of hidden layers defines the depth of the neural network, and changing this depth, or even changing the width, can influence the accuracy of classification.

To give a clear explanation of a neural network, we can consider an example with one hidden layer between the input and output layers. There are three things we need to use a neural network: some data to analyze, a model to help us classify our data (represented by an equation), and a loss function that minimizes the cross-entropy loss or maximizes the probability that each data point will be classified correctly (Equation 1).

$L = -\frac{1}{N}\sum_{j=1}^{N} \ln(p_j)$ (1) Cross-entropy loss function, where p_j represents the probability of the jth data point being correct

Our data can be in the form of coordinates or vectors. Let’s say it’s a bunch of x-values $x_1, x_2, x_3 \dots$. Our goal is to output these as some y-values $y_1, y_2, y_3 \dots$. Note that the number of inputs does not have to equal the number of outputs. Every input just has an output it should be classified as. When sending this input layer to the hidden layer, each input is weighted by some constant w_n, where w_1 corresponds to x_1. The purpose of the weights is so that the sum of the weighted input values defines some threshold value z (Equation 2).

$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$ (2) Sum of weighted input values to determine threshold

To control the weights of each neuron, a bias neuron with value 1 and weight b is added to the input layer. If z is over a certain value, the neurons will "fire" to the next layer. Assuming that happens, the values of the hidden layer can be modeled by an activation function (Equation 3) σ.

$H = \sigma(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$ (3) Activation function for values of hidden layer H

σ (Equations 4 and 5) can be represented by a variety of functions. The logistic rectified linear unit (RELU) function is widely used because of its simplicity.

$y = \max(\mathrm{x}, 0)$ (4) Rectified Linear Unit activation equation

$y = \frac{1}{1+e^{-x}}$ (5) Softmax activation equation

The Softmax activation equation is typically used for final firing to the output layer. The process for finding the values of the output layer are similar to finding them for the hidden layer. The equation for the hidden layer is just put back into σ. This is complicated to write out, but we can show it in a simplified form. This equation can be written as $A_n\vec{x}_n + \vec{b}_n$, where n is the layer you're moving from (n=1 representing neurons firing from the input layer to the first hidden layer), A_n represens the weights of the input values $\vec{x}_n$ and $\vec{b}_n$ represents the biases. We can now write a new equation involving an activation function σ (Equation 6).

$\vec{y} = \sigma(A_2\sigma\left(A_1\vec{x}_1 + \vec{b}_1\right) + \vec{b}_2)$ (6) Equation for neural network with one hidden layer

This equation shows that the values of the first hidden layer ($A_1\vec{x}_1 + \vec{b}_1$) act as the x-values to be sent off to the output layer, which is why they are again put inside of the activation function with different weights (denoted by a subscript 2). Having multiple hidden layers will produce a similar equation appearing to be more nested.

The goal for a neural network is to classify the test data as accurately as possible. A problem known as overfitting often occurs when the model performs better on the training data than the test data. Imagine a plot containing many points. Some of those points represent our training data while the rest represent the test data. Overfitting would find a polynomial that goes through all of the values of the training data, but not all of the values of the test data. Thus, when this polynomial is applied to the test data, it doesn't work as well as it did for the training set. To prevent this issue, we take some of the training set and put it into a "validation" set. After we train the data, we want to apply the model to the validation data. If overfitting does not occur, the accuracy of the validation data will be higher than the accuracy of the training data, and the loss function of the validation data will be lower compared to the training data. Many factors can influence overfitting, such as the type of activation function used or the width and depth of the neural network. A parameter known as the learning rate uses gradients to minimize the optimization parameters. Changing this influences how fast the model converges and can potentially overshoot a minimum or to converge to it at all. Another function known as the regularization function adds to the cross-entropy loss function to make sure the weights applied to each neuron don't get too large and cause overfitting. There are different types of optimizers available that will utilize different training techniques based on these parameters. Finding the optimal parameters for the dataset you're working with is key to building a strong neural network.

Above was an example of a fully connected dense neural network , where each input neuron is connected to the next neuron. Another type known as the convolutional neural network (CNN) can be thought of as a grid of neurons, where multiple grids are stacked on top of each other like layers. One such model, LeNet-5, uses convolutional networks. Instead of having each neuron connected to each other, some of the neurons on the grid can “look” into a certain section of neurons on the layer below. Similar to how inputs are weighted for fully connected neural networks, a matrix of filters are applied to each neuron being sent into to the next layer. We can imagine each neuron as representing one cell on a grid. The filter matrix is generally only applied to some of these neurons, which make up a “receptive field”. The result of this becomes the value of the neuron in the next layer. To fill up all of the neurons in this new layer, the receptive field of the layer it is looking into shifts, and the filter is applied to those values. The shift is known as the “stride”. Thus, we get layers of neurons, each getting a glimpse into certain values of the layers before it, eventually leading to an output. LeNet-5 makes use of fully connected neural networks at the end, but this doesn’t apply to all CNNs. Matrices can also be “padded” or lined with zeros to include all of the neurons during filtering, which can influence the accuracy of the model. All of these parameters introduce new factors that need to be considered for optimization of the network.

Sec. III. Algorithm Implementation and Development

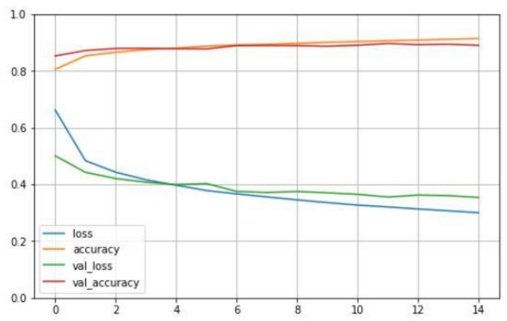
Part 1

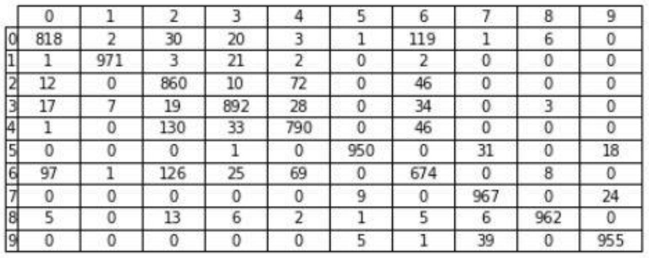
Most of the code for this part was given by the instructor. We started out by downloading all of the packages and loading all of the Fashion-MNIST data. The next step was to split up all of the loaded training data into training and validation data. Of the 60,000 total training points, we used the first 5000 for the validation data. To build the model, Keras functions from the Tensorflow package were used, where the activation function, regularization function, width and depth of the neural network were defined. An optimizer was added to analyze the accuracy of each run. Once the model was run, its training and validation loss and accuracy were plotted, and a confusion matrix was calculated for the training, and validation data. Once the model was applied to the test data, a confusion matrix for the analysis was produced (Appendix B: Lines 1-50).

Part 2

Most of the code for this part was given by the instructor. Just like the first part, we started by downloading all of the packages and loading the Fashion-MNIST data. When splitting the data into training, validation, and test matrices we had to add a third axis of size one to the matrix to work with the convolutional methods. This didn’t change anything about the data. The next part was to make a LeNet-5 model by first defining the convolutional and dense networks and arranging them so that each convolutional layer was filtered into the next layer and that result was flattened so it could be put through the dense network. The number of epochs, or times the function should run, was defined and optimized. The accuracy and loss functions for the test and validation data were then plotted to visually depict how they changed over time and determine overfitting. Lastly, the test data was analyzed and a confusion matrix depicting the number of times each object was classified in a certain category was calculated (Appendix B: Lines 51-114).

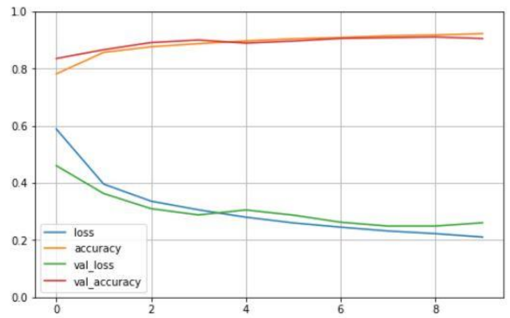
Sec. IV. Computational Results

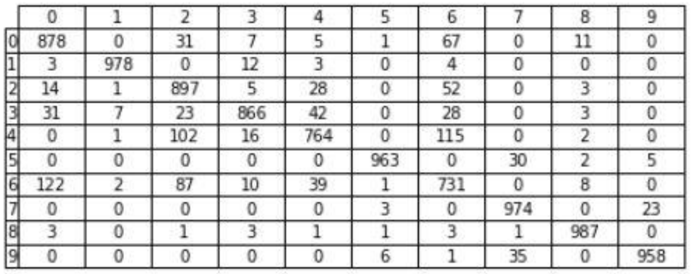




	0	1	2	3	4	5	6	7	8	9
0	818	2	30	20	3	1	119	1	6	0
1	1	971	3	21	2	0	2	0	0	0
2	12	0	860	10	72	0	46	0	0	0
3	17	7	19	892	28	0	34	0	3	0
4	1	0	130	33	790	0	46	0	0	0
5	0	0	0	1	0	950	0	31	0	18
6	97	1	126	25	69	0	674	0	8	0
7	0	0	0	0	0	9	0	967	0	24
8	5	0	13	6	2	1	5	6	962	0
9	0	0	0	0	0	5	1	39	0	955

Figure 1: Part 1, Fully Connected Dense Neural Network: A plot depicting accuracy and loss trends for the training and validation data (left) and a confusion matrix for the test data (right).





	0	1	2	3	4	5	6	7	8	9
0	878	0	31	7	5	1	67	0	11	0
1	3	978	0	12	3	0	4	0	0	0
2	14	1	897	5	28	0	52	0	3	0
3	31	7	23	866	42	0	28	0	3	0
4	0	1	102	16	764	0	115	0	2	0
5	0	0	0	0	0	963	0	30	2	5
6	122	2	87	10	39	1	731	0	8	0
7	0	0	0	0	0	3	0	974	0	23
8	3	0	1	3	1	1	3	1	987	0
9	0	0	0	0	0	6	1	35	0	958

Figure 2: Part 2, Convolutional Neural Network: A plot depicting accuracy and loss trends for the training and validation data (left) and a confusion matrix for the test data (right).

Sec. V. Summary and Conclusions

Part 1

The accuracy of this final model on my test data was 88.39% for one of the runs. When building the fully connected dense neural network, it was difficult to train it to have an accuracy over 90%. The maximum accuracy I was able to obtain was around 89%. The parameters I changed to find the best accuracy were the learning rate, the value inside the regularization function (which used the L2-norm), neural width, depth, and number of epochs.

When I increased the learning rate from 0.00001 to 0.0001, I tended to see a lot of overfitting. With this same learning rate, I found that increasing the number of epochs (tested 13, 15, and 17 epochs) tended to result in more overfitting. I then tried to decrease regularization function from 0.0001 to 0.00001 while testing against the same number of epochs (13,15,17). I found that this didn’t really affect the result as much as changing the learning rate did. Because of these results, I ended up going with 0.0001 for both the learning rate and regularization functions. I chose an epoch of 15 because it tended to give higher values of accuracy compared to a lower epoch like 13 and had less overfitting compared to a higher epoch of 17. Playing around with the width and depth of the hidden layers led me to the conclusion that the width of each layer didn’t impact the validation accuracy as much as the depth did. When I tested with a larger depth (for example, three hidden layers each with a width of 300), the model would easily give a validation accuracy over 90%, but with a lot of overfitting. This is how I ended up choosing two hidden layers, one with a width of 300 and the other with a width of 200 for my model. While I ended up going with the hyperbolic tangent activation function, the RELU function did just as well. Other activation functions including the hard sigmoid made the accuracy worse. The parameters I ended up choosing still gave a model that was overfitting, but the loss functions were decaying well and the training accuracy was only 1-2% above the validation accuracy, which isn't very significant. The confusion matrix for this test displays the predicted data in each column, and the actual data along the rows. Just looking at this matrix, we can see that the larger values mostly lie along the diagonal, which means that most of the data points were classified correctly. However, the rest of the matrix is not sparse like it should be. One large value of 126 is in the 2nd row and 6th column. This means that these points were classified as a pullover (value of 2 in MNIST) when they were a shirt (value of 6 in MNIST). We can reason this number is high because a pullover is very similar to a shirt. Other cells with smaller values may not be able to be reasoned in this way and may be due to the parameters not being optimized enough (Sec. IV, Figure 1).

Part 2

The accuracy of this final model on my test data was 89.96% for one of the runs. For this CNN, I tested the kernel size, filter size, zero-padding and learning rate parameters. I tested all of these on 5 epochs, and ended up choosing 10 epochs for a better result.

With a decrease in kernel size (filter window) for all of the CNNs I defined, I found that there was quite a bit of overfitting. The validation accuracy didn't really change, and remained stable the entire time. I then halved all of the filter values (which corresponded to the output window) and found that overfitting also occurs in this case as the training accuracy started to go above the validation accuracy towards the end. The loss function didn't really decay as well even though it was low, but it seemed to be stable for each epoch. When I took out zero-padding and kept the kernel size at 4, I found that no overfitting occurred, but the loss function didn't decay as much as when there was zero padding, which wasn't a good sign. Lastly, I tried decreasing the average pooling values to 1, but this took a long time to run and didn't help my result. After decreasing the regularization function to take in a value of 0.00001 instead of 0.0001, I was getting better results.

Picking the RELU activation function for the layers and an epoch of 10 produced validation accuracies of 89-90%. While these still overfit, the loss function still decays well and the training accuracy is only 1-2% above the validation accuracy at the end, which isn't that significant of a difference. Similar to Part 1, the confusion matrix for the test data had most of its points along the diagonal, which is what we're aiming for. This confusion matrix also has outliers which can be explained by how similar some of the fashion data are to each other (Section IV., Figure 2).

For both Parts 1 and 2, I was having a hard time improving the accuracy of my test data without seriously overfitting. I could have improved my models by further exploring the learning rate and regularization functions, as I was arbitrarily choosing them rather than looking at them mathematically.

Appendix A. Python functions used and brief implementation explanation

$tf.keras.layers.Conv2D()$ – Creates a two-dimensional convolutional layer. Takes in the kernel and filter size parameters, number of strides, activation function, zero padding, and other parameters.
$tf.keras.models.Sequential()$ – Creates an object of all of the layers you have defined.
$tf.keras.models.Flatten()$ – Takes in an input and flattens it.

$tf.keras.layers.AveragePooling2D()$ – Applies a filter by taking the average of a two-dimensional receptive field.
$tf.keras.optimizers.Adam()$ – Creates an optimizer. The “Adam” optimizer takes in a learning rate and other parameters.
$tf.keras.regulizers.l2()$ – Creates a regulizer that applies an L2 norm.
$tf.keras.layers.Dense()$ – Creates a dense layer. Takes in an activation function, regularization function, and other parameters.
$object.complie()$ – Applies the optimizer to the neural network model (represented by “object”).
$object.fit()$ – Fits the training and validation data to the neural network model (represented by “object”).
$object.evaluate()$ – Evaluates the test data on the neural network model (represented by “object”).

Appendix B. Python codes

```
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:]

from functools import partial
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_reg
ularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    #my_dense_layer(500),
    my_dense_layer(300),
    my_dense_layer(200),
    my_dense_layer(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
             metrics=["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs=15, validation_data=(X_valid,y_v
alid))
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='center', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
#convolutional neural network
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_reg
ularizer=tf.keras.regularizers.l2(0.00001))
```

```
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="v
alid")

model = tf.keras.models.Sequential([
    my_conv_layer(6,5,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(16,5),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(120,5),
    tf.keras.layers.Flatten(),
    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_v
alid))

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)

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='center', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```