AMATH 482: Neural Networks: My Personal Black box

Anh-Minh Nguyen

March 10, 2020

Abstract In this report, we will be tinkering with neural networks to help us classify a huge dataset concerning fashion items, or just clothes. What we are about to witness is, in my personal opinion, a defiance of human perseverance and reasoning for the acceptance of a far superior computational force, that is my laptop.

1 Introduction and Overview

Computers have given us the gift of being a trillion times faster then we could ever be in terms of calculation. We, the humans, have done the theoretical work, and now it's up to these small metal chips to do the brunt of the work. One thing we have been trying to speed up and refine is the decision-making process of a computer. In comes my new dear friend and enemy, neural networks. We will be using the fashion_mnist data set, which we will training our neural network on. Our goal is to achieve at least a 90% accuracy.

2 Theoretical Background

I want to start by speedrunning through two statistical staples, Linear Regression and Logistic Regression (the algorithm is regression while the decision process is classification).
Linear Regression is fitting a line of best fit, where the data looks like $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ...(\mathbf{x}_N, y_N)$, where N is the number of observations. We see that $\mathbf{x}$ can be a vector of inputs while each y is our output based on those inputs. We assume that the relationship between our predictors and response is linear. We often don't know the true value of the predictors, so we have to predict our parameters. To measure how well we have done, we use the mean squared error (MSE). Our prediction is:

$$\hat{y}_j = m\mathbf{x}_j + \mathbf{b} \tag{1}$$

and our MSE is:

$$\frac{1}{n}\sum_{j=1}^{N}(y_j - \hat{y}_j)^2 \tag{2}$$

Let's look at a multivariate linear regression.

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_1 + ... + a_{1n}x_n + b_1 \\ y_2 &= a_{12}x_1 + a_{22}x_2 + ... + a_{2n}x_n + b_2 \\ &\vdots \\ y_m &= a_{m1}x_1 + a_{m2}x_1 + ... + a_{mn}x_n + b_m \end{aligned}$$

Which we can relabel as

$$\mathbf{y} = A\mathbf{x} + \mathbf{b} \tag{3}$$

A is the matrix of our parameters weight.

Now, let's consider Logistic Regression. The data is exactly the same as our Linear Regression. However, we also have the logistic function, which looks like a curved Z being pulled apart sideways.

$$\sigma = \frac{1}{1 + e^{-x}} \tag{4}$$

To plug in our parameters, we can input a linear function into where the x is. Because the values of our function, will fall between 0 and 1, we can write them as probabilities:

$$p = \frac{1}{1 + e^{-mx+b}} \tag{5}$$

One big difference between this and Linear Regression is our loss function. For the sake of brevity, we will briefly explain it:

$$-\frac{1}{N}\sum_{j=1}^{N}[y_j \ln(p_j) + (1 - y_j)\ln(1 - p_j)] \tag{6}$$

This is called the cross-entropy loss. The reason we take the log is so we can make it convex, so we can find the local minimum. It helps work around the fact that we don't have a close form solution for the most basic loss function of logistic regression. We want our result to be a vector of probabilities that add up to 1, so we have the softmax function:

$$\mathbf{p} = \sigma(\mathbf{y}) = \frac{1}{\sum_{j=1}^{m} e^{y_j}}(e^{y_1}\ e^{y_2}\ ...\ e^{y_m})^T \tag{7}$$

Now, that we have covered these topics. Let's start with a perceptron, a network made up of linear threshold units. The perceptron has input and output layers, will possible hidden layers in between. Each layer, from left to right, has a bias term and weights that connect it to each component of the next layer. We see that 1 will represents an active neuron while 0 represents an inactive one. For the output neuron to activate, the weighted sum of the inputs should be greater than some threshold determined by our activation function.
The reason we mentioned Linear and Logistic Regression is that we see that the functions we use to represent these calculation resemble the calculations we perform within those statistical methods. For example, we can write the equation of the multilayer perceptron below as:

$$\mathbf{x}_2 = \sigma(\mathbf{A}_1\mathbf{x}_1 + \mathbf{b}_1) \quad \mathbf{y} = \sigma(\mathbf{A}_2\mathbf{x}_2 + \mathbf{b}_2) \tag{8}$$

We have that $\mathbf{b}_1$ are the biases between input and hidden layers, $\mathbf{b}_2$ are the biases between the hidden and output layers, $\mathbf{A}_1$ and $\mathbf{A}_2$ are the respective weight matrices between the layers, and $\mathbf{y}$ is our output layer.

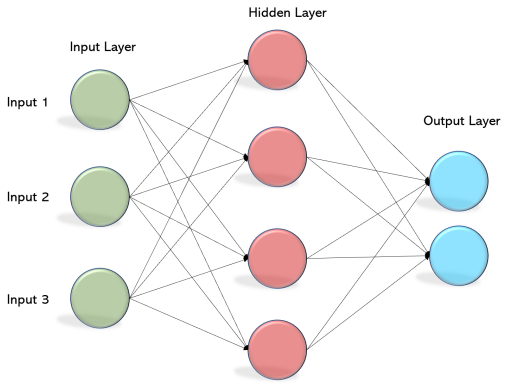

Figure 1: Example of Perceptron

(a) Convolutional Networks

Like the original perceptrons, convolutional neural networks also take inspiration from biology, specifically the eyes this time. But to greatly simplify it, specific neurons are responsible for specific regions, *receptive fields*, of our eyes. We can say that there are layers of neurons that take in the information, and process the images, from simple to complex.

This is replicated in machine learning, except we do it with numbers. Consider a matrix of numbers, or an image essentially, with the value of each pixel scaled between 0 and 1. We can look at different receptive fields of the matrix, usually *nxn* with a *filter kernel* to create a *feature map*. To center our filter at every value if desired, we can use zero padding, which creates a boundary around of 0 around the image. Sometimes, we can increase the *stride*, that lets us adjust the shift of our filter and size of our feature map. Next, we can do pooling, which is like our filter kernel, except we're fundamentally sub-sampling. Some of most popular options include *max pooling*, grabbing the max value of the pool, or *average pooling*, calculating the average of the pool.

3 Algorithm Implementation and Development

For our assignment, we will be building a fully-connected, feed-forward neural network, where every neuron is connected to every neuron in the previous layer and we move in one direction. We will be experimenting with the options of the neural network like the width of the layers, the step function, and the L2 Regularization parameters. It's going to be an adventure, tumultuous and emotional. Here, is where we go from the civilized, wealthy towns of the East Coast to the wild, wild West.

But, here's the situation. After we spend hours trying to build a neural network using fully connected layers, we'll experiment with convolutional layers and mess with some more options like stride, padding, and normalization layers. How many hours will we spend trying to hit our accuracy goal? Find out in the next section.

4 Computational Results

In this section, we go over the fruit of my laptop's labor, over a couple hours of nearly overheating. We always input our data as 28x28x1.

(a) Fully Connected Only

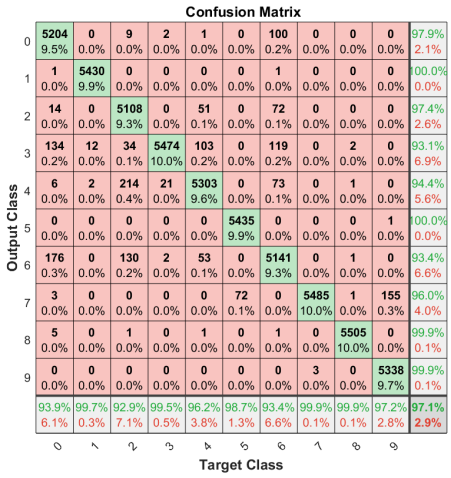

Figure 2: Fully Connected Only - Train

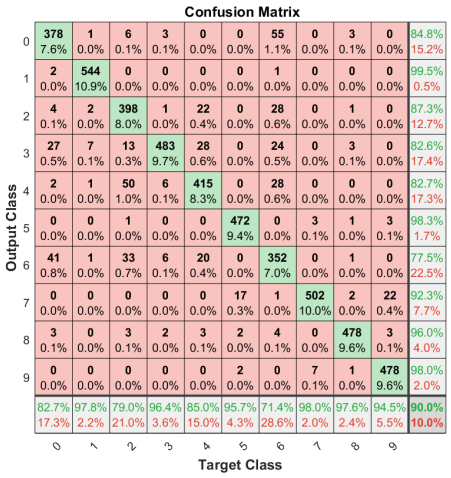

Figure 3: Fully Connected Only - Valid

In our final model of using only fully connected layers, we ended up using 3 fully connected layers activated by relu layers, with widths of 700, 500 and 10, then a softmax layer and classification layer. I discovered the *sgdm* optimizer. I did not use a bias and had an initial learning rate of 0.01, with 50 epochs. So, for the training, validation, and testing set accuracies, we respectively got: 97.1%, 90.0%, and 88.8%.

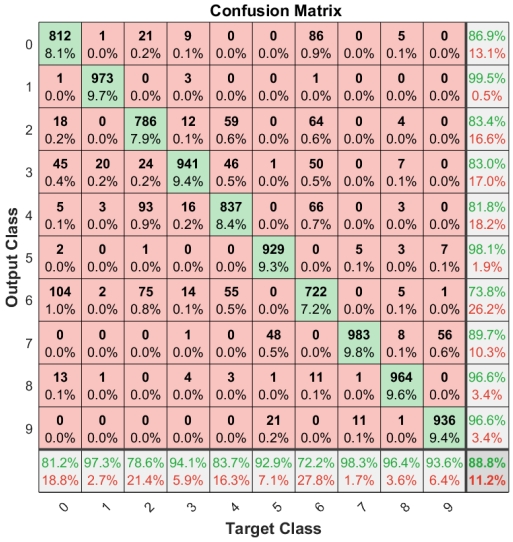

Figure 4: Fully Connected Only - Test

I think this is pretty good considering our final model didn't touch a single observation in the testing to train the model.

(b) Convolutional Neural Network

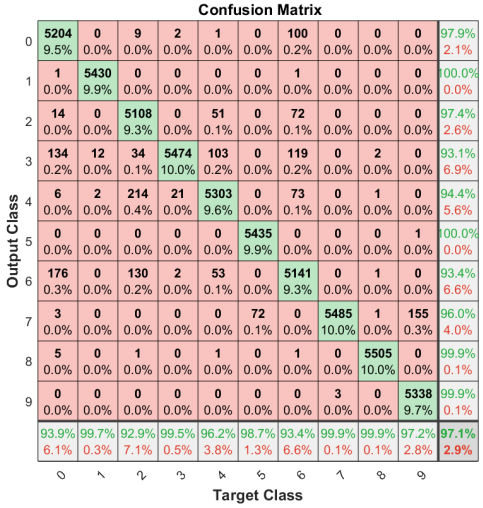

Figure 5: Convolutional - Train

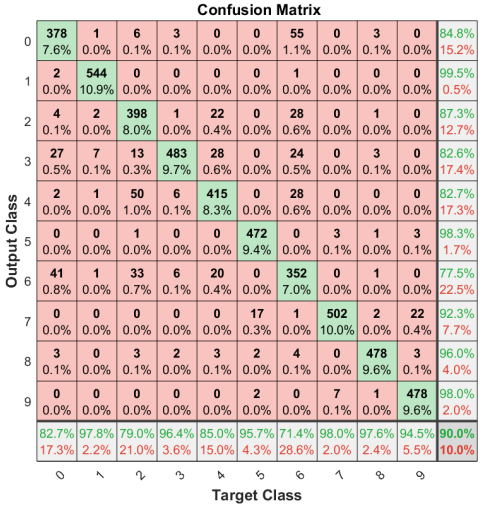

Figure 6: Convolutional - Valid

For our convolutional networks, we actually used starter code provided by MATLAB themselves. It features 3 sets of a convolution layer with padding, a batch normalization layer, and a relu activation layer. There is a max pooling layer with stride 2 after the first and second set. The convolution layers are size [3 3], and the number of filters go as 16, 32, and 64 as we proceed deeper. Finally, we have a fully connected layer with 10 neurons, a softmax layer, and finally, the classification layer.
We run it through 10 epochs, and an initial learning rate and L2 regularization of 0.0001. Also, we switched back to our default optimizer, adam. Our final model accuracies for train, valid, and test were: 92.8%, 90.5%, and 89.3%.

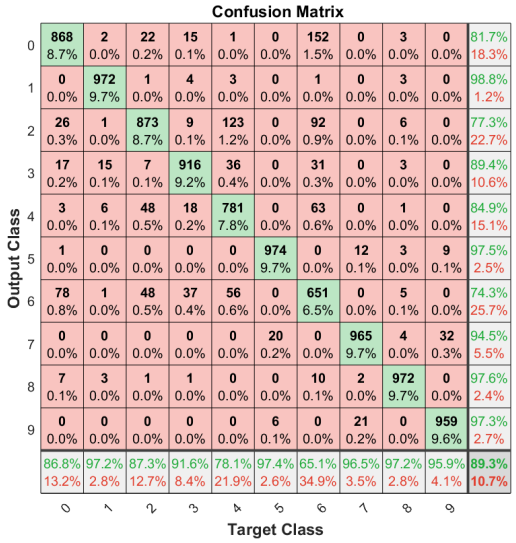

Figure 7: Convolutional - Test

We see that our convolutional neural network performed slightly, just slightly, better than the non-convolutional network.

5 Summary and Conclusions

My first impression, as one could probably tell, was that neural networks were the literal definition of black box. After hours of googling and listening to hints from the professor, it became clear that we were supposed to be "inspired" by other brave souls who had ventured into the chaos of neural networks. It took hours to find a network that breached 90% just enough that I ended with the bare minimum. Computationally, convolutional networks took my laptop and tested some of its limits, but, despite the time to run, the models produced better results consistently.
At first, I thought that there was no structure, no formula to the madness. Then, I learned that setting the initial learning rate lower could lead to better results, so I combined that with a higher max number of epochs. I lowered or removed the bias, and I'm not entirely sure why, but it seemed to perform better. It may be because it allows us to adjust our weights more freely. Like many others, I adopted a pyramid structure, a sort of compression of information.
Overall, the mechanics of neural networks confused me, but they have extreme flexibility and utility. And if anything, to find the next big thing in machine learning, we may need to look within ourselves.

6 Appendix

A MATLAB Functions

- *classify*: Compare fitted labels with the observed values
- *trainingOptions*: Specify options for our neural network
- *trainNetwork*: Find optimal weights and biases for our hyper parameters of the neural networks

B MATLAB Code

```
clear; clc; close all;

load('fashion_mnist.mat')
%%
X_train = im2double(X_train);
X_test = im2double(X_test);

X_train = reshape(X_train,[60000 28 28 1]);
X_train = permute(X_train,[2 3 4 1]);

X_test = reshape(X_test,[10000 28 28 1]);
X_test = permute(X_test,[2 3 4 1]);

X_valid = X_train(:,:,:,1:5000);
X_train = X_train(:,:,:,5001:end);

y_valid = categorical(y_train(1:5000))';
y_train = categorical(y_train(5001:end))';
y_test = categorical(y_test)';
%%
layers = [imageInputLayer([28 28 1])
        fullyConnectedLayer(784)
        reluLayer
        fullyConnectedLayer(200)
        reluLayer
        fullyConnectedLayer(100)
        reluLayer
        fullyConnectedLayer(10)
        softmaxLayer
        classificationLayer];
options = trainingOptions('sgdm', ...
    'MaxEpochs',15,...
    'InitialLearnRate',1.1e-2, ...
    'L2Regularization',1.7e-5, ...
    'ValidationData',{X_valid,y_valid}, ...
    'Verbose',true)
net = trainNetwork(X_train,y_train,layers,options);

%% Confusion for training
figure(2)
y_pred = classify(net,X_train);
plotconfusion(y_train,y_pred)
```

```
figure(3)
y_pred2 = classify(net,X_valid);
plotconfusion(y_valid,y_pred2)
%% Test classifier
figure(3)
y_pred3 = classify(net,X_test);
plotconfusion(y_test,y_pred3)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PART 2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear; clc; close all;

load('fashion_mnist.mat')
%%
X_train = im2double(X_train);
X_test = im2double(X_test);

X_train = reshape(X_train,[60000 28 28 1]);
X_train = permute(X_train,[2 3 4 1]);

X_test = reshape(X_test,[10000 28 28 1]);
X_test = permute(X_test,[2 3 4 1]);

X_valid = X_train(:,:,:,1:5000);
X_train = X_train(:,:,:,5001:end);

y_valid = categorical(y_train(1:5000))';
y_train = categorical(y_train(5001:end))';
y_test = categorical(y_test)';
%%
layers = [
imageInputLayer([28 28 1])

    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,64,'Padding',1)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
```

```
    classificationLayer];
options = trainingOptions('adam', ...
    'MaxEpochs',10,...
    'InitialLearnRate',1e-4, ...
    'L2Regularization',1e-4, ...
    'ValidationData',{X_valid,y_valid}, ...
    'Verbose',false, ...
    'Plots','training-progress');

net = trainNetwork(X_train,y_train,layers,options);
%% Confusion for training
figure(1)
y_pred = classify(net,X_train);
plotconfusion(y_train,y_pred)

figure(3)
y_pred2 = classify(net,X_valid);
plotconfusion(y_valid,y_pred2)
%% Test classifier
figure(3)
y_pred3 = classify(net,X_test);
plotconfusion(y_test,y_pred3)
```

C References

https://www.mathworks.com/solutions/deep-learning/examples/training-a-model-from-scratch.html
Lecture Notes AMATH 482 Winter 2020
https://stats.stackexchange.com/questions/376312/mnist-digit-recognition-what-is-the-best-we-can-get-with-a-fully-connected-nn-o