Assignment 2 Report

Gregory Smith

23 May 2018

Introduction

I just want to say that implementing backpropagation was a nightmare. While figuring out the algorithm and how to code it is hard enough, why not add numpy array shape errors and overflows on top of that? However, I managed to implement backpropagation before *too* long, and created a little neural network from scratch.

While this has been by far one of the most difficult assignments I've ever done, I have to say that it is one of the most **rewarding** assignments I've ever done.

I'll break down this assignment into the subsequent parts: data preparation, creating the neural network, & testing the neural network.

Data Preparation

As we only had a week for this assignment, I didn't give as much attention to data preparation that I should have. In fact, I actually did this in quite a lazy way. For reference, the data started off looking something like this:

	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	Utilities	Lot Config	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition
0	90	RL	NaN	7032	Pave	NaN	IR1	Lvl	AllPub	Corner	...	0	NaN	NaN	NaN	0	12	2006	WD	Normal
1	20	RL	63.0	9457	Pave	NaN	Reg	Lvl	AllPub	Inside	...	0	NaN	GdWo	NaN	0	9	2007	WD	Normal
2	20	RL	NaN	9216	Pave	NaN	IR1	Lvl	AllPub	Inside	...	0	NaN	MnPrv	NaN	0	9	2008	WD	Abnorml
3	50	RM	52.0	6240	Pave	NaN	Reg	Lvl	AllPub	Inside	...	0	NaN	MnPrv	NaN	0	7	2007	WD	Normal
4	80	RL	70.0	10500	Pave	NaN	Reg	Lvl	AllPub	FR2	...	0	NaN	GdWo	NaN	0	12	2007	WD	Normal

Figure 1: Uncleaned Data

Obviously, this needs to be cleaned up to be usable. Let's go through my process for cleaning.

1. NAN Values

First I search through every column in the dataframe one by one and checked to see if it had any NAN values in it. If it returned true, I cut it out of the dataframe. I feel bad about dropping features like this, but some features had a ***LOT*** of NAN values, so I chose to drop features instead of observations.

2. Categorical Variables

The method I used to find categorical variables was pretty simple, but I know for a fact it did not catch all categorical variables. Let me explain. My method searched for anything not encoded with a data-type as a number. So basically, anything that was not a float, integer, etc. This resulted with all of the categorical variables that were *strings*. While some features may have actually been categorical in the guise of continous, I did not search for or change them. Again, due to the fact that we only had a week.

I took these features with strings and used panda's *get_dummies* function for each of these features. This provides a quick and simple way for one-hot encoding. Doing this quickly shot up my number of features.

3. Normalization

I used the *min-max* scaling method to normalize my data. As far as my own research goes, this is not a great method to use while compared to others. Min-max scaling is not robust and is highly sensitive to outliers. However, since we only had a week for this assignment, I chose this method as it is easy to implement and I can normalize and de-normalize with ease.

4. Train-Test-Validate

I chose a train-test split of 80%-20% respectively, and a train-validate split of 80%-20% respectively after the train-test split. This left me with 1491 observations in my training data, 373 observations in my validation data, and 466 observations in my testing data.

5. Clean Data!

This left me with a nice, normalized dataframe that looks like this:

	0	1	2	3	4	5	6	7	8	9	...	206	207	208	209	210	211	212	213	214	215
0	0.176471	0.067670	0.333333	0.375	0.307692	0.950000	0.217302	0.221308	0.0	0.264506	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0
1	0.176471	0.029230	0.444444	0.500	0.384615	0.000000	0.136301	0.294431	0.0	0.226451	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0
2	1.000000	0.018298	0.333333	0.500	0.307692	0.966667	0.092244	0.346731	0.0	0.210625	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	0.176471	0.036951	0.333333	0.375	0.307692	0.000000	0.083525	0.208232	0.0	0.149586	...	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
4	0.000000	0.058754	0.666667	0.625	0.976923	0.950000	0.282240	0.000000	0.0	0.231726	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0

Figure 2: Cleaned & Normalized Data

After performing all of these steps, I have a normalized dataframe with no missing values and 216 features. Most of these features are one-hot encoded categorical variables, though.

Creating the Neural Network

This was the cause for most of my frustration with this assignment. Not only was the algorithm difficult, numpy arrays were giving me a lot of problems. I guess the algorithm didn't like the shape of them. I finally got backpropagation working, and everything came together after that. Let's run through my design for the neural network.

1. Activation Function

I chose to use Relu (*max(x, 0)*) for all of my activation functions. I chose this because the derivative is simple, and because it is one of the most used activation functions.

2. Cost Function

Our cost function is MSE, or:

$$\frac{\sum(\hat{y} - y)^2}{n} \tag{1}$$

While the derivative of our cost function is:

$$\frac{2 * (\hat{y} - y)}{n} \tag{2}$$

3. Mini-batches!

I designed the neural network to work with mini-batches. I set a default size of 32, and is set by the user. My algorithm returns the average cost of these mini-batches per epoch. Along with this, my data also shuffles every epoch.

4. Hyperparameters

Sorry to dissapoint, but I didn't do anything fancy with my network. Given more time, I'd like to tune the network and add some optimizations. I just have the standard hyperparameters like: size of the hidden layers, learning rate, & mini-batch size.

5. Weights & Biases

The weights are initialized when a neural network object is created. It assigns them random values and also scales them by .1 so that they'll be closer to 0. There is also an added bias that the network learns over time.

Testing the Network

I must say, this network is a sensitive little guy. You change one hyperparameter ever-so slightly...and...BAM! Everything gets screwed up. Now, I don't know how to tune a neural network, so my testing has consisted of playing with random values, small and large.

1. Learning Rate

For some reason, I have found that large learning rates actually help get lower costs. This might be due to how I cleaned the data and how it is normalized. There seems to be a lot of noise and very small local minimums, and with a smaller learning rate, you can see the cost bouncing around between these small minimums.

2. Epochs

Just for kicks, I've been training for around 50,000 epochs. One thing to notice, is that the cost for training and validating keep going down ever so slightly, and they get closer. For example, I initialized the network with 10 nodes in the first hidden layer, 10 nodes in the 2nd hidden layer, learning rate of .9, & a mini-batch size of 32. This is my plot of Cost vs Epochs:

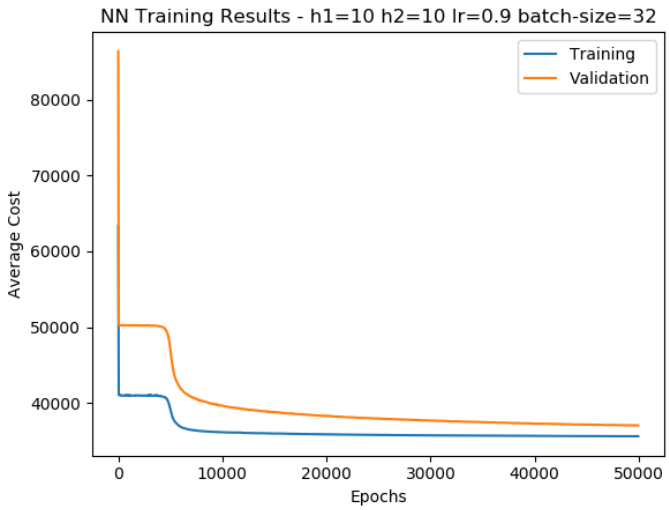

Figure 3: Neural Network Escapes a Local Minimum! (I think)

I update a list 1000 times over the course of training and use this data to plot my cost function traveling down. If I updated this list less often and had a smaller learning rate, this descent would look like a nice smooth curve. With it being constantly updated, however, we can see where it starts finding minimums. Very early on we can notice that it hits a local minimum for a *while*. Then around 5000 epochs, it breaks through even lower and continues on a gradual descent.

3. Predictive Power

So I know what you're thinking, "Cool, Greg. You made a network. Congratulations. So, how well does it predict?" I hate to admit it, but I do not get the best predictions on new data. I plotted my predictions and expected values vs the observations on *my* testing data (not the hidden testing set). Take this image for example:

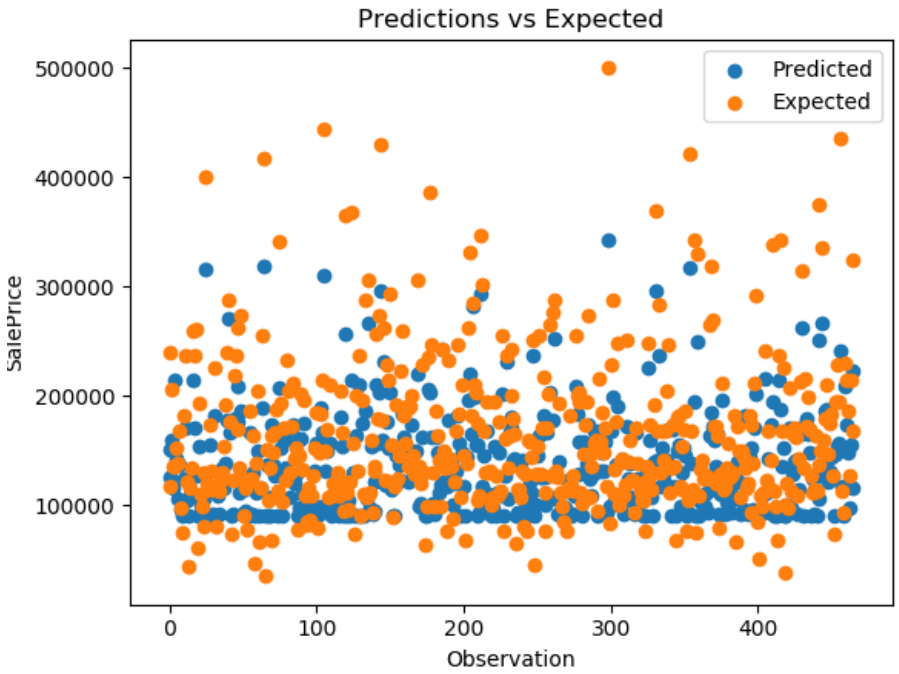

Figure 4: Be patient he's learning!

This is using the same parameters as the figure above, but at 5000 epochs. With such a large learning rate around this time, you can see most points along the general trend of the data. What's interesting to note is that while some points are moving up to the large SalePrice observations, there seems to be a hard-line that the data does not want to move from for smaller SalePrice observations.

Let's see how the network predicts at 50,000 epochs.

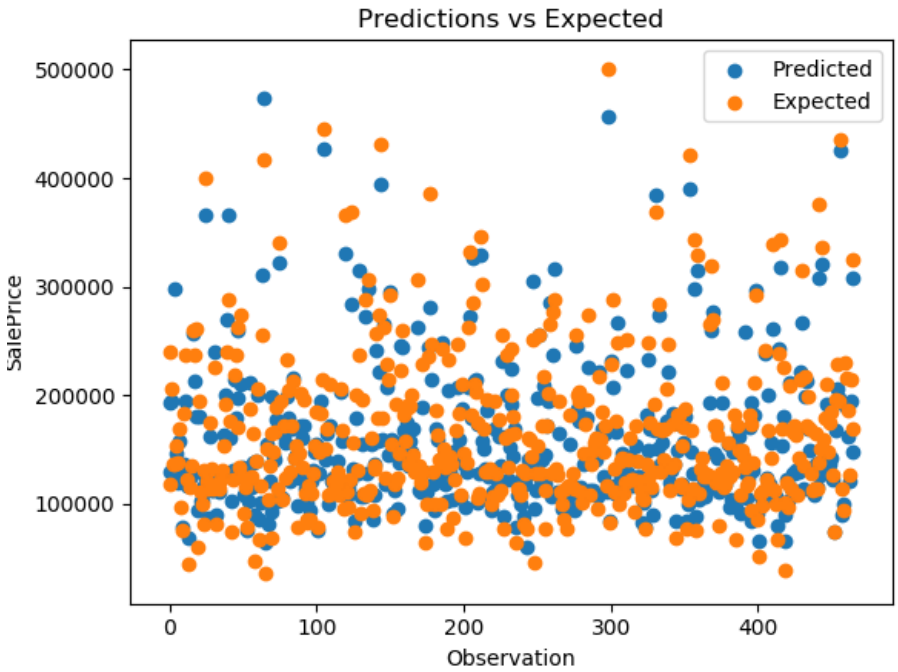

Figure 5: Close enough?

After a measly 50,000 cycles through shuffled training data, the network is getting better at predicting. We can see the predicted values moving closer towards the large vertical outliers, and we can notice that the network is getting better at predicting the smaller vertical outliers too.

Now these results aren't that great, but bear with me. I just learned how to create a neural network, implement backpropagation, and clean data. This network isn't finely tuned, and does not have any fancy-dancy optimizations. But, hey! It's not ***THAT*** bad at predicting. It's smart enough to learn the general trend of the data and then some! As my first neural network EVER, I'm quite happy with these results.

Conclusion & Recap

I believe my network would perform better I spent more time cleaning my data, and if I used a different normalization technique. From what I've read, it appears that the min-max normalization method is not robust, and is highly sensitive to outliers. If we look at Figure 5, we can see that there are a decent amount of potential outliers. This can skew the results as it is normalizing based on the minimum and maximum values (which could be very different from the mean).

When it comes to cleaning my data, I dropped any feature that had a missing value for convenience. Given more time, I could have imputed this missing data, or dealt with them in a smarter way. Those dropped features could have been very significant for the predicting SalePrice.

As I said before, I only had a week to do this. I just started a new job and have had limited time, so I haven't gotten around to tuning my network as much as I'd like. However, I'm very happy that I was able to create a neural network from scratch that works. Thanks for reading!