Comparing Neural Networks to Linear Regression Models

Max Barshay **Sahil Bobba** **Reesa John**

Abstract

Neural networks continue to garner more and more attention due to their incredible predictive prowess. In this report we compare the effectiveness of linear regression versus that of our own implementation of a neural network to determine which model is better at predicting housing prices on the Sberbank Russian Housing Market dataset.

Introduction

The dataset used in this report comes from the Sberbank Russian Housing Market dataset, from a 2017 Kaggle competition, whose goal was to predict real estate prices in Russia.

There are over 30,000 listings in the dataset, which contains a collection of over 500 features coming from two datasets, *train.csv* and *macro.csv*. The *train.csv* includes information about the property involved in the transaction such as *number of rooms, square footage, year built* as well as information about the local area, and *macro.csv* contains demographic information such as *average income, average education level*, and much more. Our goal in this report is to predict the price the house was sold for.

Dataset Creation

Creating the final dataset used for both our linear regression model and neural network model presented many challenges.

The first major challenge to overcome was handling rows with missing values without having to drop valuable pieces of information from our dataset. After combining both the *train.csv* dataset and the *macro.csv* dataset, there were over 100,000 missing values spanning over thousands over rows and hundreds of columns that we needed to handle. Another obstacle in cleaning the dataset was determining how to retain aspects of categorical variables such as *sub_area* and *product_type* without dummifying these variables. Finally, we had to determine which 50 features would be the best predictors from hundreds.

To handle the 100,000 missing values, we couldn't simply fill in these missing values with summary statistics such as column means or medians as some observations with certain categorical attributes would deviate far more from the mean than others. Instead, we decided to group rows by three categorical variables, *sub_area, product_type, and ecology*, that had clear groupings for the missing values we were trying to estimate.

We then created a function *remove_na* that took the mean for a column grouped by these variables. With 2 unique product types, 146 unique sub areas, and 5 unique ecologies giving us a total of 1460 different groups, so for every combination of *product type, sub area and ecology* there was a column average to go with it. The function iterated over the rows in the column and would fill all NA values with the mean for its respective combination of *product type, sub area* and *ecology*. This not only fixed our problem of missing values but also helped

represent the categorical variables in the final dataset without needing them in the final dataset.

While handling the missing values in this manner was effective, there were five sub areas that had too little data on them and couldn't efficiently be used to fill in missing values. As such, we decided to drop these rows that were associated with these few sub areas and our dataset went from 30,471 to 30,292 rows.

After running our *remove_na* function, most of the missing values were taken care of. However, there were still some columns that had between forty to seventy missing values. Since there was such a small number of missing values remaining, we were able to fill them with the column means without changing the overall distribution of these predictor variables.

Finally, we determined the top fifty predictors to be the fifty predictors with the highest correlation with our response variable *price_doc*. It turned out that all fifty predictors with the highest correlation absolute value were all contained in the original train.csv file, thus these columns were the final predictors we retained for the final dataset we used for testing. Additionally in order for our predictors to be comparable to one another, and to *price_doc*, we standardized the predictors by centering them around the mean and dividing it by their respective standard deviation, and divided *price_doc* by hundred thousand so that it's range was roughly 1000.

Linear Regression

In this project, our goal was to compare the effectiveness of linear regression compared to neural networks. If this data set truly is non-linearly separable, then we would expect the neural network to outperform linear regression.

Although we likely could have increased our linear regression model's performance by removing variables that were not significant, we decided not to since the goal of this project was not to create the best linear regression model, but rather to use it as a baseline for our neural network implementation.

As was mentioned in the previous section, we used the top 50 predictors that had the highest correlation with the response variable, price. Although more feature engineering could have been done to improve linear regression, we just decide to standardize all of our predictors and divide the response by 100,000 in order to remain consistent with what we did in our neural network.

Using 5-fold cross validation we got a MSE of 2817.75 and a corresponding RMSE of 48.18. This means that on average our predictions were off by around 4,818,000 rubles. The standard deviation of y was around 7,782,000 rubles. Meaning that our predictions are roughly a standard deviation away from the ground truth on average.

Once again using 5-fold cross validation we found that the average R^2 was .3387. This means that roughly 33.87% of the variation in price is explained by our model. This is a low R^2 value and seems to agree with the fact that

our model is off by roughly a standard deviation on average.

Neural Network Implementation

The *NeuralNetwork* class is our implementation of a neural network and contains two functions of interest, *fit* and *predict*. When initializing the class, there are 7 hyperparameters of interest: *layers, nodes, activation, loss, learning_rate, batch_size,* and *anneal*.

The parameters *layer* and *nodes* control how many hidden layers and the amount of nodes in each layer respectively. The parameters *activation* and *loss* take functions to calculate the activation and loss functions respectively. Additionally both methods must contain a boolean parameter called *derivative*, that when set to *True*, returns the derivative. The parameter *anneal* is what is multiplied to the learning rate in the fit method in order to decrease it whenever the validation accuracy is not improving. The remaining parameters *learning_rate* and *batch_size* represent the learning rate and the batch size for the stochastic gradient descent.

The *fit* method of the *NeuralNetwork* class takes six parameters of interest: *X, y, X_val, y_val, mse_diff, max_epochs*. When we call *fit* upon the model, the initial weights are created using He initialization. It then uses stochastic gradient descent, and splits the *X* and *y* into random batches of *batch_size* per epoch in order to train the neural network by propagating a batch forward through the network and back propagating the error to update the weights and biases at each layer.

At the end of an epoch, the model is tested on *X_val* and *y_val* to evaluate how accurate the model is in terms of mean squared error, in order to ensure we aren't overfitting. It will continue training the network until it either reaches the max number of epochs given in *max_epochs*, or until it hits a stopping criteria which uses *mse_diff*. The parameter *mse_diff* is the minimum difference threshold between mean squared errors in the validation set to determine whether the model is still learning. Should the difference be under this value between epochs, we anneal the training rate by *anneal*, and should this happen five epochs in a row, we stop training entirely.

This method in addition to training the model also returns the average loss as well as the mean squared error on the validation dataset across epochs.

The *predict* method takes one parameter, *X* and is essentially equivalent to our class's internal method to forward propagate *X* through the neural network, but does not save any values and returns what the network predicts for each of the observations in *X* in the form of a 1-D array.

In addition to the neural network object itself, we have implemented two functions to pass onto *NeuralNetwork* when initializing it: *L2_Loss* and *ReLU* for *loss* and *activation* respectively.

Neural Network Evaluation Process

Before we evaluated any models, we first split the dataset into three sets of data, the train set, the validation set, and the test set. The

train set contains three-fifths of the dataset, and the validation and test sets each contain one-fifth of the dataset respectively.

The hyperparameters we focused on tuning were *layers*, *nodes*, *learning rate*, *batch size*, and *anneal*.

Our initial model consisted of a single hidden layer with four nodes, a learning rate of 1.00E-6, a batch size of 128, and 0.5 as the value we anneal the learning rate by when the model doesn't improve on the validation set. In the graphs, both the loss (Fig. 1) and validation (Fig. 2) accuracy in terms of mean squared error converged to 824.06 and 2777.12 respectively and its performance on the test dataset resulted in a mean squared error of 1466.30, which isn't too bad.

In order to see if we could improve this model, we created our own version of grid search.

For *layers*, we tested the values 1,2,4,6. For *nodes*, we tested 4,6,8. The values for *learning_rate* were 10E-5, 10E-6, 10E-7. And finally, the values for *batch_size* were 128 and 256 and the values for *anneal* were 0.5 and 0.2.

Since each model took anywhere from a few seconds to a few minutes to train depending on how many epochs it took to terminate training, it took roughly an hour for grid search to finish.

Results

The best model had two hidden layers with six nodes, a learning rate of 1.00E-5, a batch size of 256, and 0.5 as the value we anneal the learning rate by when the model doesn't improve on the validation set.

The mean squared error for this model is 912.17. Looking at the graphs which plot the average loss and validation mean squared errors across epochs, the loss decreased for the most part (Fig. 3), fairly steadily, and converged at 519.65

And while the validation dataset's mean squared errors (Fig. 4) were jumping around in the beginning, it also eventually began to consistently decrease in a stable manner, converging at 3387.03. However, for validation, it did not stabilize at the lowest mean squared error, implying there were some issues with overfitting in this model.

Looking at the summary statistics of predictions, you can see it had difficulty in predicting outliers but did a fairly decent job at predicting values that were not outliers. While the minimum and maximum predicted values for the predictions were 26.02, and 490.59, the actual minimum and maximum values were 1.9 and 700. That being said, these were both extreme values in the entirety of the target variable, price_doc divided by 100 thousand, where 90% of our over 30,000 observations fell between 20 and 145.35.

That being said, the ends of the interquartile were still off, the ranges being between 53.04 and 72.34 for the predictions

versus 47.86 to 81.52 in the actual test set, a difference of roughly 5 and 10 for the lower and higher end of the range, which translates to nearly half and million ruble in difference (roughly 6.5k to 13k in USD), so there is definitely room for improvement.

The mean squared errors for the next four best models as well as their hyperparameters were as follows:

920.83 with 6 hidden layers, with 8 nodes at each layer, a learning rate of 1.00E-6, a batch size of 128 and an annealing rate of 0.5,

959.97 with 4 hidden layers, with 8 nodes at each layer, a learning rate of 1.00E-6, a batch size of 256 and an annealing rate of 0.5,

963.68 with 4 hidden layers, with 6 nodes at each layer, a learning rate of 1.00E-6, a batch size of 256 and an annealing rate of 0.5, and

1001.59 with 4 hidden layers, with 8 nodes at each layer, a learning rate of 1.00E-6, a batch size of 128 and an annealing rate of 0.5.

Looking at the loss graphs of these models (Fig. 5), with the exception of the second best model, they all had some difficulty stabilizing in the beginning, but ultimately all converged to a fairly low loss around 500, much like our best model.

However, like our best model, the accuracy graphs (Fig. 6) for the mean squared error tell a different story. While all the mean squared errors on the validation dataset stabilized, it did not converge around the lowest point either, all converging between a mean squared error between 3000 and 3500, more signs of overfitting on the training data.

Comparison to the Linear Regression

As was mentioned, the MSE that we got for our linear regression model was 2817.75. The best MSE that we got using a neural network was 912.17, which is more than three times better.

Since we did not include interactions or higher order polynomials in our model, we could not model any non-linear functions with our linear regression model. Thus the threefold improvement in MSE upon using a neural network suggests that the data was not linearly separable.

It should also be noted that we tailored the data more towards our neural network than to our linear regression model. The neural network had the complexity and thus the ability to compensate for some of the irregularities in the data which the linear model could not. In our development we discovered ways to enhance the performance of our linear model, but we determined that the goal of this project was more to highlight the abilities of the neural network, thus we did not try to meet the assumptions of a linear regression.

In terms of the time it took to run, the linear regression model was superior, as it took a few seconds versus the few minutes it took to train the best model which was shy of 60 epochs which makes sense as linear regression the parameters can be calculated analytically rather

than iteratively learning until convergence like that of a the neural network. That being said, the increase in time was worthwhile given its huge performance increase.

Conclusion

Undeniably, the neural network outperformed the linear regression model in predicting housing prices. That being said, our neural network implementation does require additional work if it wants to be even more effective.

As mentioned, in the results, while the mean squared errors did converge on the validation dataset, it did not converge at a minimum, which is a sign of overfitting. Thus in order to improve our implementation of the neural network, we would need to implement a method like regularization in order to prevent the weights from fitting to the training dataset

Appendix

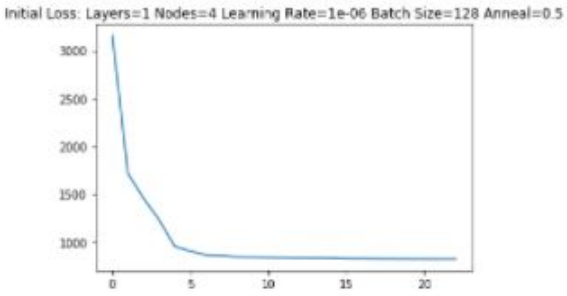

Figure 1. Loss of the initial model

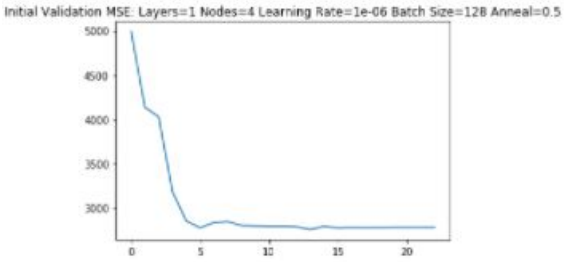

Figure 2. MSE of the initial model's validation set

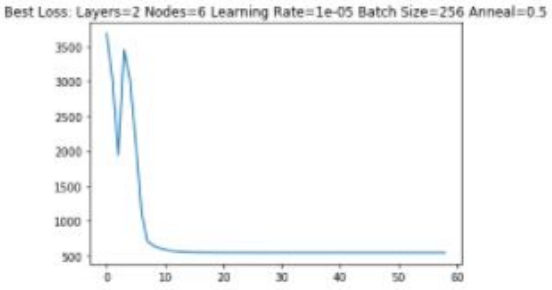

Figure 3. Loss of the best model

Figure 4. MSE of the bet model's validation set

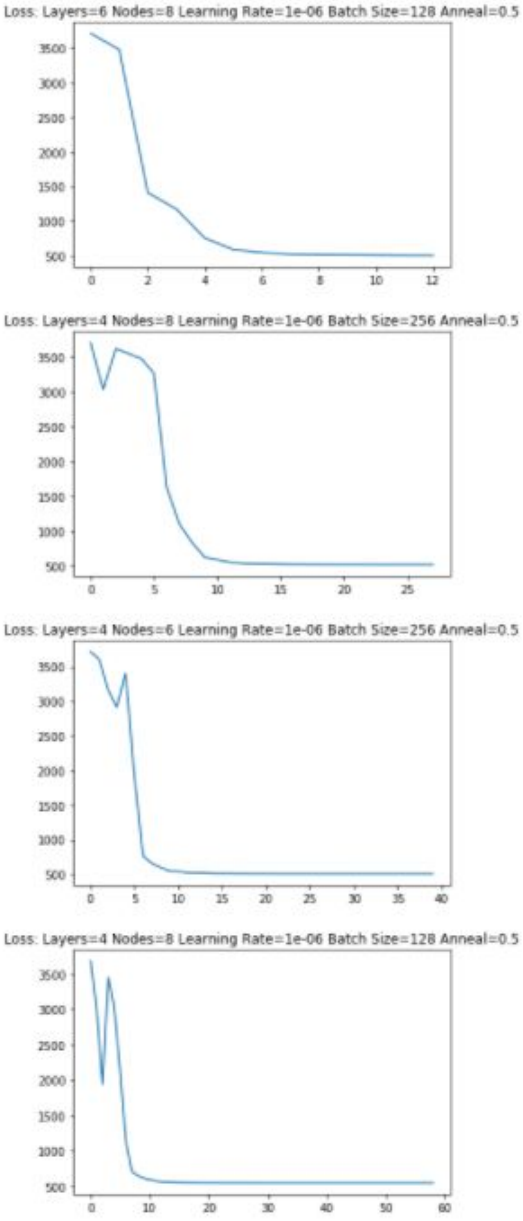

Figure 5. Loss of the next four best models

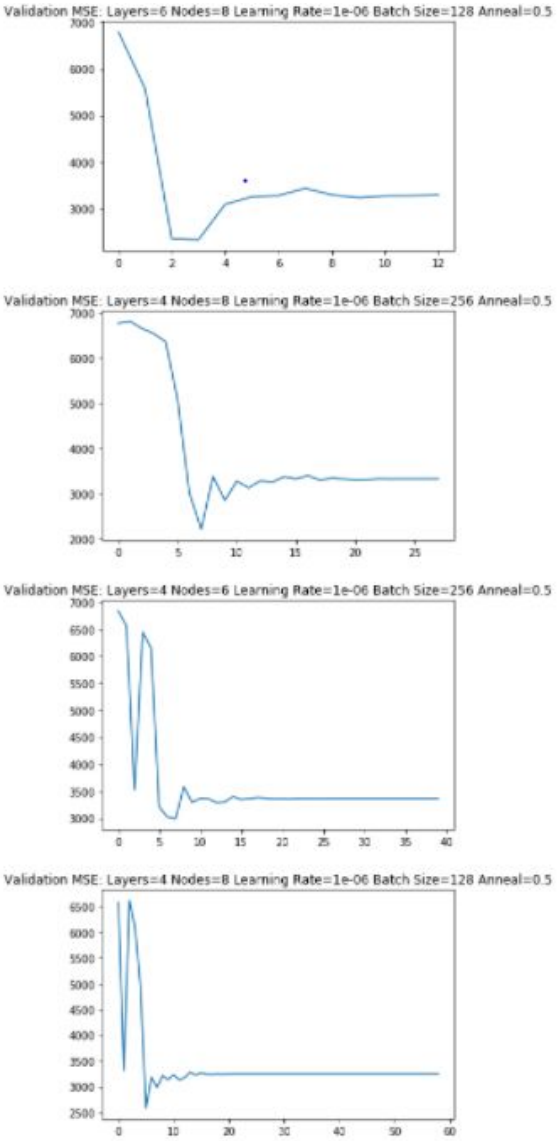

Figure 6. MSEs of the next four best models' validation set