Building a Predictive Neural Network and Comparing with Linear Regression

Lemar Popal (lepopal@calpoly.edu)
Josiah Lashley (jjlashle@calpoly.edu)
Mason Ogden (msogden@calpoly.edu)

Introduction

Neural networks, while complex, are extremely flexible and powerful for prediction. They are so flexible that according to the Universal Approximation Theorem, under certain assumptions a neural network with a single hidden layer that is arbitrarily wide can approximate any function to any chosen error threshold. This is an incredible result that truly shows their power and complexity. In this project, we will explore this power and complexity by implementing a fully-connected feed-foward neural network from scratch, and comparing its results to those of a linear regression model.

Dataset Creation

The data used in this project comes from a Kaggle competition hosted by a Russian bank (Sberbank). The goal was to accurately predict how much a property sold for based on a subset of the 391 total features. In total, there was data on 30,471 sold properties.

In order to accurately predict the sales price of a property, we first had to construct our own dataset from a few given datasets that would meet the requirements outlined in the spec. There were two datasets given: train and macro. Train lays out a collection of features about each property and the property's surrounding neighborhood, and some features that are constant across each sub area. Train also is the dataset that holds our target variable, "price_doc" which is the sales price of a house. Macro lays out macroeconomic indicators on the day that the house was sold to help show the state of the Russian economy at the time the house was sold.

The first obstacle that we ran into during the dataset creation process was how to handle missing (NA) values. The two datasets had 391 total features and many observations had missing values. We decided we should not remove the rows that had NAs because we would lose a lot of valuable information. To keep as much data as possible, we decided that we would replace all the NA values with the column's mean. Categorical features with many missing values were replaced by the mode (i.e. value that appears most often).

One requirement in the dataset creation process was that we were not allowed to dummify any variables for the geographic location, specifically the "sub_area" feature. Since we were unable to dumify this variable and since there were so many different distinct values for this column, we decided it would be better to drop this feature from the dataset entirely.

The next step in preparing our data for both linear regression and the neural network was standardizing the predictors and the response. For linear regression, this helped with interpreting the predictive accuracy, because the raw property prices were sometimes unbelievably large numbers, making non-standardized RMSE almost uninterpretable. For the neural network, standardizing the predictors is especially important, because we want the inputted predictors and the gradients being calculated in backpropagation to be centered at 0 and have constant variance. Additionally, the non-standardized response caused numerical overflow problems in some numpy math functions used to calculate the loss due to their huge size, some being billions of rubles.

In addition to the given train and macro datasets, we were also given some limits on how we may use them. One of the most important requirements was that the final dataset contain at most 50 numeric features and of those 50, only 10 of them can be from the macro dataset. To stay within these guidelines, we took all numeric features from both train and macro and ran our feature selection methods and selected the top 50 columns, with the constraint that no more than 10 are from the macro dataset.

We used several different feature selection methods. The first was Scikit-learn's "SelectKBest" function. SelectKBest is a function that, when accompanied with specified features, a target variable, and a selection method, will remove all but the k highest scoring features in the model. We used the F_regression scoring function, which calculates an F statistic for every predictor variable, that F statistic quantifying the amount of variation that the predictor explains in the response.

The second feature selection method was looking at the feature importance of each feature in an RandomForestRegressor model. Random Forest is an ensemble method that generates a large number of decision trees and the final result is an average over all the trees' predictions. Additionally, Random Forest selects a random subset of the features and a random subset of the data to train each tree. This helps prevent highly correlated trees, which means less reduction in variance. We can then estimate feature importance of a feature x_p by summing up the reduction in the Gini index from splits over x_p over all trees, because a greater reduction means greater importance in prediction. The last feature selection method simply looked at the correlation between all features and the response. We selected the features that had the highest correlation with sale price.

After performing the three feature selection methods discussed above, we found that our candidate feature sets contained data mostly from the train dataset, leading us to believe that macro data was not an important predictor. To be sure this wasn't the case, we forced the addition of macro data by ensuring each set contained at least 10 features from the macro dataset. To decide which of the potential feature sets was best, we performed a 5-fold cross validation of a linear regression model using each set and selected the one with the lowest RMSE on the test data. The final feature set, with an RMSE of 0.7436 standard deviations, is below:

'full_sq', 'sport_objects_raion', 'metro_min_avto',
'metro_km_avto', 'metro_min_walk', 'metro_km_walk', 'park_km', 'ttk_km',
'sadovoe_km', 'bulvar_ring_km', 'kremlin_km', 'zd_vokzaly_avto_km',

'nuclear_reactor_km', 'radiation_km', 'thermal_power_plant_km', 'fitness_km', 'swim_pool_km', 'stadium_km', 'basketball_km', 'detention_facility_km', 'university_km', 'workplaces_km', 'office_km', 'big_church_km', 'theater_km', 'exhibition_km', 'catering_km', 'office_sqm_500', 'sport_count_1000', 'office_sqm_1500', 'sport_count_1500', 'office_sqm_2000', 'sport_count_2000', 'office_sqm_3000', 'trc_count_3000', 'trc_sqm_3000', 'sport_count_3000', 'office_sqm_5000', 'trc_count_5000', 'trc_sqm_5000', 'cafe_count_5000', 'cafe_count_5000_na_price', 'cafe_count_5000_price_500', 'cafe_count_5000_price_1000', 'cafe_count_5000_price_1500', 'cafe_count_5000_price_2500', 'sport_count_5000', 'market_count_5000'

All the columns above were from the train dataset because we ultimately found that macroeconomic data was not the best predictor for house sale prices.

Implementations

Linear Regression

For linear regression, we chose to use the off-the-shelf implementation available in the Scikit-learn machine learning library. This allowed us to simply provide the training data and associated response variable, and get a fitted linear model with coefficients $w = (w_1, \ldots, w_p, b)$, where p is the number of variables and b is the intercept term, that minimizes the residual sum of squares between the response and predicted values.

Neural Network

The neural network was implemented using a (mostly) object-oriented approach. In total there were five classes; four for each type of layer (not including input): loss layer, output layer, activation layer, and hidden layer, and then the neural network class itself. The hidden layer and output layer classes were essentially interchangeable, both holding a matrix of outputted values, a weight matrix, and an optional velocity matrix, but were differentiated in the code for readability and the sake of completeness. Activation layers held a single matrix of outputted values, while loss layers held a scalar for average batch loss and a vector of true target values for the batch.

The NeuralNetwork class is where all computation occurs. We tried to mirror sklearn's MLPRegressor syntax for initializing and fitting the network. Other than the usual depth, width, batch size, and learning rate, other optional parameters can be passed such as the name of the activation function to be used (ReLU, leaky ReLU, tanh, or sigmoid), momentum, weight initialization method, and a multiplier used for annealing the learning rate. The two main functions were the ones for the forward pass and the backward pass. The forward pass function passes the batch forward through the network, updating the values that are saved in each layer as it goes and calculating an average batch loss at the end. The backward pass function simply starts at the loss layer, iterates backwards through the network, and depending on the type of layer, calculates and saves the Jacobian in the appropriate manner, and updates the weights if it reaches a dense layer. The fit method pulls these two together, taking in X and y, sampling a batch, and then calling the forward and backward pass methods, repeating this until the improvement in loss is less than the convergence criteria. If applicable the fit method also anneals the learning rate or uses momentum in the backward pass function. At the end, training metrics are saved which can be accessed and displayed.

Model Evaluation

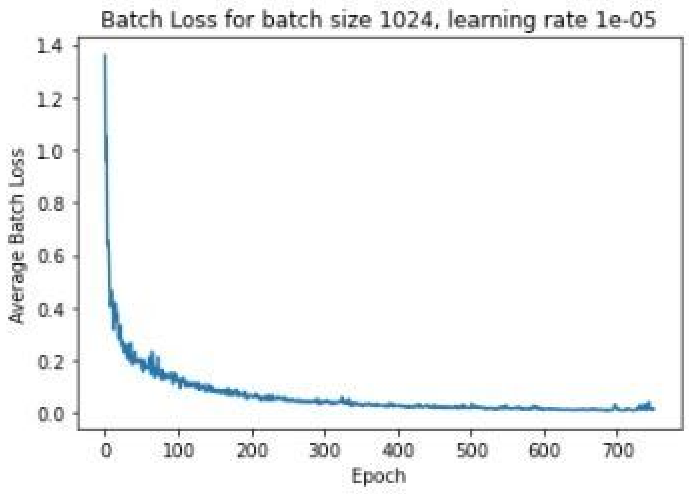
Linear Regression

For our linear regression model, we used sklearn to fit a multiple regression model with the features described above. We held out a validation set that was 10% of the full data, and trained on the rest. The model had an R^2 value of 0.56, which is surprisingly high. This means that 56% of the variation in standardized property price was explained by its linear relationship with our 50 predictors. The validation RMSE for this model was 0.487 standard deviations, which is equivalent to ₱2,327,914.22, or $30,056.21.

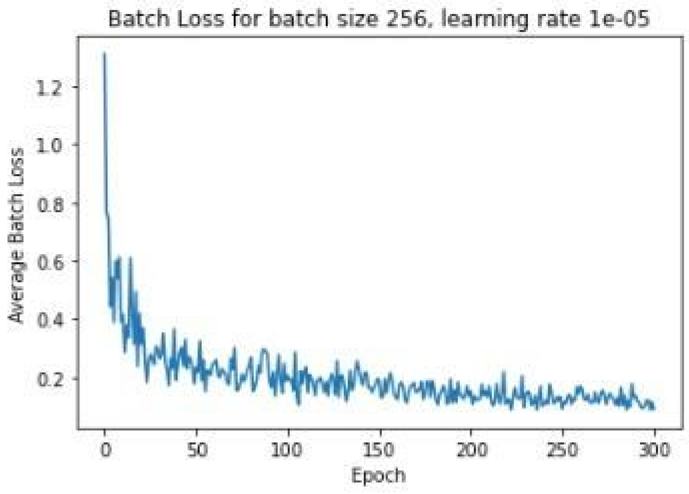
Neural Network

Finding the best configuration for the neural network proved to be difficult simply because there were so many hyperparameters: depth, width, batch size, learning rate, momentum, and convergence criteria. So, running a grid search became extremely computationally intensive very quickly. For example, if we wanted to test two values of each hyperparameter, that would result in $2^6 = 64$ models being fit. To that end, we decided to choose leaky ReLU as our activation function from the start, because it theoretically can continue learning after ReLU has become stuck with gradients of 0.

After choosing our activation function, we ran the grid search on a very limited set of hyperparameter values that we knew from our experience testing the algorithm beforehand to be effective and valid, which resulted in 864 models being tested. We found that the network which produced the lowest validation root mean-squared error (on an validation set that was a random 10% of the full data) was one that had depth 7, width 90, batch size 1024, learning rate 0.00001, momentum 0.9, and convergence criteria 0.001. We implemented and played around with annealing the learning rate, either by multiplying the learning rate by a constant λ each time, or by decreasing it when average batch loss hadn't decreased in 10 epochs, but didn't find success with either, so decided to keep a constant learning rate. Our final optimized model with the tuned hyperparameters had a validation RMSE of 0.169757 standard deviations, which is equivalent to ₽811,457.36, or $10,473.73. This may seem like a lot, but the standard deviation of prices in the data is almost five million rubles, or $80,000 so it's not too far off. Below is a plot of how the average batch loss changed over the epochs.



In terms of computation speed, having a batch size of 1024 is not ideal. However, we found that the training data was so large that anything smaller caused substantial instability in the batch-to-batch loss, because it is more difficult to obtain a representative sample as sample size decreases. Below is a plot showing the loss curve for a neural network with the same parameters as above except for batch size, which was set to 256.



Comparison

On all fronts but computation speed and complexity, the neural network out-performed the linear regression model. The neural network's predictions on the validation set deviated from the true sale price by $10,473.73 on average, while the linear regression model's predictions were $30,056.21 off on average from the true label, almost three times worse. These RMSEs are equivalent to 0.170 and 0.487 standard deviations, respectively. Despite the substantial difference in their predictive accuracy, both models did surprisingly well. When you break it down, the worse model was able to take in 50 complex features about a property, apply a relatively simple model formula, and predict the value of that property within $30,000. That's quite remarkable. And the neural network did even better.

Because linear regression is so simple compared to most models, but especially a neural network, it had the clear advantage in runtime. It trained instantly, while the neural network with the final tuned hyperparameters took about 20-30 seconds to train on average. This training time increased drastically with batch size and depth. Surprisingly, increasing the width did not affect training time much, which is probably because doing those large dot products can be done in parallel, while increasing depth means that the algorithm has to wait for the previous result in order to compute the next.

Discussion

One of the most difficult parts of this project turned out to be correctly initializing the network with the correct layer dimensions, and figuring out how the bias worked into the equation. Once the basic network was set up and running correctly, adding more and more functionality to it, such as momentum, annealing the learning rate, different activation functions, and convergence criteria, became easy. The object-oriented approach made the code relatively simple to read and understand, and made the model-fitting syntax very functional and interpretable.

One thing we did not have time to implement was weight regularization, which helps to control overfitting by subtracting a small multiple of the weight at each update step. This could have decreased our validation error even further. We would also be interested in using regularization to the linear regression with the LASSO procedure and compare that to our network, to see the effects that each has on the model.

Another difficult part of this project was feature selection. Given 289 features in the train dataset and 97 features from the macro dataset, we needed to find which subset of 50 features, of which at most 10 of them can be from the macro dataset, are the best at predicting house prices.

The amount of possible subsets is somewhere in the range of $\sum_{n=1}^{50}(289\ nCr\ n)\ +\ (97\ nCr\ (50-n))\ \approx 5.1 \text{ x } 10^{56}$. Of course, to satisfy the constraint $50-n$ cannot be greater than 10. Still, this number is extremely large and we would never be able to check each one with our models. Instead, we turned to various feature scoring methods, which brought the number of candidate feature sets down to a more manageable number where we could actually calculate the test RMSE and decide the best model.

Conclusion

Neural networks are becoming very popular in the world of data science and in this project, we were able to see exactly why that is. Compared to simple predictive models such as the linear regression model used in this project, we saw that neural networks flexibility and complexity allowed it to perform much better when given a lot of data with many features. We were able to see how flexible these neural networks are when we tuned and modified the model's parameters to fit our Russian Housing dataset best. All the parameters that go into a neural network along with forward and backward pass, allowed us to have a highly accurate model that was specific to our data.