Intro ML

Oli Callaghan, Charlie Lidbury, Rushil Patel, Maia Ramambason

November 25, 2022

Data Cleaning

Before we do anything, we try cleaning, standardising, and de-skewing the data to improve the performance of networks trained on it.

Univariate Analysis

By looking at the distribution of the input data on a per variable basis, we can spot patterns of interest and any potential problems prior to testing, which will save us time later when debugging performance. Figure 1 shows histograms of the distributions of each input variable:

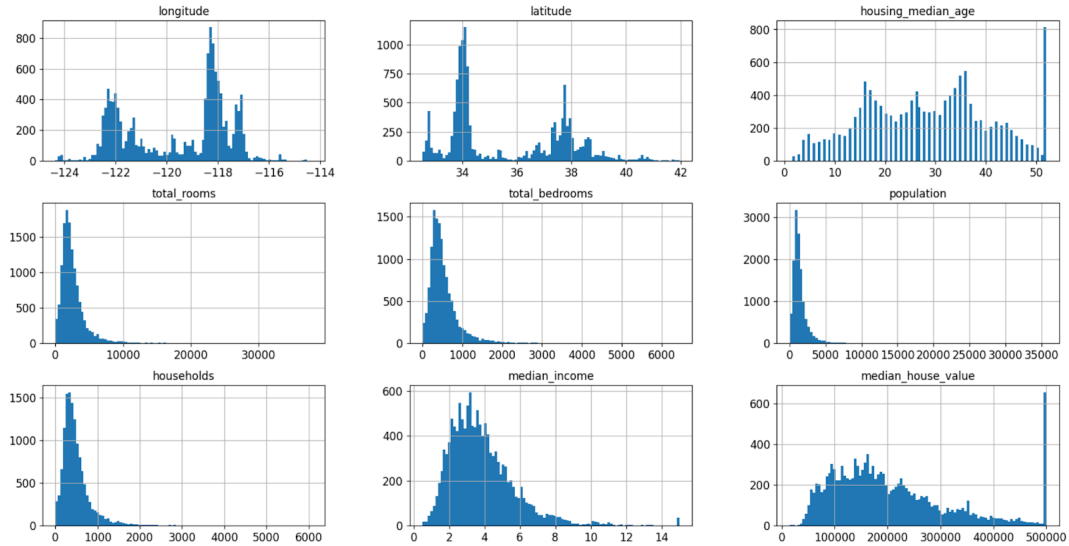

Figure 1: Histograms of each of the input variables.

Longitude and Latitude

The bimodal distributions of the longitude and latitude will slow down convergence. The data is also not taking advantage of the fact they are related. By noticing the fact that their line of best fit is the coastline, we can compute each data point's distance from the coastline via their distance from the line of best fit, which is much more precise than the Ocean Distance variable. However, this is taking advantage of very context specific details with this dataset and wouldn't generalise well to datasets where the population is not so centred along the coastline; so we have selected to not use this as a variable.

Median Age and Median House Value

These variables have evidently been clamped by whoever curated this data, as they have huge distributions on the high end. This is especially damaging because Median House Value is our target metric. To fix this we tried to remove all points that have been clamped, to prevent our model learning bad habits like the fact all houses cost at most 500,000, and hopefully increase convergence rate and accuracy. However, because the hidden test data also has this clamping applied to it, it's useful to learn these edge cases. We choose to include them in our input data.

Total Rooms, Total Bedrooms, Population, Households, Median Income

All of the remaining distributions are positively skewed. This is not a problem for neural nets because they can estimate any function; but it will make them take longer to converge. To unskew this data, we apply a Box-Cox Transform.

Correlation Analysis

By correlating each variable against every other variable, we can detect any relationships between variables and potentially take advantage of this; especially variables which correlate with our target. Figure 2 shows a heat map showing of the pairwise correlation between all of our input variables. Dark black means strong negative correlation, white means strong positive correlation.

Total Rooms, Total Bedrooms, Population, Households

The most notable feature in Figure 2 is the huge correlation with these 4 variables. Collectively they state that as the population increases, the amount of households, bedrooms and rooms. Because Total Bedrooms correlates especially well, we chose to investigate two new variables: Bedrooms per Room and Bedrooms per Population. We re-do the heat map for the new variables in Figure 3.

Bedrooms Per Person will not be useful, because it does not correlate with any other variables apart from Bedrooms Per Room; but Bedrooms Per Room correlates very well with Median House Value. Apart from Median Income, Bedrooms Per Room it is the best predictor of house value. For this reason we choose to add only Bedrooms Per Room to the data.

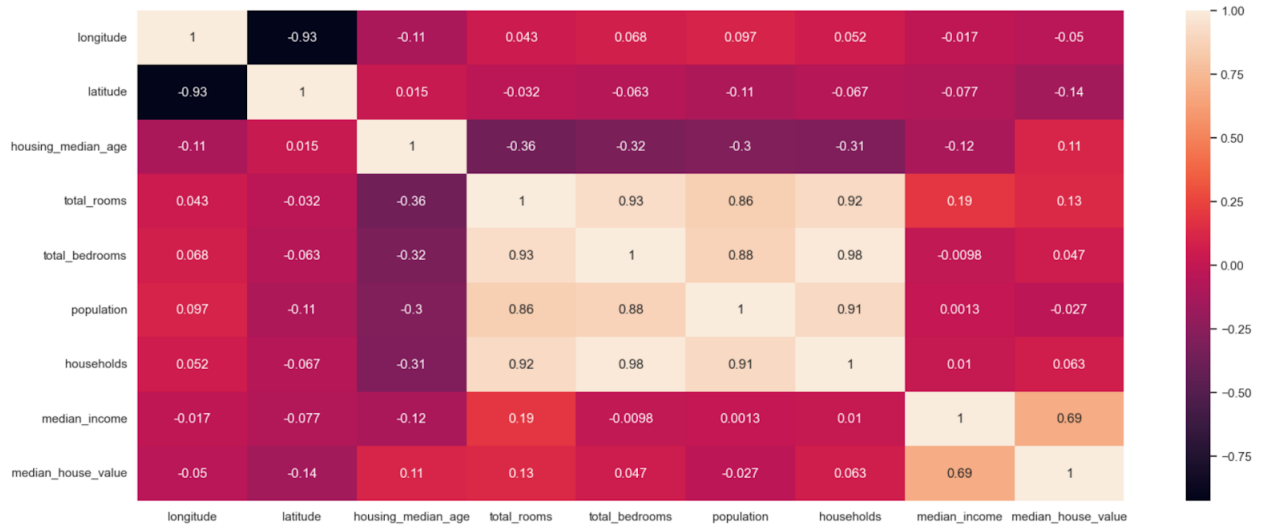

Figure 2: Heat map of pairwise correlations

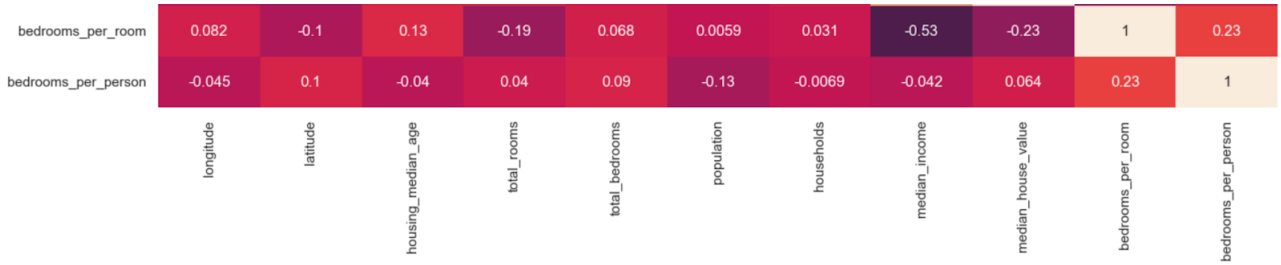

Figure 3: Heat map of pairwise correlations for added variables.

However, after adding Bedrooms Per Room to our model, it got less accurate; we believe this to be because it wasn't introducing any new information - it was already highly correlated with Median Income. The accuracy may have gone down because the size of the input increased.

Longitude and Latitude

Longitude and Latitude have a huge negative correlation because California's coastline is an approximant straight line and the population density is high on the coast. This is useless unless we choose to take advantage of this domain specific knowledge, which we choose not to for the sake of making a better generalised model.

Median Income

By far the best predictor of house value is median income, with a correlation of 0.69. This was expected, and there's not much to do with this information because the signal is already clear and the neural net will weigh it very highly already.

1 Data Standardisation

Standardisation

When preprocessing data for our model, we have chosen to utilise standardisation, rather than make use of min-max normalisation. By doing so, we are able to compare similarities between features based on certain distance measures, as each feature is fit to be modelled by the Gaussian distribution, rather than having differing distributions when using min-max normalisation. We can also retain information about outliers input into our model, something especially important for our generalising linear regression model.

We also choose to remove the skewness of the distributions via a power transformer (Box-Cox), because neural networks converge faster when their inputs are Gaussian. This isn't necessary because neural networks are capable of estimating any function, but it is useful.

Handling Missing Data (NaNs)

Through analysis on the training set, we can see that the only feature with missing data was Total Bedrooms. Earlier, we noted that Total Rooms and Total Bedrooms were highly correlated, and to reduce dimensionality, merged the two features where new values represented a ratio between the two. To fill the missing data in this new feature, we use linear extrapolation to work out the missing Total Bedrooms value using the Total Rooms value, and therefore fill the missing data values using a more reliable method than using just the mean or the median.

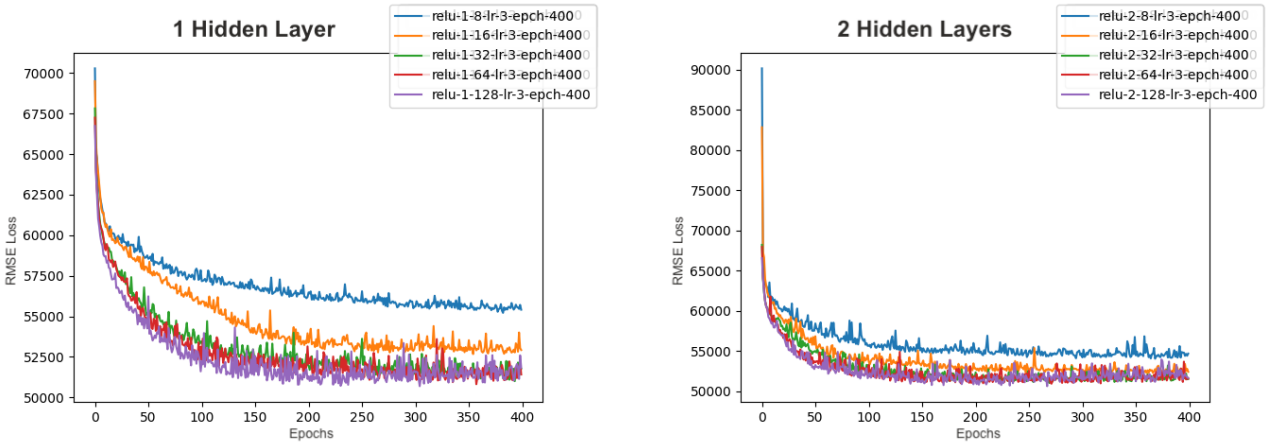

Figure 4: How size of hidden layers effects accuracy and convergence.

Model Choices and Justification

Model Choices and Training Method

This report finds that a model with 1 non-linear hidden layer with 64 neurons each performs best for predicting the median_house_value from the California Housing Dataset. The non-linear layers used ReLU (Rectified Linear Unit) as the activation function, and learnt with a learning rate of 0.001. Mini-batching with batch sizes of 16 were used, where the model was trained for 400 epochs with shuffled batches to avoid overfitting to the training dataset. A separate, held-out validation dataset was used to evaluate the performance of our model, and make the changes outlined in this report as a result.

Using RMSE as an evaluation metric

The loss function we chose for our model is the root mean squared error (RMSE). RMSE computes a value that is in the same unit as the output variable - median_house_value in this case - making it easier to reason about. We chose MSE for the loss function instead of the L1 norm since the median_house_value showed a Guassian distribution, thus we want to punish the model more for predicting further away, which allows us to more accurately generalise to house values outside of our training data set.

Model architecture choices

In order to find the optimal model architecture, we performed a search on neural network structures ranging over: number of hidden layers, number of neurons per hidden layer, and learning rate. For all of our tests, we trained networks for 400 epochs so that each had enough time to train properly before we evaluated its performance. From our search, we found that the optimal structure was a single hidden layer with 64 neurons and a learning rate of 1e-3. It is worth noting that we had chosen to set the number of neurons in each hidden layer to the same value, to reach a point where the model was slightly overfitting, so that we could later fine-tune the error rate by growing and shrinking individual hidden layers.

From the results aggregated in Figure 4, we saw that with one layer, the running average error for our model with 32 neurons was $150 less than that of 32 neurons per layer, $200 less than that of 128 neurons. When extending to two layers, we found that the average error for 2 layers of 32 neurons was $100 worse than with one layer, but still $480 and $358 more accurate than 32 and 128 neurons respectively.

There are multiple factors that may be influencing this, such as the vanishing gradient problem, where the depth of the network causes the error derivatives to vanish to zero, so the weights and thus model are never updated, however this is unlikely due to the size of the network. Alternatively, it could simply be a result of overfitting to the data. This factor is much more likely to be the driving factor, since when comparing the average RMSE of training data with evaluation data for both numbers of layers (seen in Figure 5), we find that structures with 2 hidden layers diverge much more than those with 1 hidden layer.

Activation Function

Our next decision was to determine the type of activation function we would use. In our model discovery we opted to use rectified linear units (ReLU) since we were training a regression model. ReLU is more suited to regression than other activation functions such as the hyperbolic tangent function, and the sigmoid function, which are more commonly used in classification models. In addition to ReLU, we wanted to understand whether the sigmoid linear unit (SiLU) activation function would perform more effectively than ReLU, so we trained two models, one with ReLU and one with SiLU to compare.

The nature of the ReLU activation function can lead to dead neurons - neurons that output zero regardless of input - in our network. An alternative activation function we used to mitigate this was SiLU (Swish), developed by

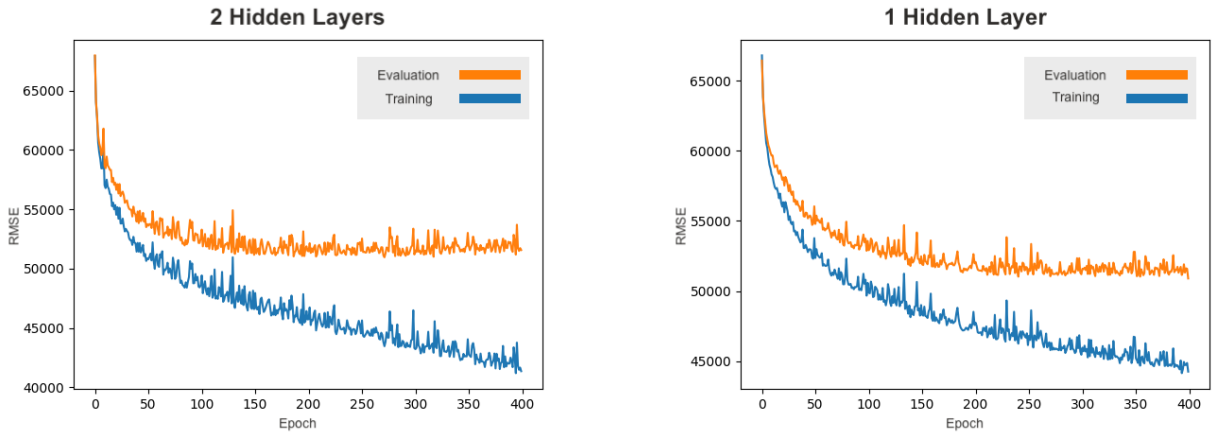

Figure 5: How hidden layers effect accuracy, convergence, and over fitting.

Google Brain in 2017. SiLU is a gated version of the Sigmoid function, and is unbounded above as a result. The idea is that SiLU has a global minimum value that is less than zero, and has a self-stabilising property because of its non-monotonic nature.

For 400 epochs, we found that ReLU produced a more accurate model compared with SiLU, however from the graphs of RMSE against epoch number in Figure 6, we noticed that the difference between training and evaluation for SiLU was closer together, and still training at 400 epochs, so we opted to train the model for an additional 400 epochs (800 epochs).

From the results of 800 epochs, we can see that SiLU is taking longer to train, and producing results with much less variation than in ReLU. Upon further inspection, it is evident that the ReLU accuracy decreases after 400 epochs, suggesting overfitting, which can be seen by the divergence in the training and evaluation data accuracies for ReLU. Therefore, we selected using ReLU for our model.

Learning Rate In order to determine the optimal learning rate for our model, we included learning rate a search parameter for our overall network structure. This is because we found that the performance of a network architecture was a function of not only structure, but also learning rate. The results of training with different learning rates can be seen in Figure 7. From these graphs, we can see that training with a learning rate of 0.001 produces a much more accurate model, compared with 0.0001 and also in 0.00001. This could be because the gradient descent algorithm gets stuck in local minima more often when using a smaller learning rate, whereas when using a larger learning rate, the algorithm can 'skip' over these local minima.

Optimiser In order to optimise our neural network, we decided to use the stochastic gradient descent algorithm. We chose this over

Batch Sizes Rather than computing the gradient required for the gradient descent algorithm once the entire dataset is used for training, which would require an enormous amount of memory, we can update the model's parameters more frequently by making use of mini-batching. This reduces the risk of getting stuck at a local minimum, as random batches from a shuffled dataset are considered at every epoch, ensuring a robust convergence.

Figures 8 show us the difference in performance when varying batches. We can see that lower batch sizes allow for quicker training time, and the error rate reduces drastically with fewer epochs. Therefore, we have chosen a batch size of 16, which is a suitable compromise for our model.

Final Evaluation After hyperparameter tuning, we scored our model by applying a 10-fold cross-validation to our best model, and then taking the average over the iterations for the root mean squared error. We found that after 400 epochs, this value was $44036.

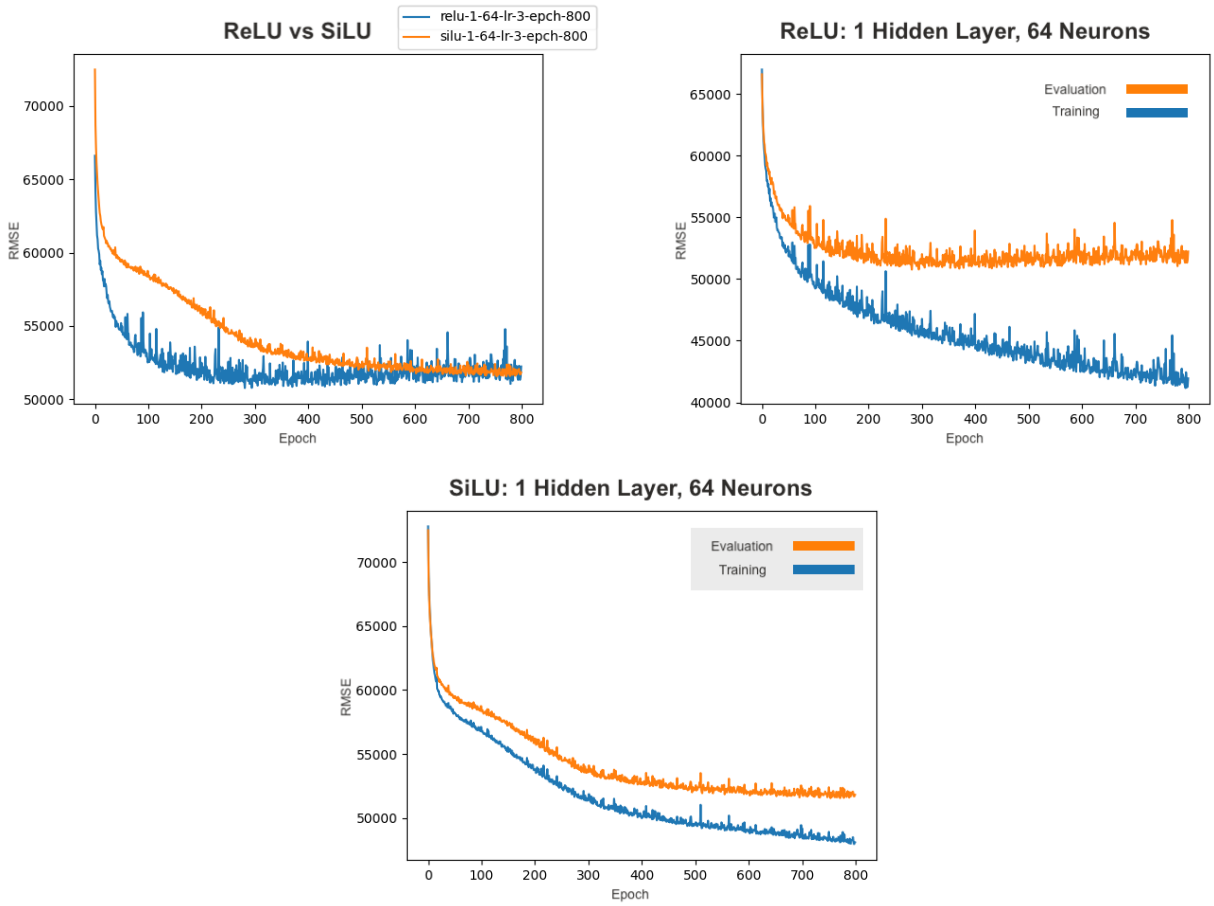

Figure 6: Effect of choice of activation function on convergence, accuracy, and over fitting.

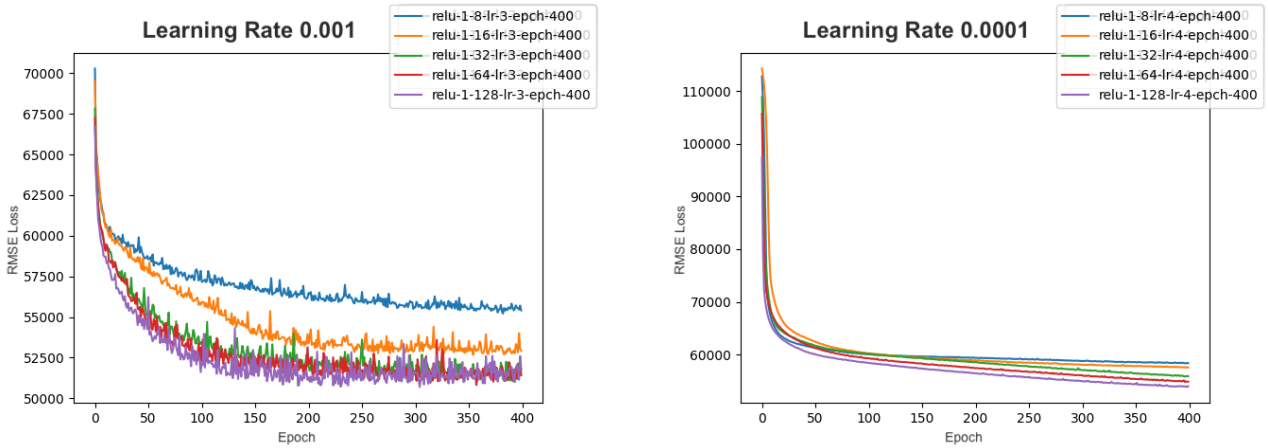

Figure 7: Effect of choice of learning rate on model accuracy.