Machine Learning Coursework 2 - Neural Networks

David Cormier, Thomas Loureiro van Issum,
Petra Ratkai, Simon Staal

November 2021

1 Introduction

The aim of this project is to train a neural network to predict the average price of a house in California based on a collection of general characteristics that were included in the 1990 census. This is implemented in Python aided by functions provided by the Scikit-Learn and PyTorch libraries. The model is a multi layer neural network that consists of: an input layer, 1 hidden layer and an output layer. The individual layers are linear layers with identity activation functions, with the exception of the output layer which uses ReLU activation.

2 Implement an architecture for regression

2.1 Preprocessor Method

The preprocessor method operates in 3 stages: imputation, one-hot encoding and normalisation. The first stage handles missing values in columns. This is done using the mean value of the column for continuous features, and the modal value for discrete features (i.e. the "ocean_proximity" feature). As such, the "ocean_proximity" feature was handled separately, and a scikit SimpleImputer [1] was used to handle the remaining columns. Following this, one-hot encoding was performed on the "ocean_proximity" feature using a scikit LabelBinarizer [2]. The possible feature values were hard-coded to the 5 seen in the data: `['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN']`. Using a 6th column to handle unseen values, or empty values as mentioned above, was considered but discounted to keep the model simple. Furthermore, it could be assumed that only the 5 discrete values seen in the data-set would be present during testing.

Finally, the remaining columns as well as the output values were normalised using sklearn MinMaxScaler [3] classes, which perform min-max standardization on each column of the data-set independently. This ensures all features are on a single scale of 0 to 1, preventing larger data values influencing the data by a disproportionately large amount, which should help gradient descent converge faster towards a minima. If training mode was enabled, all the classes listed above were fitted on the data-set, the modal value for "ocean_proximity" was calculated, and were stored as member variables of the Regressor class. This allows for the same normalisation that was applied to the training data to be applied during testing to ensure that the model can generalize well to new, unseen data points. The new, normalised data-sets are then converted to tensors and returned.

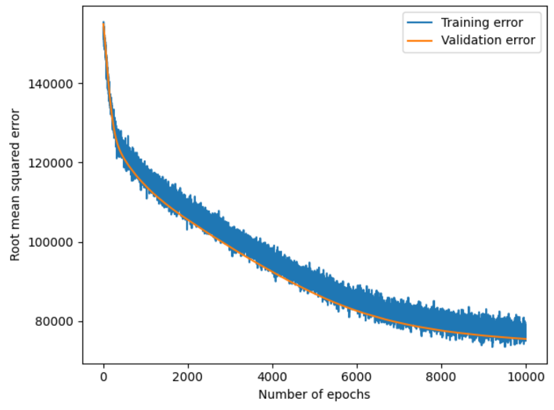
2.2 Constructor Method

The constructor has several additional parameters: the learning rate, neural-network architecture, batch size, and loss function. These were implemented in order to facilitate hyperparameter tuning. The classes required for pre-processing outlined in section 2.1 are initialised, before the data-set being passed into the constructor is processed to determine the input size of the model. The architecture of the network is specified using a list of integers, each representing the number of neurons desired for a given layer (i.e. the input `[8, 8]` would represent 2 layers each consisting of 8 neurons). The network itself is implemented using sequential linear layers [4] with linear activation functions, followed by a final linear layer consisting of 1 neuron (the output size) with a ReLU activation function [5]. The linear and ReLU activation functions are ideal for a regression model [6]. In particular, the ReLU function was chosen for the output layer as a positive real valued result was desired, i.e. the predicted house price. Other activation functions, such as sigmoid, were briefly tested, but as they are more suited to classification tasks, they performed worse and so were not investigated further. Improvements to this architecture could have been found with more robust testing of different activation functions within the hidden layers, but this was not performed due to time constraints.

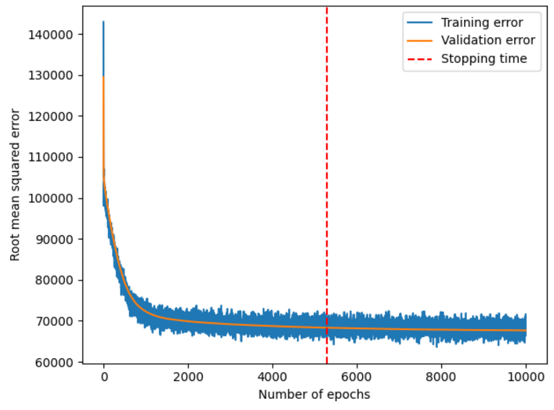
These layers are all stored in a pytorch Sequential container class [7], which chains the outputs of a layer to the input of the following layer. The weighting of each linear layer is initialised via the xavier_uniform method [8], and the biases are set to 0. Xavier initialization was selected to ensure the weights are in the correct range of values. Generally, it reduces the likelihood of weights starting in either a dead or saturated region, ensuring they can impact the final model [9]. Since asymmetry breaking is already provided by the randomization of the weights, the biases were initialized to 0 following standard practice [10]. Finally, other parameters, including the number of epochs, learning rate, batch size and loss function are saved according to the value passed to the constructor.

2.3 Model-training Method

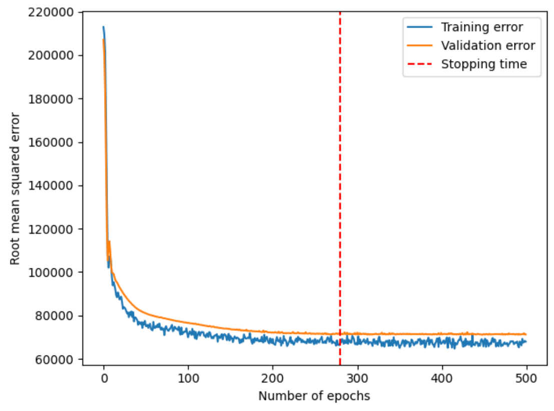
To train the model, a mini-batch gradient descent approach was selected. This has several advantages over stochastic and batch gradient descent, as it is computationally efficient whilst still benefiting from vectorization, and has the ability to escape local minimums through noisier steps whilst still producing error gradients stable enough to converge nicely [11]. In each epoch, the training data was shuffled before being split into multiple batches that were passed into the network, where forward and backwards propagation was performed for each batch. The fit function also incorporates the option to pass in a validation or development data-set to be used to perform early stopping. This feature tests the performance of the network on held-out data to ensure the model does not over-fit the training data, as well as save time during hyper-parameter tuning by allowing the model to stop when it's performance was not improving. The best performing version of the NN is saved, and if after a set number of epochs (stored in `self.early_stop`), the validation performance did not improve, the saved 'best' copy of the neural net is returned.



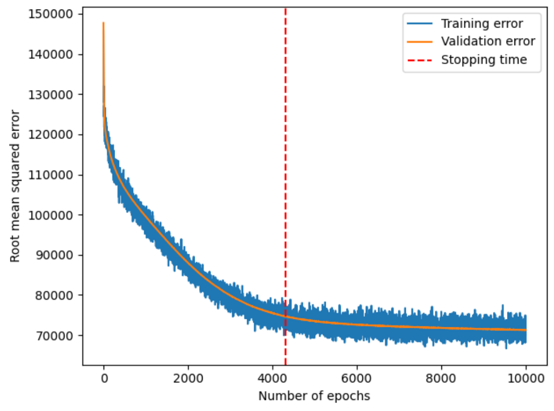
((a)) Error varying by epoch - Adadelta Optimiser



((b)) Error varying by epoch - Adagrad Optimiser



((a)) Error varying by epoch - Adam Optimiser



((b)) Error varying by epoch - SGD Optimiser

The pytorch library offers many optimisers [12], and several, as seen above, were tested to update the parameters for each layer for the backwards pass. Stochastic gradient descent (SGD) [13] is a simple optimiser that performs gradient descent based on the gradient at given datapoints. The Adam [14], AdaGrad [15] and Adadelta [16] optimisers add finer control over the gradient descent, notably by using a different learning rate for each variable.

The previously mentioned optimisers were tested to find their performances, and were plotted above (figure 1). The Adadelta Optimiser performed the worst, never stopping even after 10000 epochs. In comparison the Adagrad and SGD optimiser had a relatively similar stopping time, however the former only took 1000 epochs to reach an error of around 70000 whilst the latter did not manage to reach a training error that low before it was stopped. Finally the Adam optimiser had the swiftest stopping time of around 300 epochs, with a training error just below 70000. As such the Adam optimiser was selected for usage, as its rate of convergence far outclassed the others.

Another optimiser that was considered was the Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (LBFGS) [17], which, unlike the previous methods, determines the descent direction by preconditioning the gradient with curvature information using an approximation of the Hessian matrix of the loss function. Using this information, it is typically able to converge faster than the first-order methods mentioned above [18]. Unfortunately, the PyTorch implementation of this optimiser does not work for mini-batches [19], and whilst mini-batched versions of this optimiser are available elsewhere, due to the specification limitations these could not be used. As a result, this optimiser was not tested as the training time significantly increased.

3 Set up Model Evaluation

3.1 Prediction Method

To use the model to predict any given output, the input is first preprocessed to normalise all the values once again. This normalisation is identical to that which was applied to training set, or two identical points in different sets would appear different to the network. Then this is passed forward in order to produce the predicted values. These values are then reverted using the inverse transformation to ensure that they are directly comparable with the target values.

3.2 Evaluation Method

Our implementation uses Root mean square error as the evaluation metric. There were multiple other potential options such as: mean squared error, mean absolute error, root mean squared log error, and R^2. Each metric has their own unique benefits and downsides, however each was discarded for various reasons. R^2 measures how well the model uses the features by comparing the output to the mean of the correct y values and correcting for the number of input variables. However, the metric is bounded by [0,1] instead of being a dollar value, which makes the score less valuable as it is less capable of demonstrating the actual difference between predicted and actual value. It is also not adapted to this coursework, since we care about maximising performance with a fixed data-set rather than about finding which features are most informative and worth gathering data for. Similarly, neither MSE nor RMSLE provide a metric that provides a score directly in dollars. RMSE generally places higher value on larger errors, due to squaring them. This is particularly useful as we desire to strongly penalize large errors, ensuring they are clearly visible in the final metric.

4 Perform HyperParameter Tuning

4.1 Methodology for tuning

There were a variety of potential methods to implement hyperparameter tuning, such as random search, grid search, and manual tuning. Random search is generally feasible however it offers the least insight into the results and simply provides the best set of parameters with little additional information as such, it was deemed undesirable. Manual tuning offered far greater insight into the direct impact of each hyperparameter, which is highly desirable. However with this method it would be impossible to fully test all hyperparameters. Instead each hyperparameter would have to be tested sequentially to isolate their impacts on the model as a whole. Whilst valid this alone would likely result in not being able to find the exact optimal values for each hyperparameters and thus a worse final model.

Grid search is capable of generating the optimal hyperparameter values for any given machine learning model. This is done by systematically varying each hyperparameter and building a model, then storing it. These models can be evaluated in order to identify the set of hyperparameters that produces the single best model. The primary downside to this is the computational power to create each model, however this is not a particularly relevant negative for the coursework, due to the size of the data being used as well as the time available to conduct the testing.

As such we decided to use a combination of grid search as well as manual tuning. Grid search would be used initially to provide baseline hyperparameter values. Subsequently each hyperparameter would be varied individually to ensure that no further optimisation could occur. The manual tuning was averaged across 10 different set of results to ensure the model that performs the best consistently was selected.

In order to implement grid search, scikit offers a model selection class GridSearchCV [20] was used, which takes in an estimator (the regressor model being tuned) and a set of parameters to be tested. This requires the estimator to be compatible with the scikit-learn estimator interface, and as such, 2 additional methods, `get_params()` and `set_params`, which allow the GridSearchCV class to make copies of the regressor. Since the regressor constructor requires a dataset, a copy of the input data-set was stored when constructing the regressor so that it could be passed onto the contstructor of the copies generated by the grid search class through the `get_params()` method. Ideally, the regressor implementation would be refactored so that the regressor constructor does not require a dataset to be provided as an input, but to maintain compatibility with the specification this was not done.

The hyperparameters varied were the batch size, learning rate, number of hidden layers, number of neurons per hidden layer, and number of epochs. All of these parameters were tuned in the grid search cross validation, except the number of epochs, which was tuned separately. This was because, in order to avoid over-fitting and for the tests to run in a reasonable time-frame, early stopping was utilised during hyperparameter tuning. A validation data-set (separate from the data-set passed to the grid search) was used to determine when the training should stop. When the validation error did not improve over a certain number of epochs, the training would stop and it would return the best model obtained. Based on the results obtained in figure 1, it was noted that the validation data-set performance experienced minimal variation, and as such the waiting period was set to be 20 epochs. Once the optimal learning rate, network architecture and batch size were determined and fine-tuned, the ideal number of epochs was found by training the model using the best parameters 10 times, and averaging the stopping times. This value was then passed in as the number of epochs for the final, fully-tuned regressor.

4.2 Results Analysis

The grid search performed a sweep of the following parameters:

1. Learning rate: 0.001, 0.005, 0.01, 0.05, 0.1, 0.2
2. Batch size: 64, 128, 256, 512, 1024, 2048
3. Neuron combinations: [13], [10, 10], [8, 8], [8, 8, 8], [13, 8], [9, 4], [13, 9, 4]

The neuron combinations were selected based on various ”rules of thumb” found, typically involving a relationship between the number of input features and desired number of outputs.

Once the grid search produced optimal values for the hyperparameters, these were further tested manually by averaging the model's average loss metric. These loss metric were averaged over 10 iterations of the model for each set of hyperparameters.The data for the manual tuning on the final model can be seen below:

Neuron per Layer	Average RMSE
[13,8]	69709.87
[12,8]	69475.26
[11,8]	69701.39
[10,8]	69968.13
[9,8]	70298.15
[8,8]	69626.42
[7,8]	69866.99
[6,8]	70163.44
[5,8]	69966.49
[4,8]	69801.03
[3,8]	70029.91
[2,8]	70232.48

((a)) Varying the first layer.

Neuron per Hidden Layer	Average RMSE
[12,13]	69824.34
[12,12]	69885.69
[12,11]	69862.54
[12,10]	69616.22
[12,9]	70028.53
[12,8]	69544.45
[12,7]	70148.79
[12,6]	69987.93
[12,5]	69964.21
[12,4]	70059.39
[12,3]	69865.91
[12,2]	70059.31

((b)) Varying the hidden layer.

The manual test data above provided one improvement to the hyperparameters found by the grid search. Specifically it demonstrated a reduction in size of the first layer would produce a slightly lower average RMSE. Apart from

Batch Size	Average RMSE
16	70408.48
32	71454.74
64	70224.65
128	70233.02
256	70004.86
512	69595.59

((a)) Varying the Batch Size.

Learning rate	Average RMSE
0.001	69544.45
0.005	70158.16
0.01	69809.89
0.05	71703.19
0.1	72750.52

((b)) Varying the Learning Rate.

this the remaining manual tuning proved that the other hyperparameters were optimal as no changes could produce a lower average RMSE.

5 Evaluation of Final Model

Once the model hyper-parameters were tuned, a final regressor was trained with all parameters specified, this time using the entire data-set as training, with no data-sets set aside for validation or testing. Our final model using the following hyper parameters: 272 epochs, layers of [12,8], learning rate 0.001, batch size 512. This model predicted the house values with an average RMSE of 68267.54 on the **training data-set**. Separate evaluation was done using a held-out test set with a seperate model, which used the same parameters but used a subset of the data, and obtained an average RMSE of 69865.16. We expect our final model to perform marginally better, as it is trained using more data, but should have a performance closer to the latter value rather than the former.

Given the median house values range between 15000 and 500000, this RMSE is a reasonable error rate. As such we can conclude that the regressor is capable of making meaningful predictions about the median house values from the data provided.

References

[1] "sklearn.impute.SimpleImputer", scikit-learn, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html#sklearn.impute.SimpleImputer. [Accessed: 26- Nov- 2021]

[2] "sklearn.preprocessing.LabelBinarizer", scikit-learn, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelBinarizer.html. [Accessed: 26- Nov- 2021]

[3] "sklearn.preprocessing.MinMaxScaler", scikit-learn, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler. [Accessed: 26- Nov- 2021]

[4] "Linear — PyTorch 1.10.0 documentation", Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear. [Accessed: 26- Nov- 2021]

[5] "ReLU — PyTorch 1.10.0 documentation", Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html#torch.nn.ReLU. [Accessed: 26- Nov- 2021]

[6] S. Ronaghan, "Deep Learning: Which Loss and Activation Functions should I use?", Medium, 2021. [Online]. Available: https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8. [Accessed: 26- Nov- 2021]

[7] "Sequential — PyTorch 1.10.0 documentation", Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html#torch.nn.Sequential. [Accessed: 26- Nov- 2021]

[8] "torch.nn.init — PyTorch 1.10.0 documentation", Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/nn.init.html. [Accessed: 26- Nov- 2021]

[9] "An Explanation of Xavier Initialization", andy's blog, 2021. [Online]. Available: https://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization. [Accessed: 26- Nov- 2021]

[10] "CS231n Convolutional Neural Networks for Visual Recognition", Cs231n.github.io, 2021. [Online]. Available: https://cs231n.github.io/neural-networks-2/#ihttps://cs231n.github.io/neural-networks-2/#initnit. [Accessed: 26- Nov- 2021]

[11] D. Kapil, ”Stochastic vs Batch Gradient Descent”, Medium, 2021. [Online]. Available: https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1. [Accessed: 26- Nov- 2021]

[12] ”torch.optim — PyTorch 1.10.0 documentation”, Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/optim.html. [Accessed: 26- Nov- 2021]

[13] ”SGD — PyTorch 1.10.0 documentation”, Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html#torch.optim.SGD. [Accessed: 26- Nov- 2021]

[14] ”Adam — PyTorch 1.10.0 documentation”, Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam. [Accessed: 26- Nov- 2021]

[15] ”Adagrad — PyTorch 1.10.0 documentation”, Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html#torch.optim.Adagrad. [Accessed: 26- Nov- 2021]

[16] ”Adadelta — PyTorch 1.10.0 documentation”, Pytorch.org, 2021. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adadelta.htmltorch.optim.Adadelta. [Accessed: 26- Nov- 2021]

[17] Fletcher, Roger (1987), Practical Methods of Optimization (2nd ed.), New York: John Wiley Sons, ISBN 978-0-471-91547-8

[18] ”Improving LBFGS algorithm in PyTorch”, Sagecal.sourceforge.net, 2021. [Online]. Available: http://sagecal.sourceforge.net/pytorch/index.html. [Accessed: 26- Nov- 2021]

[19] J. Haupt, ”Optimizing Neural Networks with LFBGS in PyTorch”, Johannes Haupt, 2021. [Online]. Available: https://johaupt.github.io/python/pytorch/neural%20network/optimization/pytorch_lbfgs.html. [Accessed: 26- Nov- 2021]

[20] ”sklearn.model_selection.GridSearchCV”, scikit-learn, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV. [Accessed: 26- Nov- 2021]