Coursework 2: House Value Regression

Haaris Khan (hhk20), Charmaine Louie (ccl19), Alexandra Neagu (an720), Ishaan Reni (ir320)

Introduction

The aim of the coursework was to create and train a neural network to predict Californian house prices. The neural network was trained using data from the 'California Houses Prices Dataset'. The model was created using various Python libraries such as *PyTorch* and *scikit-learn*. A hyperparameter search algorithm was implemented to generate a set of hyperparameters which would result in a minimum RMSE for the regression model.

Approach

The approach taken to construct an optimal neural network model includes a pre-processor, regressor, and a hyperparameter search.

Pre-processor

The pre-processor prepares the raw data through a series of steps.

1) A *scikit-learn LabelBinarizer* instance is fit using all possible *ocean_proximity* strings, and then training and test data are transformed so that *ocean_proximity* is replaced with one-hot-encoded columns.
2) Missing values are imputed with the median values from the training set columns, using *scikit-learn SimpleImputer*. Median is a better reflection of the average than mean in this case as the data is skewed (not exactly normally distributed). Though test data is obtained at runtime, imputation is performed using training data medians. As both test and training data should theoretically have the same median as they come from the same distribution. Moreover, if the test data is small, the median of that may be unrepresentative. Which could lead to problems such as the one-hot-encoding having '1' in two ocean proximities for the same row. Median imputation has some downsides such as making the features less related (by averaging only 1 variable in multivariate data), but it is still better than other methods which add statistical bias to the data (such as filling with zeros).
3) A *scikit-learn MinMaxScaler* instance is fit to the training data, and then the training and test data are then transformed (using the training set parameters), such that the training data is $\in [0, 1]$, which helps gradient descent by curtailing oversized features.
4) The data are transformed into Torch tensors and then returned.

Regressor Class

The *fit()* function begins by splitting the data from the pre-processor into the validation and training datasets. The training datasets are split into mini batches to loop over in every epoch. For each epoch, the mini batches are looped through. For each mini batch set, the gradients are set to zero, the loss is computed via backpropagation, and then the optimizer updates the weights and biases. Early stopping is used to prevent overfitting. Validation loss is calculated for every epoch to check for convergence. If no improvement is made for *patience* epochs, training terminates.

predict() loops through every layer in the model and completes a forward pass for each layer. For the input layer and every hidden layer, the activation function (ReLU by default) is applied over its respective linear function. Whereas the output layer just has the linear function applied to it.

score() runs the *predict*() function on the dataset to generate the predictions and then calculates their respective loss by using the set criterion (*torch.MSELoss* in our case), and the expected values.

Training of Regressor Model

In Figure 1 and Figure 2, some of the top performing models and their training step over time are being presented.

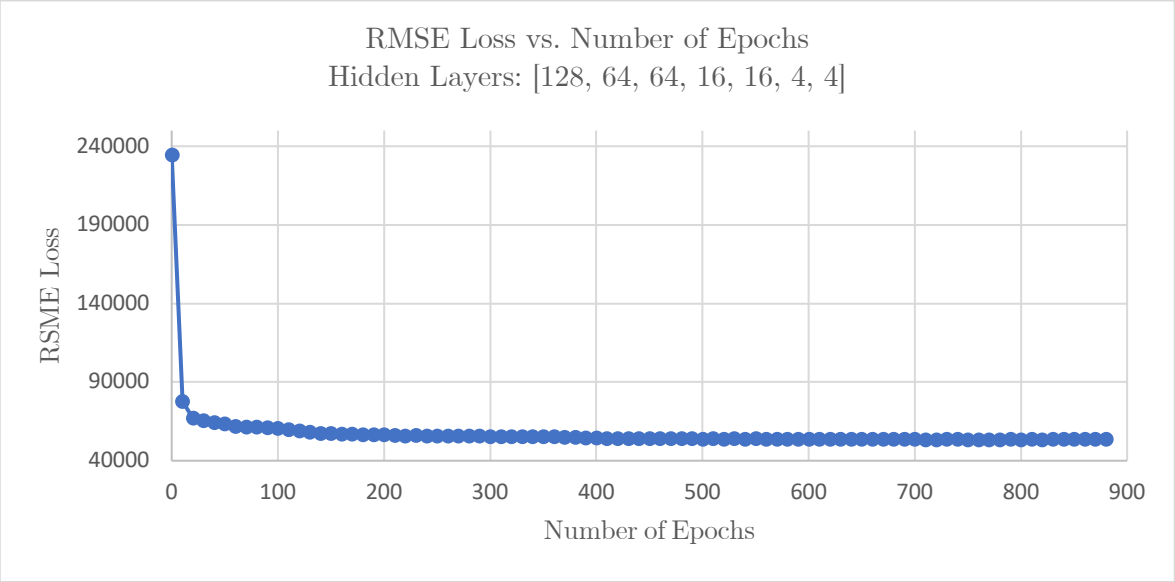

Figure 1: RMSE Loss vs. Number of Epochs for hidden layers: [128,64,64,16,16,4,4] on validation dataset

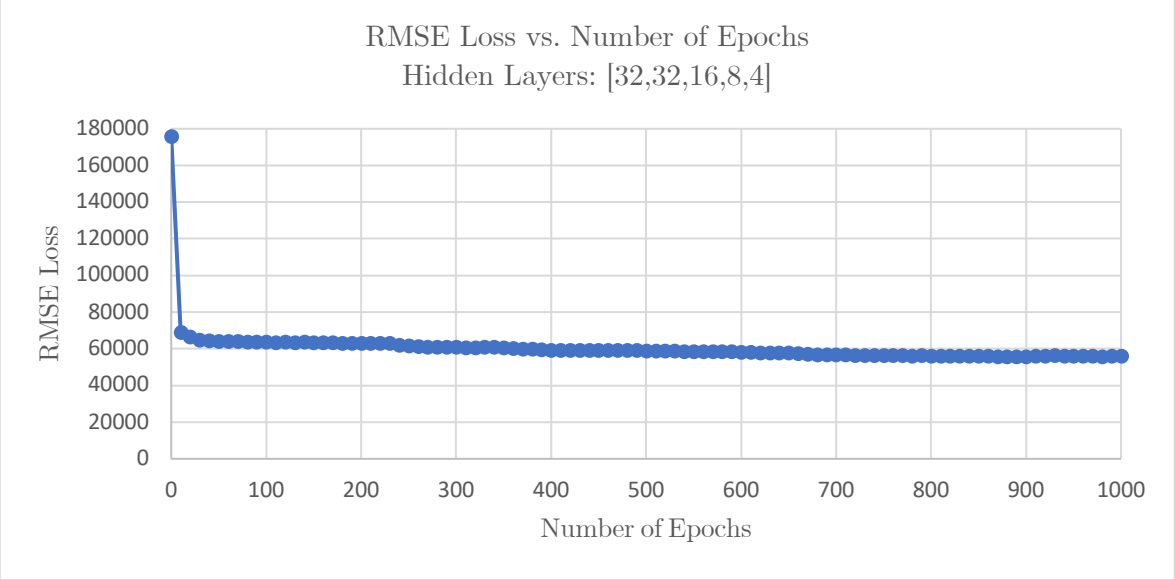

Figure 2: RMSE Loss vs. Number of Epochs for hidden layers: [32,32,16,8,4] on validation dataset

Early stopping is used (with the *patience* variable) to terminate training at the optimal time. From the plots, it can be seen that the RMSE plateaus for many epochs before ever increasing (overfitting). This precludes the need to use other regularisation methods (L2 loss, dropout, etc.). Having many samples and few features may have helped avoid overfitting here.

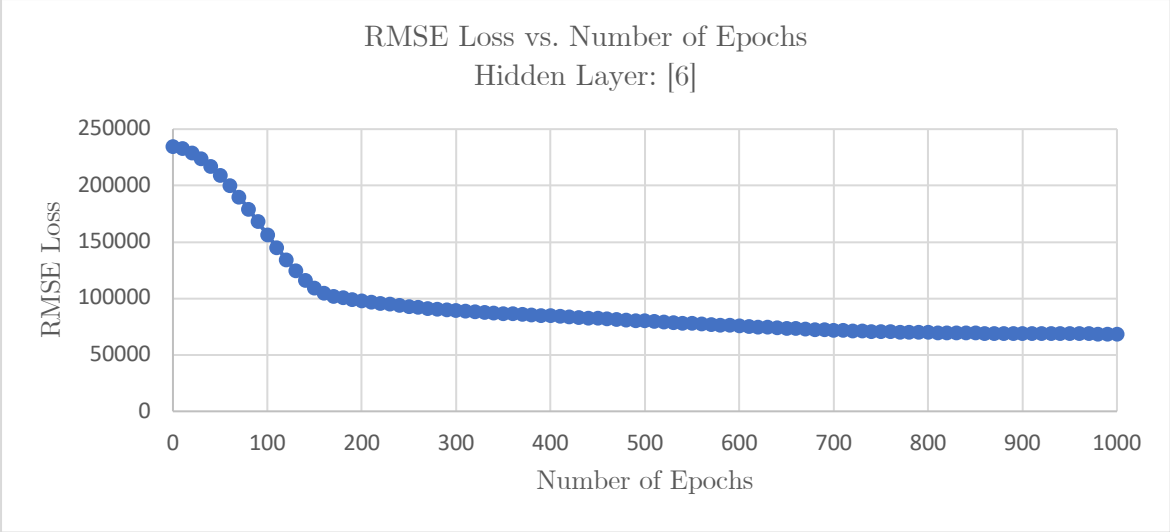

Figure 3: RMSE Loss vs. Number of Epochs for hidden layers: [6] on validation dataset

One point to notice is the model in Figure 3. This model only presents one hidden layer, however throughout the same training period it performs very similarly to the more complex neural networks from Figures 1 and 2. It can be seen that Figure 3 presents a slower learning in the beginning epochs; however, the rate of decrease remains significant throughout the rest of the training process.

Constructor hyperparameters

Table 1 shows a list of descriptions of some hyperparameters that are used to instantiate a regressor when the constructor is called, along with example values.

Argument	Default Value	Description
nb_epoch	99999	The default value of the number of epochs should never be reached, as early stopping is implemented.
patience	200	The number of epochs to wait before terminating training, after model stops improving on validation set.
nb_neurons_per_hidden_layer	[32,32,16,8,4]	List containing n elements signifying the hidden layers with values signifying the number of neurons per layer.
Minibatch_size	128	Usually set as a power of two for full hardware utilization.
lr	0.005	Learning rate for each step. Will be modified by the optimizer during training.
activation	*torch.nn. ReLU*	It was found that *ReLU* outperformed the other activation functions that were available for regression.
optimiser_name	*torch.optim. Adam*	*Adam* and *RMSProp* have both proven to be the best optimisers that were available. *Adam* was set to be the default, as it tends to work more reliably compared to *RMSProp*. The optimiser is responsible for deciding the best steps for altering the learning rate.
criterion	torch.nn. MSELoss	MSELoss was found to be the best criterion for regression. It was also possible to get the RMSE loss easily from this, making it a beneficial choice.

Table 1: Hyperparameters Descriptions

Model Evaluation

The score function was used to evaluate the model generated. It returns the MSE loss of the model, which is calculated using the *MSELoss PyTorch* criterion. For the final evaluation, RMSE is used instead of MSE as RMSE measures the average magnitude of error [1]. It also gives relatively higher weightings to higher errors which allows for easier error detection. (RMSE is different from absolute mean dollar error, which is given by L1 loss.)

Hyperparameter tuning

Hyperparameters are settings that influence the performance of the model when training. Therefore, the *RegressorHyperParameterSearch()* function was used to find an optimal set of hyperparameters. To save time, *scikit-learn HalvingGridSearch* was considered, however it prevented *minibatch_size* from being in the search, as the routine itself modified the batch sizes given to the models. The implemented search function instead uses *scikit-learn GridSearchCV*, which exhaustively tests different combinations of hyperparameters and finds the optimal estimators based on RSME. Time for hyperparameter tuning was significantly reduced by using multi-threading, which is performed by the *GridSearchCV n_jobs* argument. The search space includes relatively good values that were found through manual trials (shown in Table 2).

minibatch_size	[128, 256, 512, 1024]
lr	[0.05, 0.005, 0.0005]
optimizer_name	[torch.optim.Adam, torch.optim.RMSprop]
patience	[10, 50, 100, 200]
nb_neurons_per_hidden_layer	[6], [12, 4], [12, 8], [13, 8, 4], [64, 64, 32], [8, 4, 3, 2], [16, 16, 8, 4], [32, 32, 16, 8, 4], [128, 64, 32, 32, 8, 8, 4], [128, 64, 64, 16, 16, 4, 4]

Table 2: List of parameters added to GridSearchCV function

There were 2 different optimizers which were considered for this model. *RMSprop* is an optimizer which uses an adaptive learning rate which is based on the first moment (mean) [2]. Similarly, to *RMSprop, Adam* is an optimizer which also uses an adaptive learning rate however, it uses exponential moving averages for gradient descent calculations [2].

Through manual trials, a set of candidate layer architectures was produced. A range of numbers of hidden layers were considered for investigation on the effect on training. Equation 1 presents a heuristic used to find the number of hidden layers which would be able to prevent overfitting [3]. N_s represents the number of samples in the training dataset, N_i and N_o represent the number of input and output neurons respectively.

Although many popular heuristics describe neuron numbers similar to the input size, larger capacity models were also tested. While those models took longer to train and risked overfitting, they were shown to converge to decent RMSE values at the end of training, making them viable for the search. The number of neurons per hidden layer were usually designed in a decreasing fashion as a heuristic to aid the notion of feature extraction (where features form combinations down to the output layer). Repeating the number of neurons per hidden layer also proved to decrease RMSE while trails were being conducted.

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

Equation 1: Finding the number of neurons in the hidden layer to prevent overfitting [3]

Figure 4 and Figure 5 represents the variation of RMSE with different hyperparameters. The correlations in the graphs below are weak, due to the fact that most hyperparameters are dependent on each other. Therefore, it is difficult to find the effect of changing one hyperparameter on the overall performance of the model.

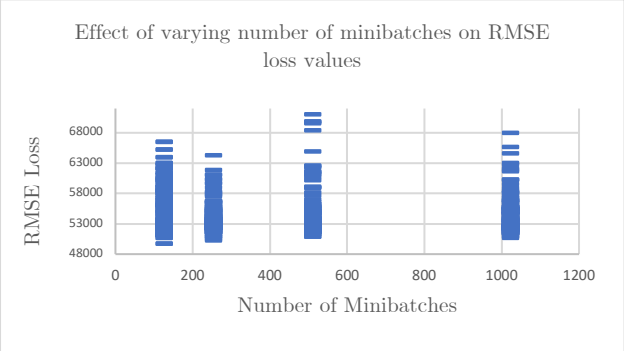

Figure 4: RMSE for different numbers of minibatches

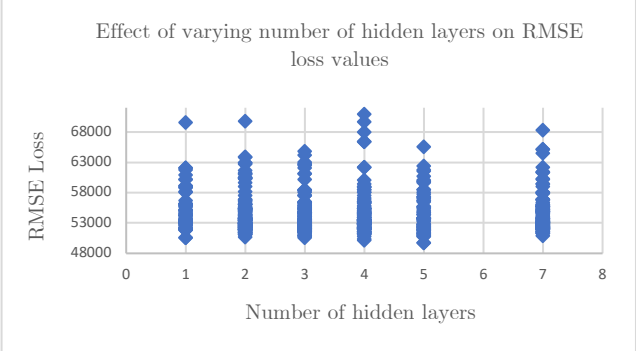

Figure 5: RMSE for different numbers of hidden layers

Table of Hyperparameter Results

Table 3 shows the top rows of the results of the hyperparameter search, sorted by RMSE.

lr	minibatch_size	nb_neurons_per hidden_layer	optimizer	patience	RMSE
0.005	128	[32, 32, 16, 8, 4]	Adam	200	49684.21
0.005	128	[32, 32, 16, 8, 4]	RMSprop	200	49684.21
0.05	256	[16, 16, 8, 4]	Adam	100	50188.92
0.005	256	[16, 16, 8, 4]	Adam	100	50372.92
0.005	1024	[64, 64, 32]	RMSprop	200	50615.87
0.005	128	[6]	Adam	50	50616.09

Table 3: Top rows of hyperparameter search result

Final Evaluation

After hyperparameter searching and fine-tuning, the optimal RMSE obtained was 47062.67. The hyperparameters which yielded this RMSE were a learning rate of 0.005, mini-batch size of 128, patience of 200, and *Adam* optimizer. The architecture of the hidden layers is [32, 32, 16, 8, 4], which represents 5 hidden layers. The L1 loss (absolute mean dollar error) was $8,124.85, and the absolute median dollar error is $ 3,156.93 which is reasonable given the high costs of the homes in the dataset (ranging from $14,999 to $500,001).

References:

[1] "Mean Squared Error and Root Mean Squared Error - Machine Learning with Spark - Second Edition [Book]," *www.oreilly.com*. https://www.oreilly.com/library/view/machine-learning-with/9781785889936/669125cc-ce5c-4507-a28e-065ebfda8f86.xhtml [Accessed: 21-Nov-2022].

[2] Jason Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning," Machine Learning Mastery, Jul. 02, 2017. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed: 21-Nov-2022].

[3] R. Hyndman, doug, hobs, Redoman, D. Marsupial, prashanth, V. Cartas, M. Thoma, user91213, chainD, D. Erez, and sapy, "How to choose the number of hidden layers and nodes in a feedforward neural network?," Stack Exchange, 20-Jul-2010. [Online]. Available: https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw. [Accessed: 21-Nov-2022].