Coursework 2: Artificial Neural Networks

Stefanos Ioannou (si122), Alfredo Musumeci (am322)
Oussama Rakye Abouelksim (or22), Jan Marczak (jjm222)

December 26, 2022

1 Preprocessing

Some values in the dataset might be NaN values and have to be transformed to be able to work with them. The strategy used here has been to replace the NaN values with the mean of the column where they belong. Therefore, all input data can be used accordingly. The column "ocean proximity" from the dataset contains categorical features from the following set of options: "INLAND", "<1H OCEAN", "NEAR BAY", "NEAR OCEAN" and "ISLAND". These values have to be transformed into numerical values so that the neural network can process them. This is done using One Hot Encoder. It takes the column, and for each possible label, i.e. the previous set, it generates a new column for the dataset. Each column will represent each of the entries and will be set as 0 if it does not contain that label, 1 otherwise.

In order to make the attributes with continuous values fit within the same scale, two normalisation approaches have been investigated, namely Standardisation and Min-Max Normalisation. Min-Max scales the attributes to be within [0,1]. Standardisation transforms the distribution to a Standard Normal Distribution. Standardisation was preferred following empirical testing. During training, the model showed a steeper loss during the first few epochs, when using standardised data, compared to min-max normalised data. The noise in the dataset may further explain the latter preference since min-max is sensitive to outliers.

All the transformers were fitted with the training set and the same transformations have been applied to transform the development and test set (without changing the parameters). This is to avoid overfitting. All the inputs of the model undergo transformation to ensure all input distributions are similar.

2 Model Architecture

In general, there is no easy way to determine the optimal network architecture ahead of training and experimenting (also mentioned in the No Free Lunch Theorem). The model was first constructed to overfit the data, to ensure minimum training loss and the model's learning capacity. This meant increasing the number of neurons and the number of layers (depth) of the network. The number of model parameters was then reduced to slightly decrease overfitting, i.e. decrease the validation loss.

At the end of these initial exterminations and given the simplicity of the problem at hand[1], a 4 linear layer sequential network was constructed:

```
NN(
  (layers): Sequential(
    (0): Linear(in_features=13, out_features=100, bias=True)
    (1): Tanh()
    (2): Linear(in_features=100, out_features=70, bias=True)
    (3): ReLU()
    (4): Linear(in_features=70, out_features=20, bias=True)
    (5): ReLU()
    (6): Linear(in_features=20, out_features=1, bias=True)
  )
)
```

[1] We interpreted the simplicity of the data by looking at the number of features. With a relatively small number of features and several features being highly correlated > 0.7

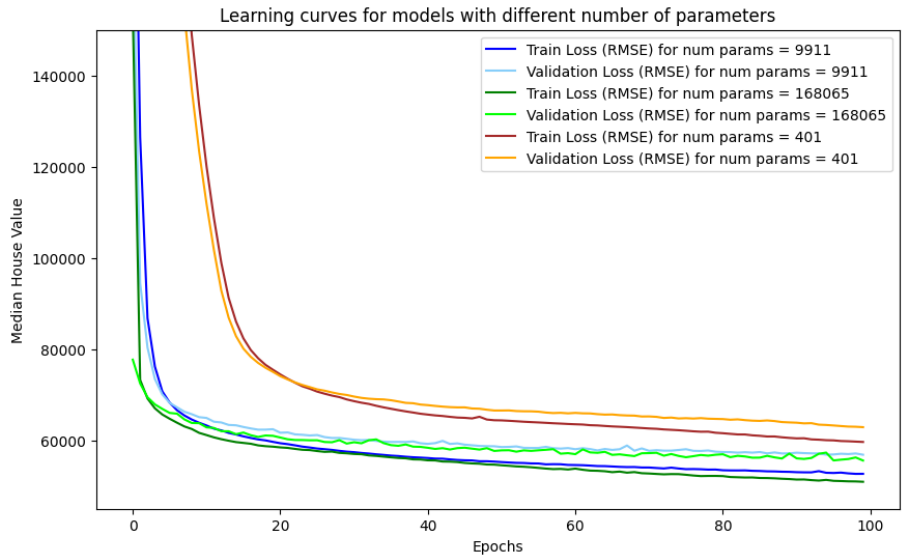

Figure 1: Learning curves for models with different complexity degrees. The number of parameters is mostly determined by the number of neurons in each model's layers. This procedure was performed on a subset of the training and validation data, to avoid memory leakage. This ensures that the model does not undertrain, and does not overly overfit the training set.

As can be seen, the neurons in the hidden layers grow substantially at the beginning and slowly decrease to the desired output shape, i.e. 1. The model has a total of 9911 parameters.

This network was used as our baseline architecture for performing later hyperparameter tuning. To ensure its capability it was also compared to other neural networks. Figure 1 shows the learning curve for models with different number of parameters, mostly determined by the layers' dimensions, i.e. number of neurons. As can be seen, a model with 401 parameters underfits the data and does not offer enough complexity to fully learn the regression task. On the other hand, there is a small difference in performance between our chosen model (9911 parameters) and the most complex model (168065). This suggests that there is a minor advantage in adding additional neurons, with the high cost of slower tuning and training.

2.1 Activation Function

The activation functions used are mainly ReLU, except for the first activation layer which uses Tanh. Although Tahn is not stricly a regression function, it can be used in conjunction with more regression-type functions like ReLU if set in the initial layers of the network. ReLU was chosen because it is widely used in industry and is shown to produce good results for regression tasks. Figure 2 shows the learning curve of the model with all ReLU functions and a model with all ReLUs but the first activation function is Tanh. Each model was tuned separately (similar to the process indicated in section 3). The model with just ReLU activations shows a slower convergence, but a lesser overfitting compared to the model having the first layer as Tanh. The latter configuration was preferred because of the lower training loss, and faster convergence.

Regarding implementation, two additional simple classes *NN* and *CDataset* were created for defining our Neural Network and Tensor Dataset(s) respectively. Our model is created and stored as a variable in the *Regressor* class. *CDataset* with *DataLoader* allows easy configuration of batch size. For training and validation, two separate functions (thus abiding by S.O.L.I.D.) were created, which are then later called in a single loop where the training and validation errors are collected.

3 Training

The **train-validation-test ratio** chosen for this problem is 7:1:2. Since the task is simple, less data was needed to train the model, hence more focus was placed on sufficiently testing the model. A random seed was chosen for the splitting to ensure no test set instances were used in any hyperparameter tuning process.

The **mini-batch** learning approach was preferred over 'Batch' or stochastic, as it offers faster network training and is mostly used in practice. A **batch size** of 16 was preferred since it required

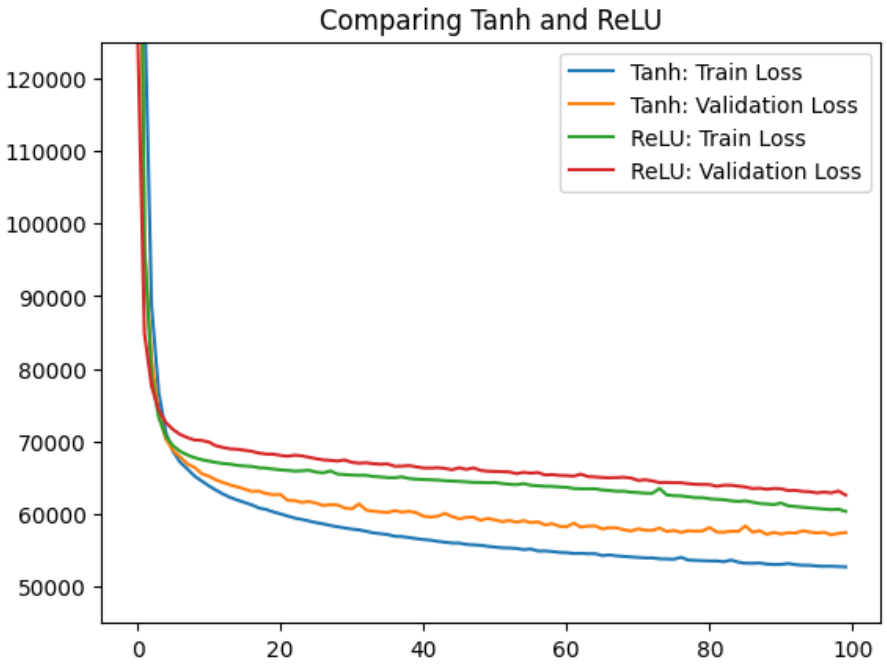

Figure 2: Compare the training and validation loss (Root Mean Squared Error) of model architecture with ReLU as the first activation function, compared to Tanh as the first activation function

less memory than greater batch sizes, i.e. 32, or 64. However, with a batch size of 16 the model runs the risk of calculating an inaccurate estimate of the gradient (smaller sampler, higher noise). In this setting, this does not seem to be the case, as there as there is very little overfitting (see Figure 5 for the learning curve).

For training the model we used Mean Squared Error (MSE) **loss function**, which is typical for regression models. We also considered using Mean Absolute Error (MAE) to possibly reduce MSE's sensitivity to outliers. However, because of the nature of MAE, we speculated that it would train a model only capable of predicting easy instances. We recognize that Huber Loss, an alternative loss function, combines the best properties of both MSE and MAE. However, this was not investigated in this project.

To account for the importance of the learning rate in the gradient descent, an adaptive learning rate optimiser "**Adam**" was used. Adam works differently from the traditional gradient descent approach as it maintains a standalone learning rate for each network weight and adapts it separately. This makes it desirable for neural networks and was used instead of a more conventional learning rate decay approach.

In order to reduce model overfitting, we use **L2 normalisation** and find an appropriate weight decay parameter through RayTune testing. By doing that we avoid the 'exploding gradient' problem by preventing the weights from growing significantly.

4 Hyperparameter Optimisation

Hyperparameter optimisation was performed using RayTune for configuring the search space, and Tensorboard to visualise the results. To choose values for the learning rate (lr) and weight decay (r) parameters samples where drawn from two log-uniform distibution between 1e-4 and 1e-1. For each combination of hyperparameters the model was run for a maximum of 70 epochs. Each model was run on a random subset of the training and validation set to reduce bias in selecting hyperparameters. This approach was preferred over more elaborate methods to reduce hyperparameter selection bias; k-fold cross-validation being a computationally expensive procedure given the large size of the training set. Initially, 40 models were trained with different hyperparameters. 40 more models were trained by taking samples from a reduced search space according to the performance combinations of hyperparameters in the initial 40-model run. ASHA scheduler[2] was used to monitor the tuning process and interrupt the training of a model if it was performing worse than a previous model after 50 epochs. To compare the performance of different model configurations

[2] https://docs.ray.io/en/latest/tune/api_docs/schedulers.html

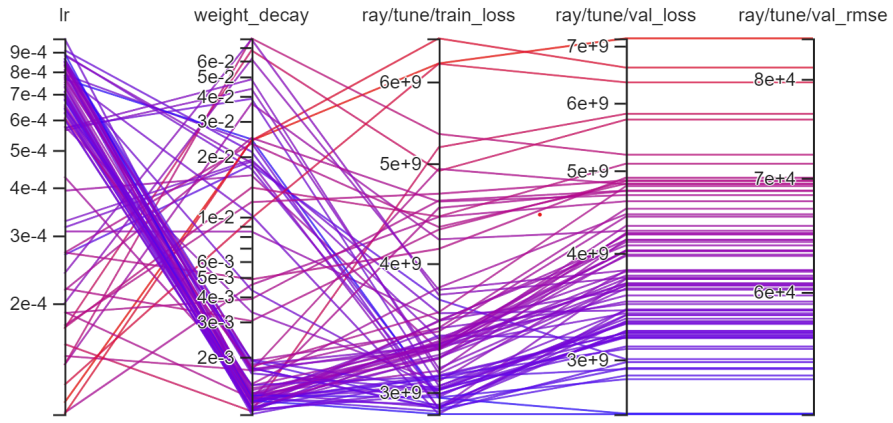

Figure 3: Visual representation of Tensorboard's environment during hyperparameter optimisation. The axis plotted represent the learning rate, the weight decay, i.e. regularisation term, training loss (Mean Squared Error), validation loss (Mean Squared Error), and Root Mean Squared Error on the validation set (from left to right).

Metric	Mean Value	Standard Deviation
Root Mean Squared Error (RMSE)	54616.75	136.81
Mean Squared Error (MSE)	2983008512.0	14941908.74
R^2	0.715	0.006

Table 1: The Mean Squared Error (MSE), R2 score and Root Mean Squared Error (RMSE) averaged over 3 models evaluated on the test set.

the training loss, validation loss and validation Root Mean Squared Error (RMSE) were plotted. The configuration with the lowest difference between training and validation loss and the lowest RMSE was selected. Figure 3 shows a screenshot of the Tensorboard environment and some of the configurations run during the tuning process. A lower learning rate leads to slower training. Increasing the weight decay decreases the gap between the training and the validation loss by increasing training loss. On some occasions, the training loss was higher than the validation loss. This may be because of the high value for regularisation, or just randomness.

The model has then trained with the selected hyperparameters (learning rate= 0.00073378, weight decay= 0.0010347) for 100 epochs. After 100 epochs the model started to severely overfit (see Figure 4), i.e. as the training loss was decreasing the validation loss was increasing, so training was terminated. Figure 5 shows the learning curve of the model trained with the optimal hyperparameters.

5 Evaluation

The evaluation was performed using two metrics, namely RMSE and the coefficient of the determinant (R^2). RMSE was used instead of MSE since the output is in the same units as the target values. A high R^2 value indicates good model fitness. R^2 values range from 0 and 1, although it is also possible to get negative values for very bad predictions. Table 1 shows the evaluation results of the model on the test set. The R^2 score shows that the model has a good fit for the data, and the RSME suggests a good performance with fine stability (indicated by the Standard Deviation). For a more elaborate evaluation of the model's capabilities, approaches to quantify epistemic uncertainty should be implemented.

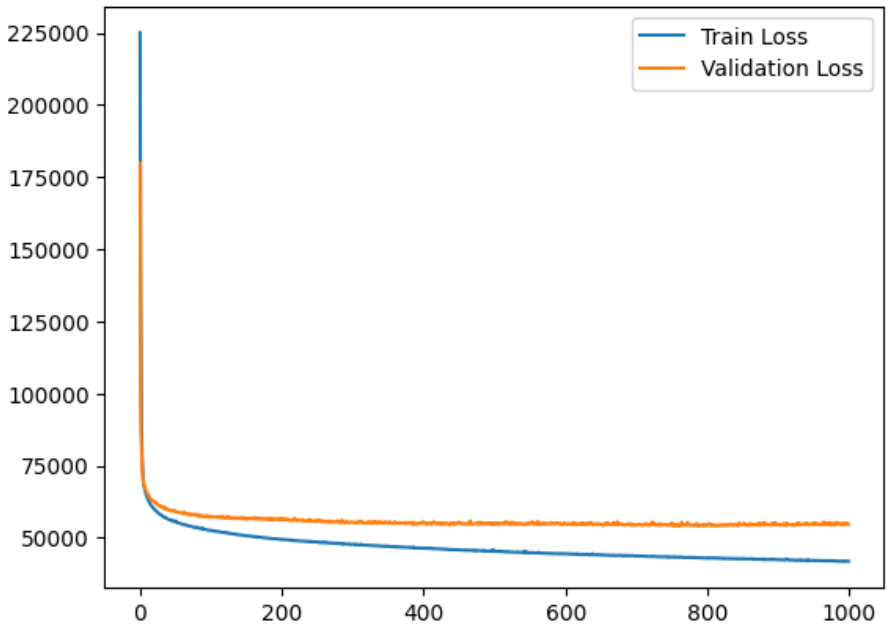

Figure 4: The training loss and validation loss (Root Mean Squared Error) of the model trained with the selected hyperparameters against epoch number (1000).

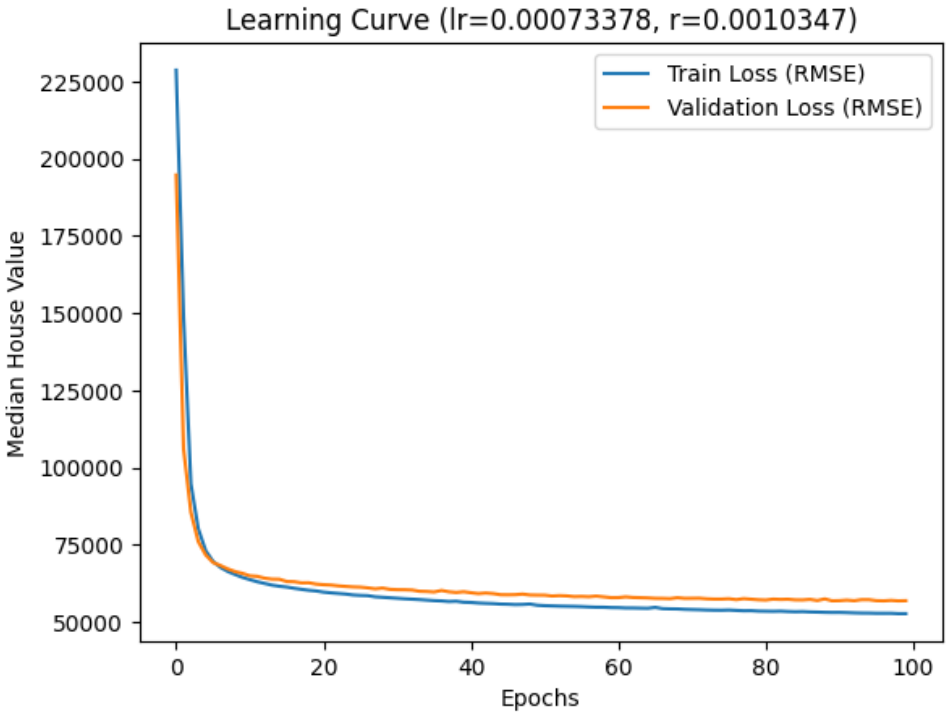

Figure 5: The training loss and validation loss (Root Mean Squared Error) of the model trained with the selected hyperparameters against epoch number.