CW2: Neural Networks

Report on California House Prices Regression

`Imperial College`

February 28, 2025

1 Introduction

Part 2 of 'Coursework 2: Artificial Neural Networks' involved developing, optimising and evaluating a neural network architecture to predict California house prices from the 1990 Census data. Predicting house prices is a widely studied regression task with real-world implications for understanding economic trends and guiding real estate investments. The California House Prices dataset contains records for block groups which include 9 demographic and geographical features. These are: longitude, latitude, median age, total number of rooms, total number of bedrooms, population, number of households, median income and proximity to the ocean. The target variable was the median house value within each block group, making this a standard regression task with continuous labels.

A neural network architecture was implemented using PyTorch and its performance evaluated. Section 2 discusses data preprocessing and the model architecture design, whilst section 3 details the evaluation of our model's predictive performance using Root Mean Squared Error (RMSE) on a held-out test set, as well as the test/train split and loss graphs. Section 4 discusses hyperparameter optimisation, tuning key parameters such as the size of the hidden layer(s), learning rate, and batch size to improve performance. Finally, section 5 summarises the final results and the limitations of our model and approach.

2 Model Architecture and Design Choices

2.1 Data Preprocessing

The data preprocessing was handled in the `_preprocessor` method of the `Regressor` class. This method handled the feature and target variable inputs as Pandas Dataframes. Numeric and categorical columns were identified and the mean and standard deviation of each numerical column were calculated and stored during training, and used to fill missing values. The missing values from the categorical columns were identified and placed in an arbitrary "missing" category.

One-hot encoding of categorical columns was necessary, as the neural network required numerical inputs. This was done by identifying the categorical column(s) (in this case, the `ocean_proximity` feature), transforming each category into its own binary column, and finally, setting 1 for the presence of a category and 0 for the absence of the category. This essentially transformed any categorical columns into an orthonormal basis set. By preserving the column ordering, the one-hot encoding process was applied consistently across both the training and test data.

During training, the target house price variables were normalised and stored, and the same normalisation procedure was subsequently applied to both the validation and test data. A Z-score normalisation was applied to the numeric columns. Min-max normalisation was also considered, since it preserved the distribution shape of the data and was less affected by distribution skew in features like 'median_income'. However, Z-score normalisation was determined most suitable as it captured outliers well. This was essential given the large spread of data values in several of the feature columns (see Table 1). Z-score normalisation also ensured that the large-scale data did not dominate the smaller-range data. This ensured that 'total_rooms', for example, ranging in the thousands was not at risk of dominating smaller-range features such as 'latitude', 'longitude' and 'housing_median_age'. Furthermore, when both versions of the model were run, the Z-score normalisation produced an RMSE of $\$59,700$, significantly lower than the RMSE that resulted from the min-max normalised model, $\$61,100$. Finally, the processed data was converted to PyTorch tensors, ready to be handled by the neural network.

Column	**Mean**	**Standard Deviation**	**Coefficient of Variation (%)**
longitude	-120	2.01	1.7
latitude	35.6	2.14	6.0
housing_median_age	28.6	12.6	44.1
total_rooms	2640	2210	83.7
total_bedrooms	539	427	79.2
population	1420	1150	81.0
households	499	386	77.3
median_income	3.87	1.90	49.1

Table 1: Mean, standard deviation, and coefficient of variation for each numeric column in the California House Prices dataset.

2.2 Network Structure

The constructor method, `_init_`, initialised the neural network model by first applying the `_preprocessor` method to the input training data.

The neural network architecture began with an input layer sized dynamically to match the number of features from preprocessing. The number of epochs, learning rate and batch size were all implemented as configurable parameters, which could be optimised later during hyperparameter tuning. Following the input layer were four configurable hidden layers, each with a default size of 32. The first and second hidden layers applied a ReLU activation function, the third a Sigmoid activation, and the fourth returned to a ReLU activation. The output layer consisted of a single neuron that predicted the median house value, making the network suitable for regression tasks. The model was optimised with the Adam optimiser and MSE was used as the loss criterion, penalising larger errors. Further, under the assumption of normally distributed errors, minimising the MSE is equivalent to performing maximum likelihood estimation, making MSE an optimal criterion.

This design balanced simplicity and performance, capturing complex patterns while maintaining computational efficiency. This architecture was empirically determined to produce a model with high accuracy relative to other similar architectures, using different numbers of hidden layers and different activation functions (see section 3). The ReLU activation prevented vanishing gradients, ensuring stable training, while the Adam optimiser dynamically adapted the learning rate for improved convergence.

The training method `fit` followed a structured approach to optimise the model over a number of epochs. First, the input and target values were preprocessed and converted into PyTorch tensors. A DataLoader was then used to facilitate mini-batch gradient descent with shuffling, improving computational efficiency and reducing variance in gradient updates. For each epoch, the training loop iterated through mini-batches, performing a forward pass and predicting values based on the input data. The loss was then computed using the MSE loss function, which measured the difference between predicted and actual house prices. A backward pass was performed using `loss.backward()`, which computed the gradients of loss with respect to the model's parameters. Finally, the Adam optimiser updated the weights using these computed gradients, adjusting the network parameters to minimise the loss.

To improve learning, batch processing and data shuffling were incorporated to prevent order-based biases and enhance model generalisation. Throughout training, the loss was recorded and monitored across each epoch, and a visualisation of the loss curve was generated to assess the training progress and convergence behaviour (see Fig. 1).

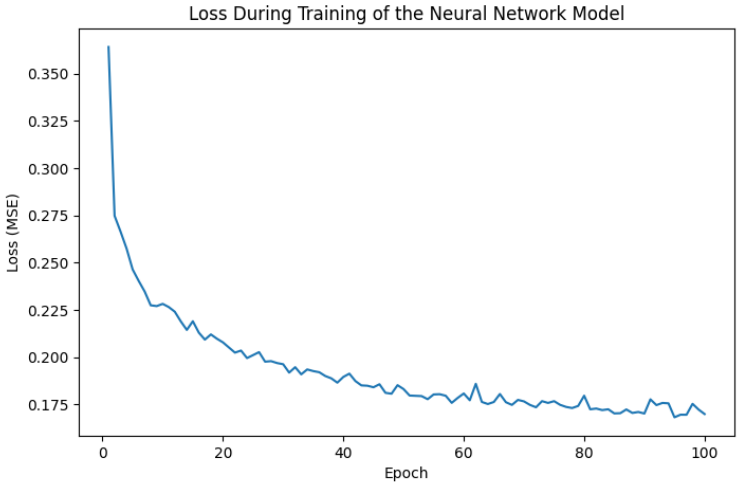

Figure 1: A graph displaying the loss during training over each epoch. (Parameters: number of epochs = 100; hidden layer size = 32; learning rate = 0.01; batch size = 64).

3 Evaluation Setup

The `predict` method first applied the same preprocessing steps used during training, thereby maintaining identical data transformations between training and evaluation. In order to minimise use of inefficient Python loops and maximise efficiency, PyTorch's tensor operations were used. The model was set to evaluation mode, and predictions were computed using `torch.no_grad()`. The predictions were then rescaled back from their normalised values to the original target range using the stored mean and standard deviation values obtained from the training data. This ensured that the output of the model was directly comparable to the true house values.

The `score` method built upon the predict method by evaluating the model's performance on a given test dataset. It computed the Mean Squared Error (MSE) using scikit-learn's `mean_squared_error` function. As mentioned above (see section 2), one benefit of using MSE was the fact that it penalised larger errors more heavily. This was useful for predicting house prices, as large errors could disproportionately impact model performance. However, this also meant that extreme outliers had a stronger influence on the training process, which could be a

drawback in datasets with highly skewed price distributions. The RMSE was then computed, which made interpretation easier by providing an error measure in the same units as the target variable.

The full dataset was first split into training, validation and training datasets using an 70:20:10 split. The validation training set was used by the predict and score functions to roughly select a network structure that performed well. After selecting the network structure, the validation dataset was re-combined with the training dataset in order to train the model on a larger dataset.

Initially, only a single hidden layer was used, which resulted in an RMSE of $\$59,700$ on the validation data set. Whilst this meant that the model's predictions deviated by $\approx 28.8\%$ from the true house prices, further analysis revealed that this magnitude of error was tolerable. Considering that the dataset had an mean house price of $\$207,300$ with a standard deviation of $\$115,700$, the RMSE ($\approx 0.52\sigma$) was reasonable given the substantial variability inherent in the data. Thus, despite the seemingly high percentage error, the model's performance was robust, providing a solid baseline for future enhancements.

However, it was found that the model's accuracy was nevertheless improved with further adjustments to the architecture, as empirical tests were run with differing numbers of hidden layers and combinations of activation functions. Increasing the number of layers to three (ReLU, ReLU, ReLu) reduced the RMSE by 10.7% compared to the single-layer ReLU model. A more optimal configuration was found to be the four-layer network (ReLU, ReLU, Sigmoid, ReLU), which further reduced the error by 12.5% relative to the single-layer model (see Table 2). This is because using only ReLU can cause overfitting due to its unbounded nature, but adding a Sigmoid in the third layer stabilises learning by normalising outputs. This prevents large values from propagating and enhances nonlinearity, improving the capture of complex interactions.

When evaluated on the test dataset, this optimised model produced an RMSE of $\$52,789$, only slightly higher than the validation RMSE. This suggested good generalisation with some minor overfitting to the training data. This final RMSE was 25.5% of the mean house price and approximately 0.46σ. This indicated that the model captured meaningful trends on the data, as the error was significantly smaller than the actual variation in house prices.

Hidden Layers	Configuration	RMSE ($)	% Reduced from single layer
3 Layers	ReLU, Sigmoid, ReLU	54,243	9.1%
	ReLU, ReLU, ReLU	53,317	10.7%
4 Layers	ReLU, ReLU, ReLU, ReLU	53,239	10.8%
	ReLU, ReLU, Sigmoid, ReLU	52,255	12.5%

Table 2: RMSE Comparison for Different Hidden Layer Configurations on the Validation Training Set

4 Hyperparameter Optimisation

4.1 Search Strategy

A hyperparameter search was conducted to determine the best configuration for the neural network. The `perform_hyperparameter_search()` function systematically iterated through a

Cartesian product of hyperparameter combinations for preset ranges, before selecting the best-performing model on the validation split. The model was trained on 80% of the training dataset (which was 90% of the full dataset), with 20% ofreserved for validation. This hyperparameter search varied the learning rate (`0.0001, 0.001, 0.01`), hidden layer size (`16, 32, 64`), batch size (`16, 32, 64`), and number of epochs (`50, 100, 200`). A total of 81 hyperparameter combinations were tested and the optimal configuration was selected by minimising MSE.

4.2 Findings

The best combination of hyperparameters was found to be a size of 32 for the hidden layers, a learning rate of 0.001, a batch size of 16, with the convergence occuring at around 100 epochs. This model yielded a RMSE of $\$49,904$ on the validation split (the lowest of all the combinations), indicating that it generalised best on unseen data. This optimised model produced a RMSE of $\$50,791$ against the test split. This represented an average prediction error of about 24.5% relative to the mean house price ($\approx 0.44\sigma$).

Models with hidden layer sizes of 32 and 64 generally outperformed those with 16, demonstrating that deeper networks capture more complex patterns and lead to better generalisation. However, very large networks (i.e., size 64 neurons) sometimes resulted in higher RMSE, likely due to overfitting. This result highlights the importance of finding an optimal balance where the model is complex enough to learn from the data but not excessively large, which could introduce training difficulties or unnecessary computational costs.

A learning rate of 0.001 consistently performed well across different configurations, achieving lower validation RMSE values. Lower learning rates (0.0001) resulted in higher RMSE values, suggesting slower convergence and possible underfitting. Higher learning rates (0.01) led to increased RMSE, indicating instability and potential overfitting.

Larger batch sizes than 16 led to worse performance, possibly due to reduced gradient update frequency, making optimisation less effective.

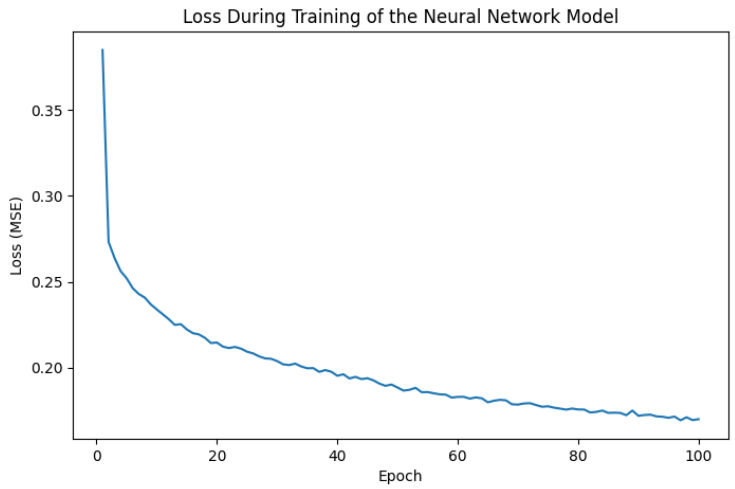

Figure 2: Graph displaying the loss during training for the optimised data. (Parameters: number of epochs = 100; hidden layer size = 32; learning rate = 0.001; batch size = 16).