Machine Learning Coursework 2 – Neural Networks

Ryan Ward, Panagiotis Doumas, Hamed Mohammed

1 Introduction

This coursework's aim is to create and train a neural network to predict the price of houses in California using the California House Prices Dataset. The dataset contains ten attributes covering all the block groups in California from the 1990 Census. We implemented a multi-layer neural network in Python using functions from the Scikit-Learn, PyTorch, NumPy and Pandas libraries. We decided not to use our neural network mini library from part 1 because to test our regression model for part 2, we need to use the datatype tensor and changing part 1 to accommodate would result in a negative feedback loop, causing our implementation for part 1 to be incorrect as per the specification.

The network consists of an input layer, one hidden layer and an output layer. The input and hidden layers use a linear activation function applying the identity operator, whereas the output layer uses ReLU activation; this is because ReLU is simpler and tends to converge faster when training the network.

2 Implement an architecture for regression

2.1 Preprocessor Method

The preprocessor method pre-processes the input and output of our model. Using sklearn SimpleImputer, we handle the missing values in the dataset by setting them to the mean value in each column, except for the ocean_proximity textual values. ocean_proximity values are handled separately by performing one-hot encoding on the column using sklearn LabelBinarizer. The possible values were hard coded to the five options in the data, ["<1H OCEAN", "INLAND", "ISLAND", "NEAR BAY", "NEAR OCEAN"].

We then normalise the input and output values to a scale of 0 to 1 by performing min-max standardisation using the sklearn MinMaxScaler. Followed by converting the datasets to tensors and returning them. Normalising the numerical values prevents larger values from disproportionately affecting the data, resulting in faster gradient descent convergence and improved learning.

2.2 Constructor Method

The constructor initialises the classes needed for pre-processing. To facilitate hyperparameter tuning, further parameters were added to the constructor: learning rate, batch size and neuron arrangement. Parameters are set with a default value to avoid errors when testing.

The architecture of the network is given by a list of integers, each representing the number of neurons for a given layer; [4,4,4] represents 3 layers each with 4 neurons. We implemented the network using sequential linear layers with identity activation functions accompanied by a single neuron final layer with a ReLU activation function. We use the PyTorch Sequential container to chain the outputs of each layer with the next layer.

We implemented a function named _init_weights which initialises the weighting of each linear layer using the xavier uniform method and the biases to 0. The xavier uniform method initialises the weights such that the variance of the activations is the same across every layer.

ReLU was the optimal choice as it guarantees a real positive result, which is desired considering we are working with attributes such as total _rooms.

2.3 Model-training Method

We implemented the fit method by first calling the preprocessor function to normalise the input and output values. Then using the Pytorch library, we make use of the Adam optimisation algorithm to update network weights iterative based in training data. We decided to select the Adam optimiser over other methods, such as Adagrad because the Adam optimiser had the smallest training error and it converged towards the minima at a better rate than other similar algorithms when we ran them.

Using a for loop we loop over all the epochs, shuffling the training data and splitting it into multiple batches. After setting the gradients of all model parameters to 0, the batches are passed into our neural network one at a time, propagating forwards and backwards. As the function loops through the batches, it calculates the root mean square error and appends it to the array batch_mse_loss.

Within the fit function, it can take elective parameters, val_x and val_y; by default they are set to None. We use this validation data set to measure the performance of our model, and then compare it to the best-performing model saving the superior one. If our model does not improve after a certain number of epochs, which we specify with self.prompt_stop, the function returns the current best model.

3 Set up Model Evaluation

3.1 Prediction Method

We start by calling the preprocessor function to normalise the input values again. This is then forward passed through the network to produce the predicted output values; they are then separated from the graph by returning a new tensor which doesn't require a gradient and reverted using the inverse transformation so that they are comparable with the target output values.

3.2 Evaluation Method

We implemented the score method by first calling the preprocessor function to normalise the input values again. We use root mean square error as the evaluation metric and it is returned by the function. Due to the squaring in RMSE, it greatly punishes larger errors which guarantee they are visible in the final metric**.**

4 Perform HyperParameter Tuning

The function starts by shuffling and splitting the data set into train, test, and validation sets. We use GridSearchCV, which takes in the regressor and a set of parameters to be tested, to perform an exhaustive search over the parameters. The varied hyperparameters were the batch size, learning rate, neural network architecture, and the number of epochs.

This Sklearn method divides the domain of the hyperparameters into a discrete grid and then tries every combination of values of this grid, calculating the root mean squared error and tuning the parameters using cross-validation. It takes the best-performing combination of hyperparameters for our model and stores them, then calls the previously defined fit function to get the optimal model with these parameters. It returns the best parameters.