Imperial College London

COURSEWORK 2: NEURAL NETWORKS

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Intro to Machine Learning

Authors: Aidan Holmes, Meg Howard, Hugo Frelin, Igor Sadalski

Date: November 23, 2023

1 Intro

Deep neural networks are a powerful tool for regression tasks, as they can be proven to be universal function approximators. This means that they in theory can approximate any underlying distribution from which the data is sampled, but they have the drawback that if not designed correctly can overfit the data observed. The design of the model is thus crucial for performance, but unfortunately, it is rarely straightforward. We have therefore utilized multiple regularisation techniques, as well as searched the hyperparameter space as exhaustively as possible within our computational and time constraints to optimize our model.

2 Model

2.1 Architecture

The model is built using PyTorch's nn.Module. It is designed to dynamically construct a neural network based on specified parameters given when called. This flexible architecture allows for customized input and output sizes, as well as varying numbers and sizes of hidden layers, augmented by dropout for regularization.

The network's layers are stored in an nn.ModuleList, a PyTorch container. The construction starts by creating the input layer, which connects the input size to the first hidden layer size using a linear transformation. This is immediately followed by a ReLU activation function and a dropout layer for regularization. The class then iteratively constructs additional hidden layers (if any are specified) in a similar fashion: each hidden layer is represented by a linear transformation connected to the next layer size, paired with a ReLU activation and dropout. Finally, an output layer is added, connecting the last hidden layer to the output size without an activation function. The forward method defines the data flow through this network as it sequentially applies each layer in the model to the input data x, transforming it step-by-step until the final output is produced. This method is a critical component of the model, as it specifies how the input data is processed through the layers to produce the output.

2.2 Preprocessing

The preprocessor method prepares input data for the neural network, catering to both training and testing scenarios. It is designed to handle datasets with a mix of numerical and categorical features. The method starts by making a copy of the input data to ensure that the original dataset remains unaltered and avoid the warning exceptions typically thrown by the Pandas library in this scenario. This step is vital for data integrity, especially when dealing with data structures like Pandas DataFrames and to avoid warning exceptions. Missing values in the data are addressed by filling them with zeros, ensuring there are no gaps that could lead to errors during model

training or predictions.

In the case of training (when the training parameter is True), the method takes several additional steps to prepare the data. It first deals with categorical data, specifically the last column in the input DataFrame, which is the categorical ocean_proximity column. This column is one-hot encoded using LabelBinarizer, a technique for converting categorical data into a format that neural networks can process more effectively. The method then normalizes the numerical features in the dataset using a StandardScaler. The scaler is fitted based on the training values, before then fitting the training values to be normalized. This normalization transforms the data to have a mean of zero and a standard deviation of one, a form that is often more suitable for training neural networks. The method also includes a provision for normalizing the target data y, if provided, by calculating and applying its mean and standard deviation.

For testing purposes (when training is False), the method assumes that the encoders and scalers have been previously fitted during the training phase. It applies these pre-fitted transformations to the new data. This ensures consistency in how data is preprocessed across training and inference, a crucial aspect for the model's performance. The method is designed to return the processed input data x and target data y (if provided) as PyTorch tensors, making them compatible for use with the PyTorch-based neural network model. This careful and systematic approach to preprocessing ensures that the data fed into the neural network is in the optimal format, enhancing the model's ability to learn from the data and make accurate predictions.

2.3 Fitting

The code begins by preprocessing the input data (X, Y) using the preprocessor method described above, which prepares the data for training. It then creates a TensorDataset and a DataLoader from the processed data using the Torch library, facilitating batch processing and shuffling of the training data. It is important to shuffle the data before training, as this reduces the chance of any correlation in the data due to the initial ordering. For this dataset, this likely would not be a huge issue, but it is still good practice. The neural network model is instantiated with specific parameters like input size, output size, hidden layer sizes, and dropout probability. These were initially chosen randomly, before being optimized in the hyperparameter search

The training process involves multiple epochs, where an epoch represents a full pass through the entire dataset. Within each epoch, the code iterates over batches of data, performing forward and backward passes to compute the loss using Mean Squared Error (MSE) and update the model's weights through stochastic gradient descent (SGD). The code also implements early stopping, a technique to prevent overfitting. Early stopping monitors the training loss and stops the training process if the loss doesn't improve by a defined delta margin for a certain number of patience epochs. Additionally, the code prints the loss at each epoch and triggers early stopping if the

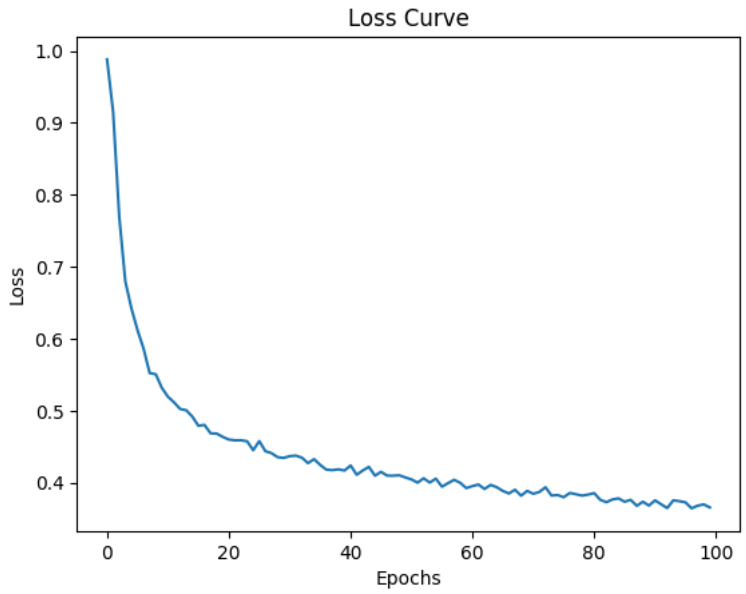

Figure 1: Loss curve for a model with unoptimized hyperparameters.

condition is met, ending the training process to avoid overfitting and save computational resources. The loss was plotted as a function of epochs, for a generic model configuration, and can be seen above.

3 Evaluation Setup

After training our regressor model using the above 'fit' function of the Regressor class, its performance is finally ready to be evaluated. This is done by first using the 'predict' function, to make predictions on new data, and then scoring these predictions using the 'score' function.

3.1 Predicting

First, predictions are made on the input data x. The predict function assesses whether it is being called from within the score method; when it is, the input data fed into predict is already preprocessed, and so is fed straight into our neural network model described in the architecture section 2.1. If on the other hand, the predict function is called directly, it's assumed the data hasn't been preprocessed, and the input array (of shape batch size, input size) is thus put through our preprocessing method before being fed into our model.

After this, the model provides some output predictions that then have to be post-processed, essentially reversing/denormalizing the scaling done on our input features (to make them a more similar scale). This is because our model, trained on normalized data, provides prediction outputs on this transformed scale. Converting these outputs back to the original scale allows us to interpret them in the context

of the original data (i.e. actual house prices). Finally, this function returns these predicted output values from the given input.

3.2 Scoring

The score method then evaluates the performance of the model's outputs using the given input arrays of x, and the corresponding output labels, y. This raw input data is preprocessed before calling the predict method on the input data x, which computes the corresponding y value predictions.

After this, the root mean square error evaluates the performance of these predicted values compared to the actual y values input to the method. We chose to use the RMSE performance metric specifically since this indicates how far the predicted values fall from the true input values using Euclidean distance, which is ideal for a regression problem dealing with continuous target variables but also returns the average error in the same units as the target variable. This enables an understanding of the model's performance in an interpretable way, with the scale of errors being the same as the problem domain - i.e. an RMSE of $10,000 means that on average the predictions are off by approximately $10,000.

4 Hyperparameter Search

To perform a hyperparameter search we used grid search over five main hyperparameters: learning rate, number of epochs, hidden layer sizes, batch size, and dropout. Their ranges, selected values, and justification for using where presented in Table 1. The final optimized solution yields a loss of around half of what was observed at the convergence of Figure 1.

For each possible combination of hyperparameters we first initialize the regressor, then we train it. Once trained we evaluated it using its build-in score function and updated our best parameters if the model was better, returning in the end the best combination.

5 Final Evaluation

The final optimized model, whose loss curve during training is shown in Figure 2, has an RMSE of 5.4×10^4 on the testing data. As evident from Figure 2 it converges to a much lower loss than the suboptimal parameters, indicative of better performance. Perhaps the use of an even more extensive grid search could have allowed for the creation of an even more powerful model, but as it stands this architecture successfully meets the performance criteria on LabTS.

Parameter	Options	Selected Value	Justification
Learning Rate	[0.001, 0.01, 0.1]	0.1	Lower learning rates seem to have performed worse, with the model perhaps getting caught in local optima.
Epochs	[30, 50, 100]	100	Sufficient epochs to ensure convergence.
Hidden Layer Sizes	[16, 16], [32, 32], [64, 64], [32, 64, 32]	[64, 64]	Depth and width of layers impact model complexity and representational capacity. Too complex of a model can also cause overfitting.
Batch Size	[128, 256]	256	A balance between computational efficiency and model convergence.
Dropout Probability	[0.1, 0.3, 0.5]	0.1	Helps prevent overfitting by randomly dropping neurons during training.

Table 1: Selected Hyperparameters and Justification

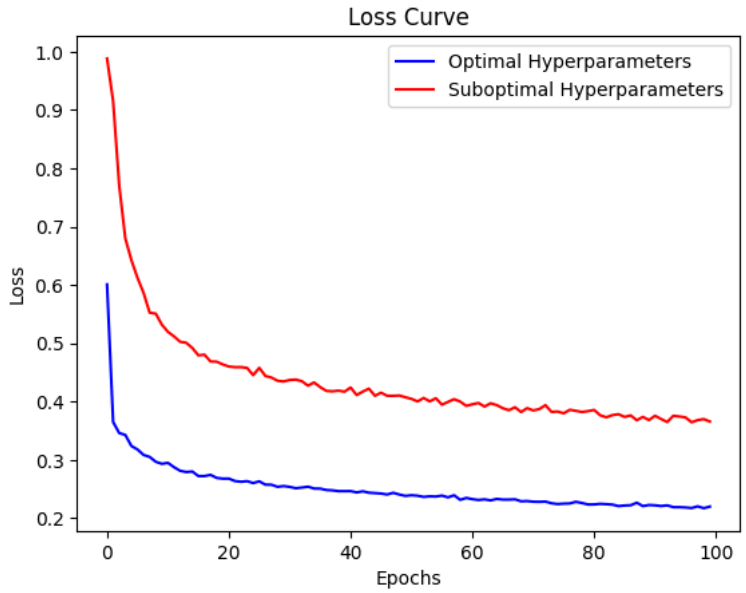

Figure 2: Loss curve for a model with best hyperparameters.