Introduction to ML - Artificial Neural Networks

Yilun Cheng, Pengfei Xiao, Qiancheng Fan

November 24, 2023

1 Regressor Description

1.1 Neural network structure

The core neural network is implemented based on the torch.nn.Module, based on linear layers and activation functions. While the input layer and the output layer is fixed according to the given dataset, the hidden part of the model is implemented with configurable hidden layer number and the number of neurons on each layer. The default structure of the network contains 2 hidden layers with 32 nodes each. We decided to put 32 neurons in each hidden layer to ensure the model can capture enough number of hidden features from the inputs, such as the probability of having an outdoor garden from the longitude, the latitude and the population, since areas with more sunshine and less population may tend to have gardens outdoor. The number of 2 hidden layers is decided for the model to be able to capture different levels of abstraction or features, while not making the network too complex.

Between every two layers an activation layer function is applied, with default type of ReLU considering its simplicity and efficiency in regression tasks. Sigmoid and tanh, on the other hand, is more suitable for solving binary or bipolar classification tasks.

1.2 Opetimizer and Loss function

Except for containing a network described above, the Regressor class also holds other functional instances for the regression task, including the optimizer and criterion. For the given task, we choose a Stochastic Gradient Descent (SGD) optimization algorithm as the optimizer since it is more computationally efficient compared to batch gradient descent, and requires less training time for large datasets. Also, since SGD updates the parameters based on mini-batches of the data, it requires less memory than other popular algorithms like Adam. For the training loss criterion, we choose the popular Mean Square Error (MSELoss) to represent the distance from the predicted median house values to the actual values.

1.3 Data pre-process

In the data pre-process phase, we introduced two helper methods from Sklearn: LabelBinarizer and RobustScaler. The LabelBinarizer is imported to apply the one-hot encoding algorithm to the textual values in the given dataset. While there are 5 possible values for the 'ocean_proximity' column, 5 columns are concatenated to the dataset replacing the original 'ocean_proximity' column, where a value of 1 indicates that the instance have a proximity value of this column, and 0 indicates the other way.

The RobustScaler is used for data normalization. In the pre-processor, we perform normalization on the feature values and the target values separately, to improve convergence and the gradient descent stability. Normalization also helps to uniform the impact of each feature both on training and loss calculation.

As for the missing data in the dataset, we fill out the missing slots with the average value of the corresponding column.

1.4 Model training

For the fit() function in model training, we used utility functions from PyTorch like TensorDataset and DataLoader. These helps us divide input training data into small batches and perform step-wise training. For each batch, we perform the forward-pass and the backward-pass accordingly.

In the forward-pass, the regressor applies the model's parameters to the input features to compute the predicted output, and use the criterion to get the loss value. In backward-pass, the model computes the gradients of the loss with respect to the model parameters, and propagate the gradients backward through the network to use the chosen optimization algorithm (which is SGD in the implemented model) to update the parameters to minimize the loss.

Before the optimization starts, we set a norm clip to the gradient to avoid gradient explosion, making the model more stable against extreme value. For certain training epochs, we report the average training loss across all batches in the epoch, to keep track of the change of the training loss. In the evaluation phase later, however, the loss function is applied only once to the whole test dataset to evaluate the performance of the model.

2 Evaluation setup

2.1 Data set segmentation

Before doing the model training, we first split the dataset into training set and testing set randomly with split ratio 8:2 by using sklearn's train_test_split method. And we keep the random sate seed as 1 to control for the variables of the experiment. The training process only take place on the training set, and we use testing set to check the performance of final trained model.

2.2 Performance evaluation

The performance of a model is assessed by the loss of the model. The smaller the loss, the better the performance. When we calculate the loss, we first normalise the label value on both estimated label and goal label and calculate the Mean Square Error on the normalized data to get the model performance. Also, to evaluate the model performance we use the test set rather than the training set.

3 Hyper parameter picking method and result analysis

3.1 Hyper parameter type

In our experiment, we have 5 hyper parameters, learning rate, epoch number, batch number, hidden layer number and the number of neural on each layer. For each variable, we have a list of values to chose from, which is shown in table 1.

Type	Candidate values
learning rate	0.5, $1e-2$, $1e-3$, $1e-5$, $1e-7$
epoch number	50, 100, 500, 1000,4000
batch number	8,32, 256, 512
neural number	16, 32, 64,256
hidden layer number	1, 2, 3

Table 1: Hyper parameter candidate table

3.2 Hyper parameter picking method

We tried different values for each parameter and generate the final choice of parameter values based on the result of RegressorHyperParameterSearch function. In this function, what we do is loop through all possible combinations of hyper parameters from a given list of choice of value. For each combination, we generate a model based on it and test its performance, each time we find a better combination,

we record its parameter value as best choice. And at the end of this function, we return the recorded parameter value that have the highest performance.

3.3 Analysis of the effect of different hyperparameters on the results

When analysing for each hyperparameter, we control the other hyperparameters at the same default value and only vary the analysis object to analyse the results. For the choice of default value, we use some most common value, which is show in table 2. Then, let's talk about each parameter seperately.

Type	Default value
learning rate	$1e-3$
epoch number	500
batch number	256
neural number	64
hidden layer number	2

Table 2: Hyper parameter default value

3.3.1 Learning rate

From the graph 1 we can see for learning rate, when the learning rate is very small, the loss becomes very high, that possibly because when the learning rate is very small, each time the weight matrix get changed, the variation is too small, and the number of epoch we have is limited, when learning rate too small, we do not have enough epoch to change the model, which makes the model underfit. But a very large learning rate is also not a good choice, it makes the model unstable and easily get influenced by noise. So based on the experiment result, the learning rate around $1e-3$ and $1e-2$ is reasonable and suitable.

Figure 1: error rate vs learning rate.

3.3.2 Epoch number

From the graph 2 we can see for epoch number, the larger the epoch is, better the performance is. Which make sense because more training epoch makes the model fit the training set better and thus get better accuracy. But increase of number of epoch siginificantly increase the total training time, and when the epoch is large, boosting the epoch has a diminishing effect, which makes boosting the epoch less and less time cost-effective. And predictably, when the epoch is too large, the model will overfit, which makes the performance on testing set worse, so keep the epoch number at a reasonable range is more suitable.

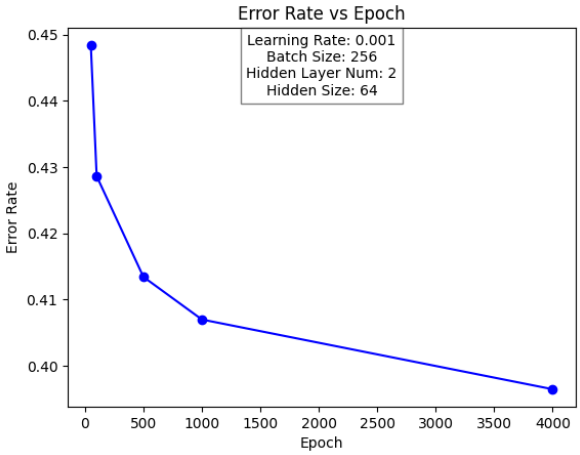

Figure 2: error rate vs epoch number.

3.3.3 Batch number

From the figure 3, it is observed that, in terms of batch size, smaller values result in superior performance. This is rational, as smaller batch sizes, despite requiring longer training times, exhibit better overall performance. Larger batch sizes reduce the stochasticity of gradient descent, leading to a decrease in the generalization capability of neural networks, ultimately resulting in a notable increase in the obtained error rate. Additionally, larger batch sizes may give rise to memory/GPU overflow during the training process, particularly evident on machines with lower hardware capabilities.

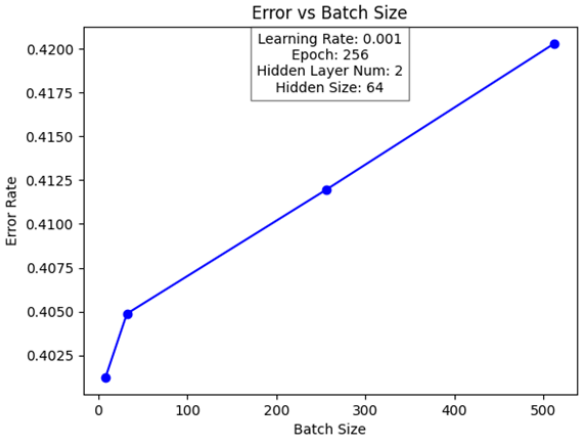

Figure 3: error rate vs Batch number.

3.3.4 Neural number

From the figure 4, it can be observed that as the number of hidden neurons increases, the error rate gradually decreases. However, when the number of neurons in a single-layer is raised to a certain extent, the impact of increasing the neuron count on performance gradually reaches a bottleneck. This might be attributed to the relatively low complexity of the problem, where the input dimension is limited, and consequently, altering the number of hidden neurons has a relatively limited impact on the performance of the neural network.

3.3.5 Hidden layer number

From the figure 5, it is evident that as the number of hidden layers in the neural network gradually increases, the error rate decreases. This aligns with expectations, as increasing the complexity of the

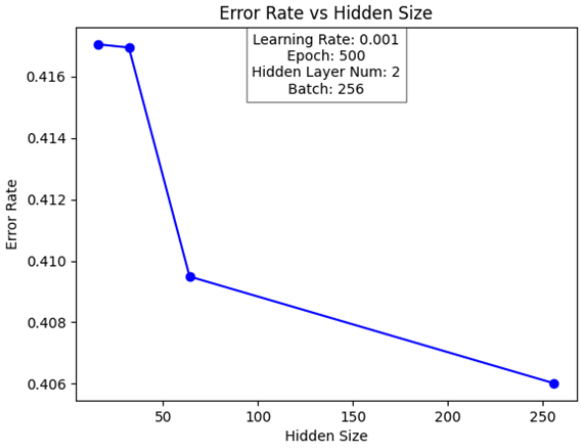

Figure 4: error rate vs Neural number.

neural network typically enhances its performance on specific tasks. However, the increase in the number of layers also amplifies the complexity of the training process. Moreover, merely augmenting the number of layers in the neural network may not significantly impact its performance under certain conditions.

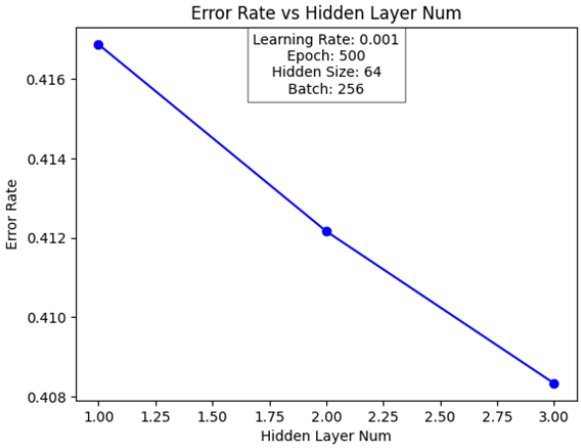

Figure 5: error rate vs Hidden layer number.

4 Final evaluation of our best model

In our experiment, the best model we get has an error rate 0.3167. The hyper parameters it uses are: learning rate 0.01,epoch number 500, batch size 32, hidden layer number 3, neural number 64. And in the best model evaluation experiment, we did not include candidate values of epoch number 1000 and 4000, and neural number 256, because it will be too long to evaluate the best model if we include them. But the result we get is still very representative. The parameters obtained are also in good agreement with the results we analysed earlier.