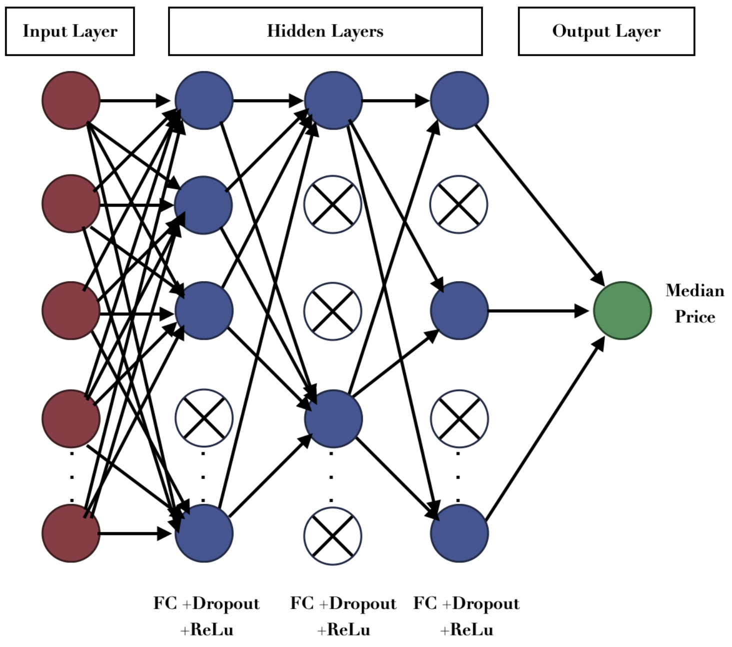
# Introduction to Machine Learning: Coursework 2
# Artificial Neural Networks: Part 2

Hanna Behnke, Daria Galkina, Doruk Taneli, Alexander Reichenbach

December 1, 2020



## Contents

## 1 Architecture of Regressor

The summary of the regressor workflow can be found in the Figure 1.

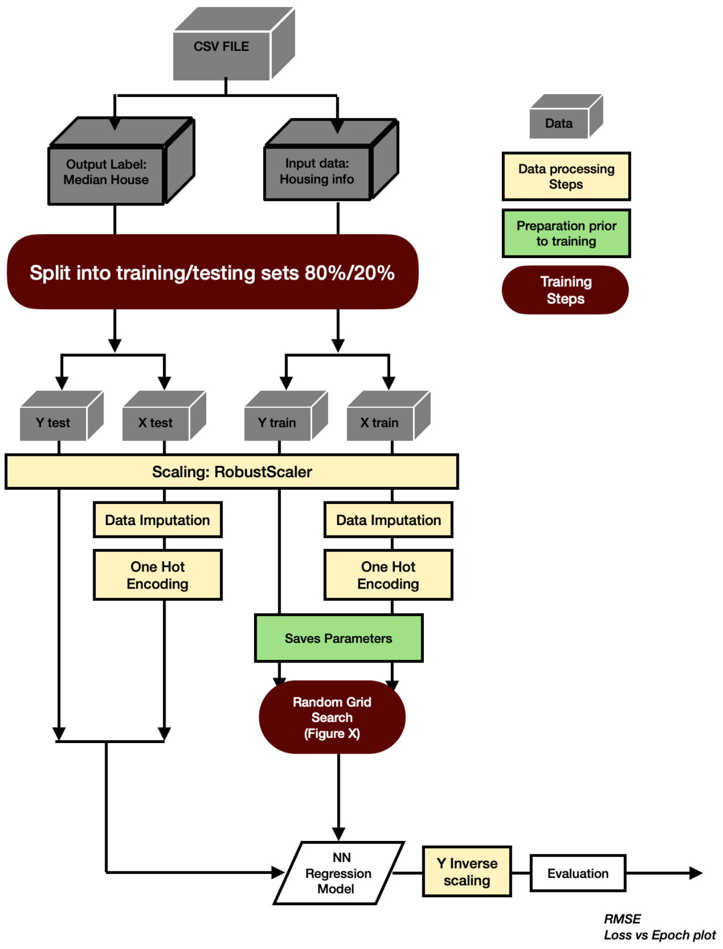


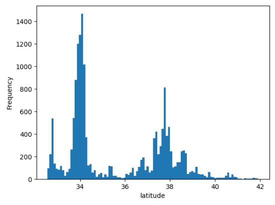
Figure 1: Summary of the Regressor workflow

### 1.1 Constructor

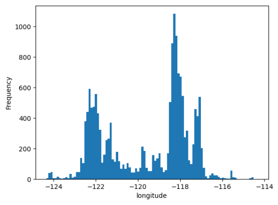
The constructor takes the training dataset and the regressor hyperparameters as input parameters. The hyperparameters for the final model are provided from the result of RegressorHyperParameterSearch function. However, they are initialized with some arbitrary default values for LabTS tests. The constructor initializes the parameters that will be fit and/or applied to data in the preprocessor, and calls preprocessor on the training dataset. It then initializes: the hyperparameters, some helper attributes as global parameters, and the neural network as net.

### 1.2 Data Preprocessesing

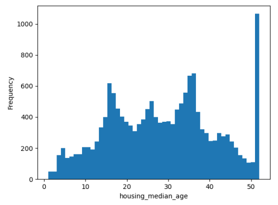
We analyzed the distributions of the housing.csv, using the code we wrote in analysis.py. The resulting histograms can be seen in Figure 2 below. None of the columns are uniformly distributed. Some are close to the gaussian normal distribution, however they are all skewed and most of them have outliers, so we decided that RobustScaler would be the best option for normalization.



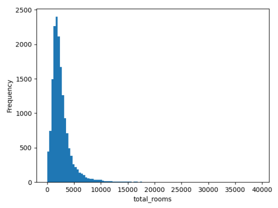
(a) Latitude



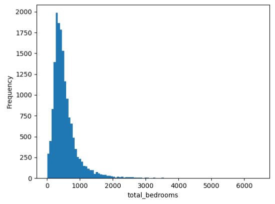
(b) Longitude



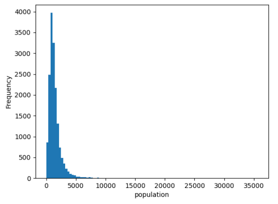
(c) Housing median age



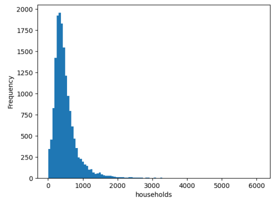
(d) Total rooms



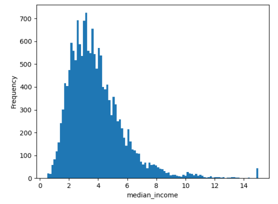
(e) Total bedrooms



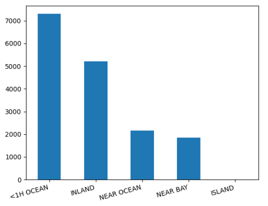
(f) Population



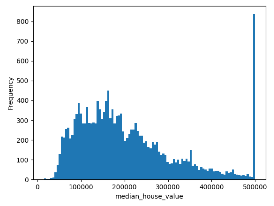
(g) Households



(h) Median income



(i) Ocean proximity



(j) Median house value

Figure 2: housing.csv distributions

The preprocessor function does the following for the input array, plus saves the parameters if its the training set to be used on test set:

- Normalizes all of the numerical columns using sklearn RobustScaler.
- Fills the empty values with mean of the column.
- Replaces the textual column ocean_proximity with one-hot-encoding.

For the target array, it normalizes the array using RobustScaler, and saves the scaler parameters so the output can be scaled back using inverse transform during the predict stage.

## 1.3 Architecture of the Network

The network contains input, 1 output layer and 3 hidden layers. We decided to use 3 hidden layers, thinking that it will be the right balance between not overfitting and achieving good accuracy. The activation function used on each layer but the output layer is the ReLU activation. Each of those layers are fully connected and is followed by a dropout layer. As we are not dealing with images there is no need for a convolution layer and it makes sense to have each neuron connected to the all neurons of the next layer. The diagram of the network can be found on the title page.

### 1.3.1 Dropout

The dropout layers were implemented into the architecture to reduce possibility of the over-fitting. It deactivates the neurons randomly after each fully connected layer. The random deactivation of the neurons during training leads to simulation of assembling neural networks with different architectures, therefore reducing the chance of over fitting.

| Layer | Type | Input | Activation | Output |
|---|---|---|---|---|
| inpt | Input | No. X feat. x1 | - | No. X feat. x1 |
| hid1 | Filly Connected | No. X feat. x1 | ReLU | H1 x 1 |
| drop1 | Dropout | H1 x 1 | - | H1 x 1 |
| hid2 | Filly Connected | H1 x 1 | ReLU | H2 x 1 |
| drop2 | Dropout | H2 x 1 | - | H2 x 1 |
| hid3 | Filly Connected | H2 x 1 | ReLU | H3 x 1 |
| drop3 | Dropout | H3 x 1 | - | H3 x 1 |
| outp | Output | H3 x 1 | - | 1 x 1 |

### 1.3.2 Activation function: Rectified Linear Unit (ReLU)

The computation of the ReLU activation function is simpler than others activation functions. For example, it does not have the exponentiation operation found in the sigmoid activation function. In addition, the ReLU activation function makes model training easier when using different parameter initialization methods, as in contrast to saturated functions, the gradient of the ReLU activation function in the positive interval is always one. Looking at (Figure 3) the ReLU function's slope is not close to zero for higher positive values of x. This helps the optimization to converge faster. For negative values of x, the slope is still zero, meaning that the weights attached to the node are not trained, potentially becoming dead but most of the neurons in a neural network usually end up having positive values.

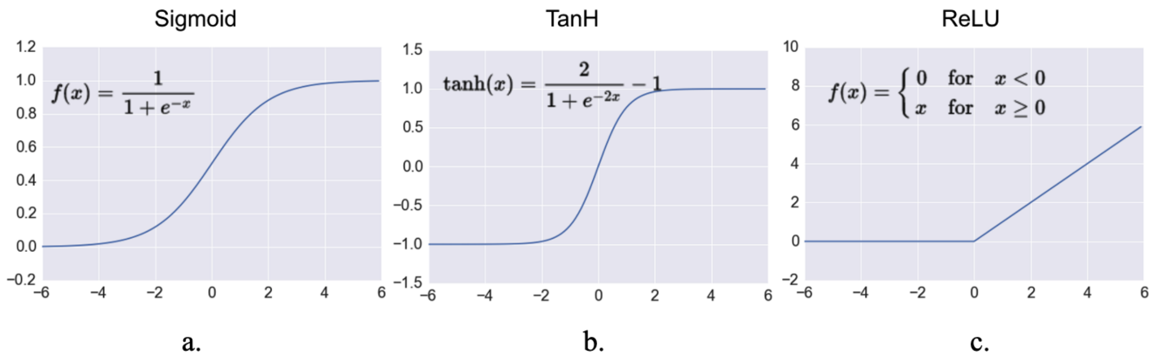


Figure 3: Most used activation functions: (a) Logistic Sigmoid (b) Hyperbolic Tangent and (c) Rectified Linear Unit.

## 1.4 Fit Method

This method trains the network that was initiated in the constructor. Firstly, the network that was initialised is set to be in the training mode(*net.train()*). The workflow of the fit method is summarised in Figure 4. In addition it plots the loss over epoch for training.

### 1.4.1 Optimisation and Loss

Optimisation is required to reduce the losses and to provide most accurate result possible. In fit method we decided to implement Adam optimizer. It adaptively selects the learning rate for each parameter, therefore, in comparison to the Stochastic Gradient Descend method, it does not require as careful tuning of the learning rate parameter. In addition, it potentially may speed up the training if the learning rate across the parameters varies. The earlier layers the gradient might be small, therefore increasing the learning rate of the relevant parameter might speed up the training.

For the loss function we were choosing between the Mean Square Error and the Huber Loss, the latter is good with outliers. It is very similar to the way MSE penalises, where if the prediction is off by a small amount it is barely penalised, if it is off by a bigger amount the penalisation is very high.

### 1.4.2 Additional Implementations

In this training method we implement batch training and shuffling. The data are assigned randomly to the batches. The combination of the training and shuffling allows to ensure that the optimisation will not converge at the local minima, as by changing the order of the rows, we also change the local minimas. In addition, it is suggested that it may lead to faster convergence by having the randomisation.

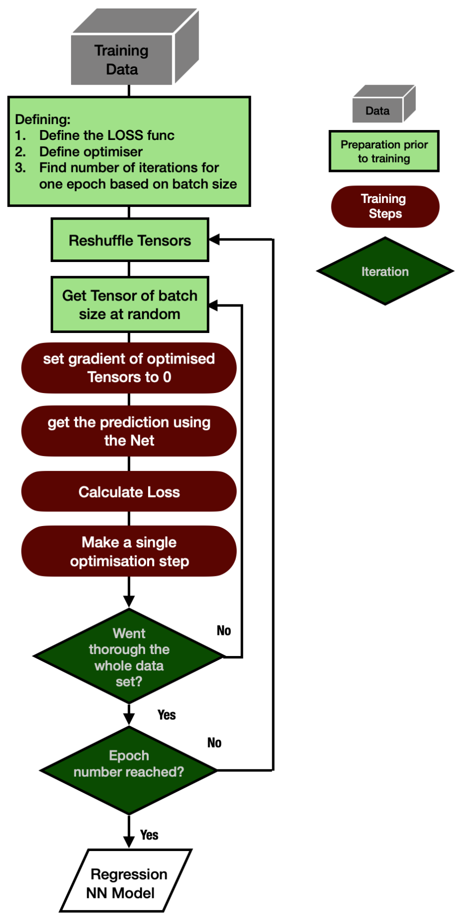

Training Data
Defining:
1. Define the LOSS func
2. Define optimiser
3. Find number of iterations for one epoch based on batch size
Reshuffle Tensors
Get Tensor of batch size at random
set gradient of optimised Tensors to 0
get the prediction using the Net
Calculate Loss
Make a single optimisation step
Went thorough the whole data set?
No
Yes
Epoch number reached?
No
Yes
Regression NN Model
Data
Preparation prior to training
Training Steps
Iteration

Figure 4: The summary of the fit method workflow

## 2 Evaluation of the Architecture

### 2.1 Prediction Method

The prediction method sets the trained model in the evaluation mode (*net.eval()*), which notifies all the layers that you are evaluating, therefore dropout layers will work in eval mode instead of training mode, becoming less computationally expensive. We then pass the x tensor to the model and run it. Note, it is run with *torch.no grad()*, which deactivates the autograd engine, reducing memory usage and speeding up the calculations. In this setting, the model can not access the backdrop, which is not needed when evaluating. Lastly, We inverse scale the output we got.

### 2.2 Evaluation Method

We use the *score* method for the evaluation of the regressor. Because it is a regression task, we are using root mean squared error for evaluation. We get the predicted value from the *predict* method, and calculate RMSE using the output of the *predict* function and the ground truth given as input.

## 3 Fine-tuning the Architecture

### 3.1 Methodology

To look for the best hyperparameters, we decided to use sklearn RandomizedSearchCV as it is less computationally expensive, and has lower chance to overfit than the exhaustive GridSearchCV. However, RandomizedSearchCV has its own weaknesses. Because it is randomized, it does not converge to the same results on each run, thus does not give us the very best possible hyperparameters. Plus it does not do a continuous search, so we cannot analyse and comment on each hyperparameter's effect on performance.

The search scoring looks to minimise the MSE. In addition it performs 5-fold Cross Validation during the search, taking an average score for the model. The search iterates the calculations 3 times. As we are using Random Search it is not as computationally expensive to conduct this set up in comparison to Grid Search. Due the use of the cross validation, we split the data into 80/20 train, test sets. Although the fine-tuning method using Bayesian would be preferable, i.e. (Hypersearch library), as it builds a probability model of the objective function to propose smarter choices for the next set of hyperparameters to evaluate. However, due to the structure of the coursework we could not implement this.

## 3.2 Results

We are looking for the following best performing hyperparameters: number of epochs, learning rate, batch-size for batch learning, number of nodes in hidden layers, and dropout rate. The learning rate is important to the optimiser, and even though we are using Adam, which is less sensitive the learning rate due to its adaptive approach, it is still important to tune it.

As explained before, we cannot analyze each hyperparameter's effect on performance in detail due to randomized search. However, after running multiple randomized searches, we have been able to identify a trend. Although the combinations of the tuned parameters is always different, the general best parameter, will have the epoch of around 600 - 750, the dropout rate of 0.1, learning rate of 0.002 or above, the batch size of either 64 or 96, and the number of nodes in the hidden layers decrease at each consecutive layer. The learning rate and the batch size seem to be interlinked, with larger batch size the learning rate was smaller. It is suggested that the learning rate can compensate for the larger batch size, therefore overall result should be similar. Likewise, higher dropout rate imposes larger number of nodes in the hidden layers, and once again results in similar structure.

## 3.3 Final Model

Our best performing model has the following parameters and architecture:

Training Parameters:
*epoch: 720, batch size: 64, learning rate: 0.001*

Net Architecture:
*(hid1): Linear(in features=13, out features=70)*
*(hid2): Linear(in features=70, out features=37)*
*(hid3): Linear(in features=37, out features=19)*
*(out) : Linear(in features=19, out features=1)*
*(drop): Dropout(p=0.1)*

The model's loss graph in Figure 5 below shows the improvement during the hypertuning. Depending on the way the data is split for training/testing, the final model achieves rmse between 48k - 55k.

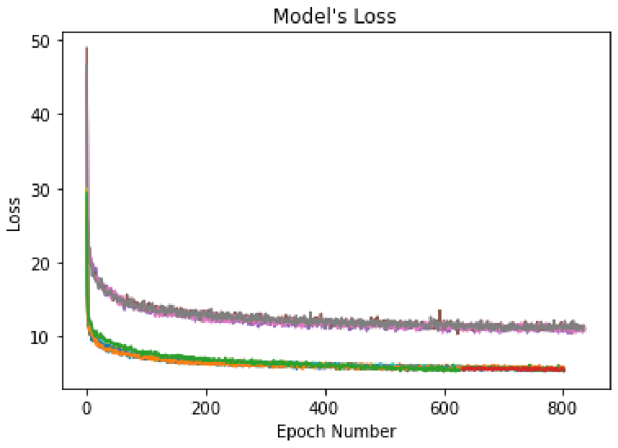


Figure 5: Loss improvement during hypertuning