Imperial College London

NEURAL NETWORKS COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

# CO553 - Introduction to Machine Learning

*Authors:*
Daniel Cahn
Guilherme Freire
Alexandre Allani
Bohua Peng

Date: November 25, 2019

Commit Hash: dbf57d8c3bbfe2f1bfebce7448285b0deb16005f

# Classification Task

## Architecture

We began with a simple architecture, based on our collective experiences with similar classification problems. We started with an architecture comprised of three hidden layers, each with 64 neurons, and separated by ReLU activation layers, followed by an output layer of a single neuron with sigmoid activation. For binary classification, sigmoid activation is helpful for normalising the output between 0 and 1, corresponding with a probability of a sample being in the 0 or 1 class. We initially trained our model using Mean Squared Error as a loss function, a learning rate of 0.001, and an Adam optimizer.

## Evaluation

We began evaluation by evaluating performance on a validation set, using a 0.15 validation split of the initial data. During training, we measured the performance of the model by validation loss, validation accuracy and validation AUC. We use validation loss as a good early indicator of overfitting; as loss begins to plateau, we can expect the model to begin to overfit the data, and we can stop training early. Validation accuracy is a simple indicator of performance, but is less helpful on our imbalanced dataset, especially given the 1:10 ratio between those who made claims and those who didn't, in our dataset. We use the AUC of the ROC curve to keep track of the precision and recall of our model, balancing the need for both our positive and negative classes to be classified correctly. These three metrics combined gave us a good understanding of the evolution of the model through training.

Training in this regime, we were able to obtain 90% accuracy and a little over 50% AUC of ROC on a test set. It is important to note that at this stage the test set was not balanced with respect to the labels of examples, and therefore pure accuracy can be a misleading metric.

## Improvements

We quickly found that the imbalanced nature of the training set lead to slower training on the positive class, as when a batch is sampled from the data it will mostly contain mostly data with label 0. The network is thus unlikely to be greatly affected by the 1s in any batch, instead learning the 1s to be outliers. We therefore implement a batch generator to ensure that classes are balanced, according to a specified ratio. We also implement a custom split function to split between training, validation, and test sets to ensure a specified ratio between 0s and 1s. We also replace early stopping in favour of a learning rate decrement during plateaus, so that the model can improve albeit more slowly, to still avoid overfitting. We also changed the

loss function from using the Mean Squared Error to using the Binary Cross-Entropy, to ensure that the loss accounted sufficiently for the minority class.

After initial manipulation of hyperparameters based on manual analysis of performance, we switch to automated Hyper-Parameter Optimization, more specifically a random hyperparameter search. The choice of a random search seemed more appropriate over grid search because it allows us to search a much larger space of hyper parameters(Random Search for Hyper-Parameter Optimization). In order to maximize our chances of finding a good model, we made almost every aspect of our network tunable. The tunable hyperparameters are:

- activation functions (chosen from Leaky ReLu, ReLU, sigmoid or tanh);
- batch sizes (chosen from increasing powers of 2 from $2^4$ to $2^8$);
- epochs to train (chosen uniformly from 10 to 60);
- learning rate (chosen uniformly in logspace from $10^{-1}$ to $10^{-6}$); optimizer (chosen from Adam, Adagrad, Nadam, RMSprop and SGD);
- coefficient for L1 regularization (chosen uniformly in logspace from $10^{-5}$ to $10^0$);
- coefficient for L2 regularization (chosen like L1 coefficient);
- number of hidden layers in the network (chosen from 1 to 4);
- and number of neurons per layer (chosen from increasing powers of 2 from $2^4$ to $2^{10}$).

Of course, the input and output dimensions are fixed, as they are problem dependent. We use AUC as the metric for our randomized hyperparameter search to choose the best model.

The final hyperparameter search was looking for the model with best AUC and after running on 200 random samples of hyperparameters was able to obtain an AUC of 0.695. The final model uses the following hyperparameters: Stochastic Gradient Descent as the optimizer, learning rate of 0.004, L1 coefficient of 0.189, L2 coefficient of 1.0, trained for 20 epochs with a batch size of 128, model made of three layers with sigmoid activations with 512, 4, 4 neurons respectively.

# Insurance Premiums

## Pre-processing

To pre-process the data, we begin by transforming categorical fields using One-Hot Encoding, where categorical fields such as fuel type are converted into a vector of 0s and 1s, where a 1 is placed in the column where a sample matches a column, and 0s are placed everywhere else. For categorical fields such as for the various codes, we first remove any entries that appear fewer than 100 times in the training set, to avoid over-fitting, and have them replaced with a vector of zeros during the One-Hot transformation.

For ordinal fields, including `pol_coverage`, `pol_pay_freq`, and `pol_usage`, we use ordinal encoding, converting increasing values to increasing integers. Importantly, we choose not to convert in a more complicated way for ordinal fields that are not uniformly separated. For exactly, for `pol_pay_freq` we could have monthly be 1 and quarterly be four, but we prefer to let the Neural Network handle any non-linear behaviour in the fields.

For fields that have `N/A`, including ones where `N/A` is a zero, like `drv_age2`, we are careful to convert the number to the mean of the non-`N/A` or non-zero entries. Finally, we transform all numerical fields by scaling them according to their Z-Score, because most of them are normally distributed.

## Balancing the dataset

After the preprocessing and the first few tests, we were able to achieve around 90% accuracy and but a really bad recall, and AUC around 50%, i.e as good as random. We found that the accuracy was high, because our network nearly always predicted 0, the majority category, and nearly never predicted 1. We therefore balance our training batches dataset, as in part 2, and balance to a custom ratio how the data is split between the training, validation, and test sets. Balancing the data increased the performance of our neural network with an AUC of 0.82. We are aware that balancing the data, biases our neural network since it sees more times data labeled 1 than data labeled 0, however it leads to better performance as measured by AUC, and is thus likely to perform better in determining insurance premiums.

## Network Architecture

After pre-processing our data using one-hot encoding, we inevitably have very large and sparse vectors. Additionally, we leave room for the network to over-train on far less relevant parts of the feature-space, for example specific information on the drivers location; while a town's population might have some influence on the likelihood of an accident, it's unlikely to play a large influence. There are many ways to deal with these issues, and we found that using input stream worked best for our purposes. We implement four input layers that respectively take input features

related to the four categories of features: **vehicle**, **policy**, **driver**, and **location**. The four input layers each reduce the dimensionality significantly, hopefully creating a smaller feature space containing the relevant data from the inputs. This also serves to reduce over-fitting, by fitting the network on features like how risky a certain car model is, rather than how likely an accident is in that particular model. After the first layer, we concatenate results and feed them through the hidden layers, each separated by ReLu, and finally output a single neuron with a Sigmoid activation. Once again, Sigmoid allows for normalizing probabilities of an accident to between 0 and 1.