Neural Networks

Analog to what we did with other Machine Learning approaches, here a simple neural network was trained to predict the amount a customer spends.

An extended dataset was created with engineered features that would not bring data leakage. Regarding the features used from the extended dataset:

- ‘age’, 'annual_income', 'loyalty_score', 'purchase_frequency', from the original dataset.
- Plus: 'spend_per_purchase', 'spend_to_income_ratio', 'log_annual_income', 'log_purchase_frequency', 'region_grouped', 'is_high_value', 'is_champion'.
- Details can be found in 0.feature_engineering.ipynb in folder src.

The target to predict was ‘purchase_amount’.

Description of how it works

The process happens in two files, summarized here:

FNN_KFolds.ipynb: runs each architecture for the 4 folds, calculates the average error (mse), manually checks how the curve losses act within the folds, sends the run parameters and several metrics to MLflow, compare in MLflow, pick the model with lower error and smooth behaviour

FNN_evaluate.ipynb: runs the picked architecture, calculates the error metrics on the validation subset and on the test subset.

Seed and random_state

Several commands are used within the notebooks to make results reproducible, related to seed and random_state in the split, but also a customized function init_weights(m) to make the neural network weight initialization reproducible.

Splitting: train, validation, test

In FNN_KFolds.ipynb, it splits the dataset in 80% training and 20% holdout, untouched.

In FNN_evaluate.ipynb, it does the same reproducible split, then splits the holdout in 50%, validation and test.

Feature preprocessing

Numerical features are scaled and categorical features are one-hot encoded, after split to minimize data leakage risk.

In FNN_evaluate.ipynb, after the split, for numerical features, I fit the scaler to train subset only, transform in the train, validation and test subsets.

In FNN_evaluate.ipynb, after the split, for categorical features, I fit the encoder to train subset only, transform in the train, validation and test subsets.

Cross validation

KFold 4 folds, each manually validated, putting special attention to the Loss Curve. KFold will use 3 folds as a KFold training subset and 1 fold as a KFold validation subset.

Architecture

I started with a simple architecture of 2 layers and expanded up to 4 hidden layers. I tried an small setting of 8x8x8x8 and went bigger from there. I tried increasing architectures, like 4x8x16x32 and decreasing architectures, like 32x16x8x4. The biggest I went was 8192x4096x2048x1042, which was up to 15,000+ neurons, up to 46M+ parameters; such run took approximately 45 minutes.

Loss and Optimizer

The loss was set as the mean squared error (mse). The optimizer was Adam.

GPU setting

The environment was tested and able to use the GPU. The subsets of features and target were transferred to the GPU, as well as the model. That way, the tensors calculations were taking place in the GPU. The hardware monitoring was showing high activity in the GPU cuda cores and vram, highly noticeable during bigger runs.

Adaptive Learning Self-adjusting Parameters Algorithm

I set an algorithm that when the absolute value of the gradient became smaller than threshold, it increased the batch_size by a multiplication factor and it decreased the learning_rate by another factor. The maximum value for batch_size is the size of the subset. A minimum value of learning_rate was set. The algorithm was set in a way that used the better performance of a smaller batch_size and higher learning_rate, to quickly approach the converging area without starting unstable behaviour, and then moving swiftly to the full batch_size and smaller learning_rate.

Early Stop Algorithm

The loop was set to run 10,000 epochs. A patience parameter was triggering an early stop if consecutives epochs obtained no improvement in the KFold validation subset. Too early stops were yielding worse results, patience parameter was set high.

Logging architecture and results in MLFlow

The model itself, a group of parameters and a group of resulting metrics were logged in a local MLFlow server. MLFlow has a nice feature to visualize the comparison of results.

But if the results are too distant from each other, the automatic scaling of the feature does not allow to actually find out the best one, then you need to look at the actual table of results in MLFlow. This can be observed in “nn_MLFlow_comparison_not_useful.jpg” in same folder where this report is.

To be able to see how the comparison plot is useful when is readable, see “nn_MLFlow_comparison_useful_1.jpg” and “nn_MLFlow_comparison_useful_2.jpg”, in same folder where this report is.

Let's remind, that after MLFlow showed the best architecture, then that architecture was run under FNN_evaluate.ipynb and metrics were measured in the validation subset and test subset.

Highlights of the output

The best neural network had the architecture: 32 neurons, 16 neurons, 8 neurons, 4 neurons, 1 output neuron.

On the subset test, the Mean Absolute Error (mae) was 1.11 and R2 was 1.00.

To give contest to the average error of about $1, the target purchase_amount has an average of $425 and an standard deviation of $140.