Notes to Models

This section describes the statistical models that we used to predict apartment prices based on their feature set. The goal is to contruct different models of varying complexity and compare both their in-sample and their out-of-sample performance on the Airbnb data.

At this point in the data pipeline all preprocessing steps are completed such that all covariates have the correct data type to be used by the models. Whereas the selected classical Machine Learning models described in section 1 use input features as `numpy` arrays, the Neural Network in section 2 first converts them to *Tensors*, the canonical data type across all popular Deep Learning Frameworks.

1 Classical Models

When attacking the prediction task directly with a Neural Network, there are two questions that we cannot answer:

1. How do we know if a performance metric is *good*?
2. Is fitting a Neural Network appropriate in the first place? Maybe we can get away with a much simpler, more interpretable and, thus, preferable model?

In order to get some insights, we selected four classical Machine Learning models of varying complexity from the `scikit-learn` library to serve as benchmark models for our custom Neural Net. These include:

- Linear Regression: simple, well understood in terms of underlying theory and highly interpretable.

- Ridge Regression: still very interpretable with a closed form analytical solution, adds one hyperparameter to the equation.
- Random Forest: very flexible model with many hyperparameters determining e.g. the number of regression trees and the tree depth. Can be applied to many contexts and often works 'out of the box'.
- Histogram-Based Gradient Boosting: modern and fast tree-based gradient boosting algorithm. Comes with a large number of tunable hyperparameters, some of them similar to the Random Forest parameters, some of them more specific to the *Boosting* instead of the *Bagging* approach such as the learning rate. Similar to the very popular XGBoost and LightGBM implementations that regularly outperform deep Neural Networks in Kaggle competitions on tabular data.

All models were fitted with *Cross Validation*. Except for the Linear Regression this included (extensive) *Hyperparameter Tuning*. Since the parameter space for both of the tree-based models is very high-dimensional we used a *Randomized Search* algorithm rather than the more common *Grid Search* approach. This allows for greater variation along more 'important' dimensions while still keeping the computation time feasible.

2 Neural Network

In concurrence with the Seminar's topic the focus of our project is centered around constructing a custom Neural Network that is trained in a supervised setting and learns to map feature combinations to price predictions for the Airbnb data set.

2.1 Architecture

The network's design is restricted by two factors:

1. The number of input features in the *first* layer has to equal the number of features used for prediction.
2. The number of output features in the *last* layer is fixed to 1 since the network directly predicts the price as a scalar quantity.

All intermediary layers and thus both the *width* and the *depth* of the network

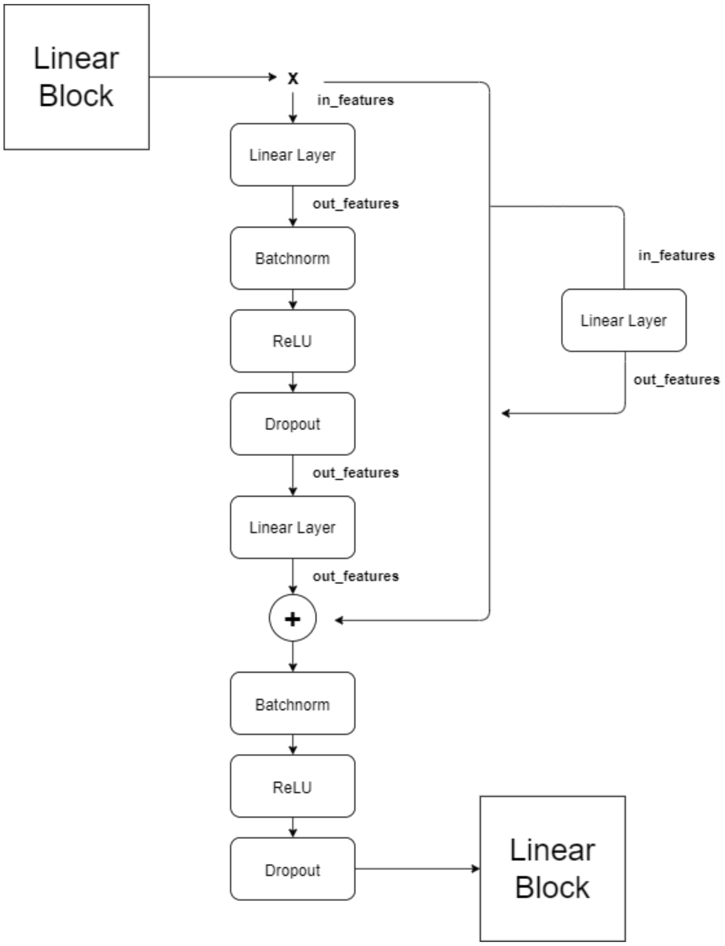

Figure 1: Architecture of each Block in the Fully Connected Neural Network

are free to choose. After experimenting with many different configurations, mostly inspired by empirical findings in the literature, we finally settled with a fairly simple *block structure*.

Starting from roughly 60 input features, the network width is first blown up to 256 features before steadily decreasing it again to a single output neuron in the final layer. More precisely, the architecture consists of 6 intermediary **blocks** with 64, 128, 256, 128, 64 and 8 output features in addition to the linear input and output layers.

The structure of each so-called `Linear Block` is displayed in figure 1. We now explain the reasoning behind including or excluding elements in this configuration.

Linear Layer

Each `Linear Block` is based on a four-element cycle that is repeated twice. Each cycle starts with a Fully-Connected Layer containing the majority of trainable weights in the network. The first linear layer determines the output dimension of this block, in our case mostly doubling or halving the number of input features. Since both linear layers are immediately followed by a `Batchnorm` Layer, the additional `Bias` term is redundant and thus omitted.

Batchnorm

`Batchnorm` Layers often lead to a more well-behaved training process. By means of first standardizing the activations within each minibatch and rescaling them afterwards, the network *learns* a convenient magnitude of the activations. This tends to be particularly effective for *saturating* activation functions such as `sigmoid` or `tanh` activations, positive effects for training, however, have also been observed for the `ReLU` activation function, which is used in our design. Of all individual components of the `Linear Block` structure, Batchnorm was the least important and provided only marginal benefits.

ReLU

Out of the numerous options for including nonlinearities into the network and thus expanding the function space that the network is able to learn, we settled for the default choice of the *Rectified Linear Unit* activation. While briefly experimenting with the related `Leaky ReLU` and `Exponential ReLU`, the differences were insignificant and could be easily attributed to other factors such as random weight initialization.

Dropout

The `Dropout` Layer at the end of each cycle turned out to be the single most impactful factor for the network's *generalization* ability. In fact, due to the different behaviour of dropout during training and inference, we could directly control the relative performance on training and validation set by varying the dropout probability p.

More specifically, the network overfitted drastically by setting p to zero. This actually provided the valuable insight that the current function class is flexible enough to model the task at hand properly. Thus, there was no need to artificially expand the network size beyond the current quite compact architecture with roughly 280.000 trainable parameters, saving computation time as a nice side-effect.

Figure 2 displays the shift in performance metrics on training and validation set when steadily increasing the dropout probability p. While not beneficial for overall performance, dropout delivers a leverage to make the model perform much better on out-of-sample data compared to the dataset that it was actually trained on.

This behaviour can be explained by the *ensemble* interpretation of dropout: In each epoch, a different subset of neurons is excluded to contribute to training and thus a distinct subset model, nested in the full model, is responsible for the prediction output. If the dropout probability is extremely high, these subset models are simply too different from each other and do *not agree* on a common range of sensible prediction values.

In contrast, during inference (and possible deployment of the model), any randomness should be avoided and the network may use all of its neurons. Although there is no influence on the model *weights* in this stage, the model pools all subset models together and *averages out* the diverging effects of each component, ultimately resulting in better performance metrics.

Residual Connections

One special feature of the `Linear Block` architecture in 1 is the *Residual Connection* adding the block input to the output of the second linear layer within each block. This component was originally implemented having a fairly deep network in mind since *Residual Blocks* introduced by the famous `ResNet` architecture are a very popular tool for stable gradient flow by preventing the issue of vanishing gradients.

As mentioned above the actual depth of the final model turned out to be limited. Therefore we decided to enable inclusion and exclusion of skip-connections in the network by a `boolean` parameter allowing for easy comparisons between the model designs. Although the residual component did not have a major impact on performance, it was slightly beneficial in most settings such that there was no reason to remove it from the architecture.

There are two aspects worth mentioning about the residual connection in figure 1:

1. The addition of the input with the output of the second linear layer is performed **before** the final `Batchnorm` and `ReLU` layers of this block. This guarantees a nonnegative input in the expected range for the next `Linear Block` since the raw outputs of a fully connected layer are (in theory) unbounded.

2. One caveat of the residual architecture is to ensure that at addition time all contributing terms share the same dimensions. In case of a fully connected layer dealing with two-dimensional tensors, this amounts to an equal number of features in the second dimension. As illustrated by the annotations in figure 1, this restriction requires an extra step if the number of input features is different from the number of output features in this particular block. In our case the dimnesions adjustment is performed by an additional linear layer exclusively for the block input x.

2.2 Further (Hyper)-Parameters

Optimizer and Weight Decay
We used the popular `Adam` algorithm for training the Neural Network. Compared to vanilla Gradient Descent (with optional momentum) the loss curves during the training process looked much smoother. Moreover, while pure Gradient Descent slightly benefitted from an additonal weight penalty, neither the classical L_2 weight decay nor any other weight regularizer indicated improvements for training with Adam.

Loss Function
For the regression task of predicting the continuous price variable, we chose the standard continuously differentiable *Mean Squared Error* loss function. This come with the additional nice property that, under the assumption of a conditional normal distribution of the price with constant variance, i.e. $y_i \mid \mathbf{x}_i \sim \mathcal{N}(\mu_i, \sigma^2)$, the model *minimizing* the MSE-Loss simultaneously *maximizes* the Gaussian Likelihood, providing a *probabilistic* interpretation.

Learning Rate
Throughout training we kept the learning rate fixed at 0.01. Using a larger learning rate at early stages of training showed promising effects such that we experimented with different learning rate *schedulers*, either decreasing the learning rate in fixed intervals linearly or exponentially, or decreasing the

learning rate whenever the loss seems to stagnate for some given window.

However, the benefits of faster loss decay at the beginning of training were outweighed by too small learning rates at later stages in the training loop. As a consequence, all schedulers resulted in stagnating loss much faster than training with a constant learning rate from start to end.

Moreover, since computation time was not too expensive due to the compact network size, we could make up for the slower initial progress that could be achieved with a scheduler by simply training the model for more epochs (while saving the model weights at the point of best performance on the validation set), which ultimately lead to a lower training and validation loss.

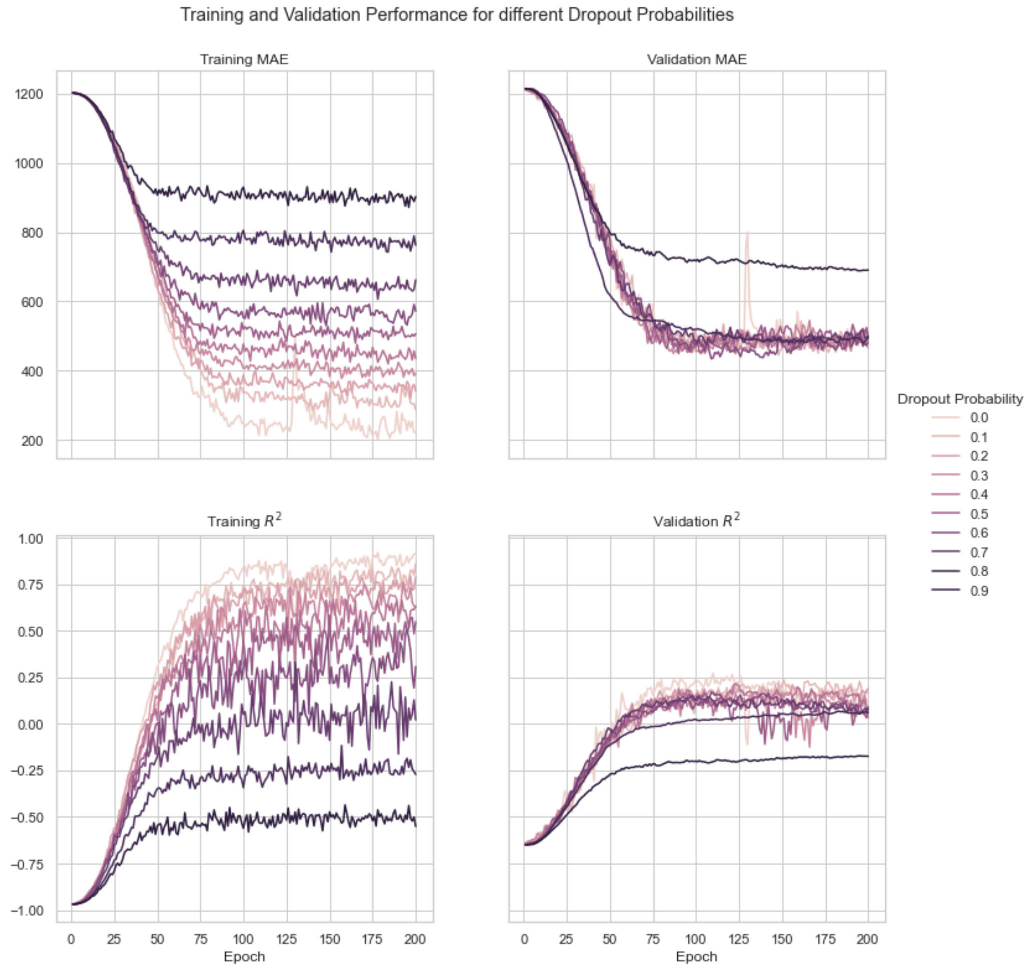

Figure 2: Impact of Dropout Probability on Training and Validation Performance