Notes to Fully Connected Neural Network

Contents

1 Architecture

When the main Neural Network enters the data pipeline, all input variables have already been transformed to numeric predictors, i.e. originally categorical variables are encoded as *Dummy* Vectors and inherently numeric variables are zero centered and scaled to unit variance. Thus, both the price variable as the outcome as well as all predictor variables can be transformed to **Tensors**, the canonical data type across all popular Deep Learning Frameworks.

The network design choice was only restricted by two factors:

1. The number of input features in the *first* layer necessarily equals the number of features contained in the input data set.
2. The number of output features in the *last* layer is fixed to 1, since the network directly predicts the price as a single scalar.

Hence, both the *width* and the *depth* of the network as well as all intermediate components are free to choose.

After experimenting with many different configurations that proved to empricially be beneficial 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 and then, almost symmetrically, steadily decreased to the single output neuron in the final layer. More precisely, the architecture consists of 6 **blocks** with 64, 128, 256, 128, 64 and 8 output features in addition to the first linear layer connecting the input features with the first block and the last layer connecting the last block with the prediction output.

The structure of each so-called `Linear Block` is displayed in figure 1. The following paragraph explains 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 the block, in our case mostly doubling or halving the number of input features for this block. Since both linear layers are immediately followed by a `Batchnorm` Layer, the additional `Bias` term is redundant (vanishes due to centering during Batchnorm) and is therefore omitted.

Batchnorm
`Batchnorm` Layers are often connected to a more well-behaved training process. By means of first standardizing teh activations within each minibatch and rescaling them afterwards, the networks *learns* a convenient magnitude of the activations. This tends to be particularly impactful for *saturating* activating functions such as the `sigmoid` or `tanh` activations, however positive effects for training 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 only provided 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 its cousins, the `Leaky ReLU` and the `Exponential ReLU` the differences were barely noticeable and could be just as well attributed to other factors such as the random weight initialization.

Dropout

The `Dropout` Layer at the end of each cycle turned out to be the single most impactful factor for *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. By setting p to zero the network overfitted drastically. This actually provided very valuable information proving 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 further, which kept the architecture quite compact with roughly 280.000 trainable parameters and saved a lot of computation time.

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, the dropout delivers a leverage to let the model perform much better on out of sample data compared to the dataset that it 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 small subset models together and *averages out* the diverging effects of each component resulting in better performance metrics.

Residual Connections

One special feature of the `Linear Block` architecture is the *Residual Connection* adding the block input to the output of the second linear layer within each block. This component was originally implemented with the thought of a fairly deep network in mind since *Residual Blocks* introduced by the famous `ResNet` architecture are arguably the most popular tool for a stable gradient flow by preventing the issue of vanishing gradients. As mentioned above the depth of the final model turned out to be limited. We decided to enable inclusion and exclusion of skip-connections in the network by a simple `boolean` input parameter to the main model fitting function 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 model.

There are two aspects worth mentioning about the residual connection in figure 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 outputs of a simple fully connected layer are (in theory) unbounded.
- One caveat of the residual architecture is to ensure that at the time of addition all contributing terms have the same dimension. 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 additional step if the number of *input features* is different from the number of *output features* in this particular block. In our case this adjustment of the feature dimension is performed by means of an additional linear layer specifically for the block input x.

2 Further (Hyper)-Parameters

Optimizer and Weight Decay

Similar to `ReLU` as the default activation function `Adam` has evolved to the default choice for performing the optimisation step when training the neural network. Indeed, the loss curves during the training process looked much smoother when using Adam instead of vanilla Gradient Descent. Additionally, there was no need for introducing a *penalty term* on the weights of the network while using Adam, whereas Gradient Descent (with optional `Momentum`) did benefit slightly from a L_2-Penalty.

Moreover, we did not make use of other penalty implementations such as a L_1 penalty which can often induce *sparseness* into the model. On the one hand adding neither the L_1- nor the L_2-Norm of the weights to the loss function showed promising improvements in performance, on the other hand, for interpretability reasons, we preferred to induce 'sparseness' *explicitly* by manual and automatic feature selection prior to the model training itself rather than relying on *implicit* effects inside of the network 'black box'.

Loss Function

Since we are in a *Regression* context for predicting the continuous price variable, we chose the standard `Mean Squared Errro` loss function. This has the additional nice property that, in case of a normal distribution with constant variance, i.e. price $\sim \mathcal{N}(\mu, \sigma^2)$, the model *minimizing* the MSE-Loss simultaneously *maximizes* the Gaussian Likelihood, providing a *probabilistic* interpretation.

Learning Rate

Throughout training the network with different configurations of hyperparameters we kept the learning rate fixed at 0.01. Since using a larger learning rate at early stages of training did indeed show promising effects, we experimented with different learning rate *schedulers*, either decreasing the learning rate in fixed intervals by a constant or exponential rate 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 was overweighed by the difficulties of too small learning rates at later stages in the training loop. All settings we tried introduced a too aggressive learning rate in flatter regions of the loss surface whereas a constant learning rate still lead to a steady loss decrease.

Moreover, since computation time was not too expensive due to the compact network size, we could easily make up for the slower initial progress with a fixed learning rate by simply training the model for more `epochs` (typically around 200 where the model *weights* were saved at the point of best performance on the validation set), ultimately leading to a lower training and validation loss.

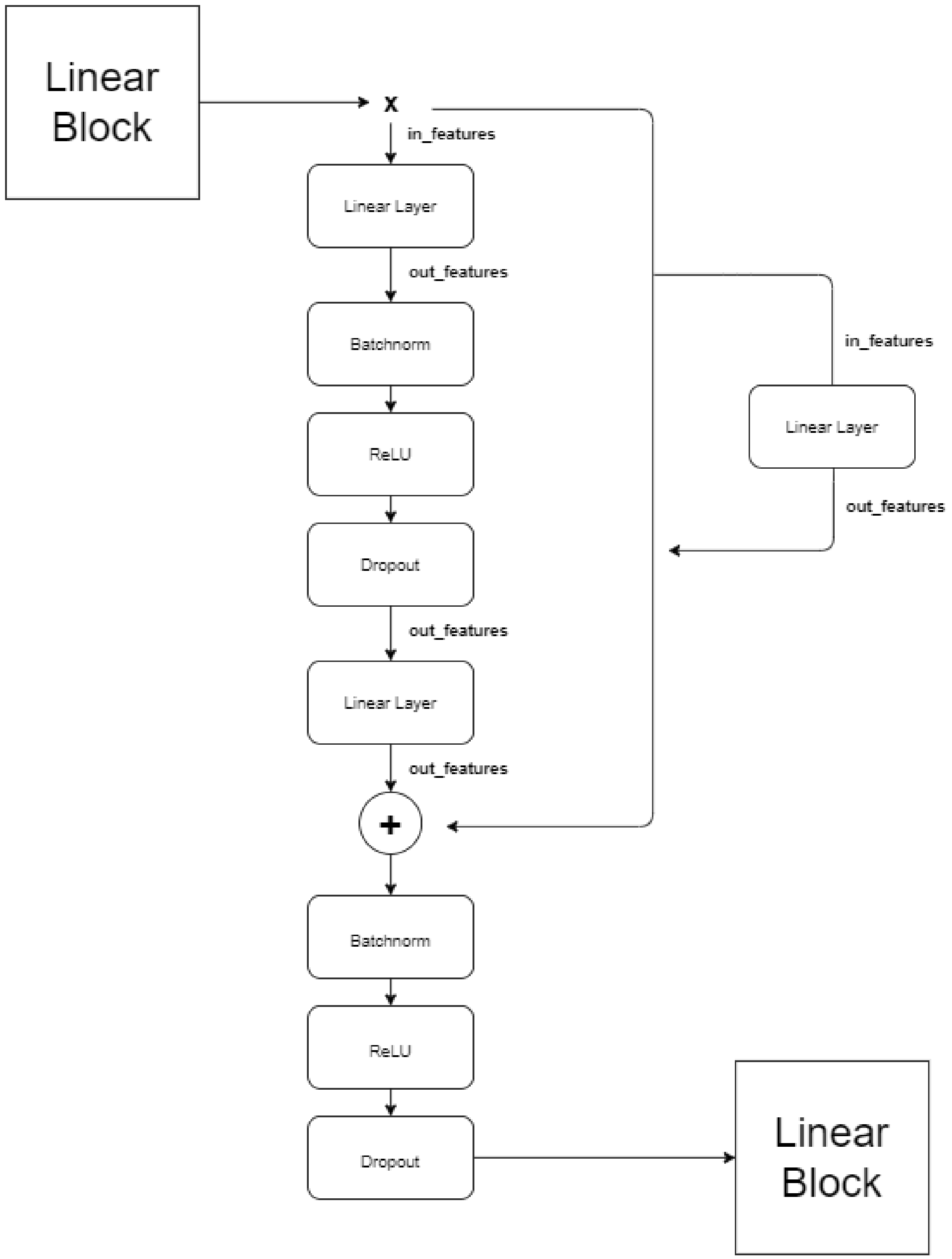

Figure 1: Architecture of each Block in the Fully Connected Neural Network

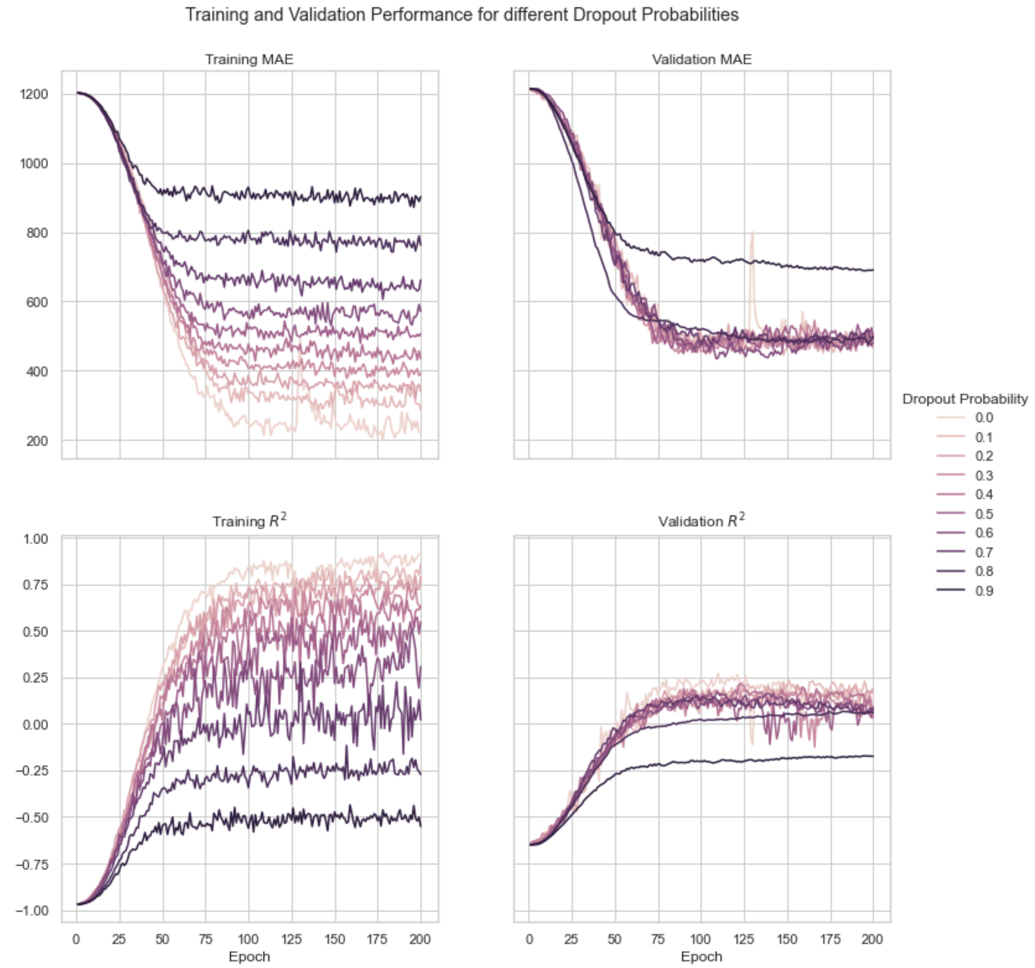

Figure 2: Impact of Dropout Probability on Training and Validation Performance