Your Title

Author 1, Author 2
Chair of Statistics, University of Göttingen

February 10, 2022

Abstract

The text of your abstract. 200 or fewer words.

Keywords: 3 to 6 keywords

1 Introduction

Describe the dataset and and define the problem. Do not explain in detail the basics of ANNs as we did in the introductory lecture.

Figure 1: Caption

2 Methods

2.1 Preprocessing

This section covers all necessary preprocessing steps that build the foundation for all further modeling tasks. Here, we mostly focus on feature *engineering* by extracting information from the image and text data contained in the Airbnb Dataset. However, feature *selection* and transformations of the price variable turned out to be essential components of the preprocessing pipeline and are discussed in Appendix A.2 and Appendix A.3.

In addition, some basic data cleaning steps at the very beginning were necessary. These mostly consisted of converting data types, splitting text-based variables into more convenient numeric or boolean features and aggregating rare categories of categorical variables into one larger *Other* group to stabilize estimation. In order to later use the predictor variables as model inputs all features of categorical nature were transformed to Dummy variables whereas all numeric features were standardized.

Extending the original feature set with new informative covariates which are created from (combinations of) existing variables is a central aspect of every data analysis. Besides many numeric features the Airbnb data contains *images* as well as *reviews* in raw text form. These data types in particular require some extra care which will be the focus of this section.

2.1.1 Image Processing and Modeling

Besides the metric and text-based features the Airbnb Data contains *images* of the listed apartments as well as of the corresponding hosts. This section describes how we used these images (while focusing on the apartment pictures) for the purpose of price prediction.

The main idea was to build a Convolutional Neural Network that predicts the price solely based on the image content itself and add these predictions as well as the number of available images for each listing to the main feature set. Since there exist multiple images per apartment, the predictions were averaged afterwards within each group to obtain an output array of equal length to all remaining features.

Before the images can be used as model input they had to be *scraped* from each listing's webpage. Further, some basic transformations were necessary. Details about these image proprocessing steps can be found in Appendix A.1.

Image Modeling
As mentioned above, we used a pretrained Convolutional Neural Network for modeling. Ideally, due to learning from a large collection of labeled images in a supervised setting, this model is able to extract meaningful features from our own much smaller input data out of the box. As usual in *transfer learning* the weights of the pretrained model are frozen and the output layer is replaced by a trainable custom layer specific to our needs.

One potential issue arises if the dataset used for pretraining differs from the new custom data: If the pretrained network is very deep, the learned features before the final layer could be very specific to the Output Classes of `ImageNet` and not generalize well to our images.

There are multiple options to handle this scenario:

- Out of the vast collection of freely available pretrained models, choose one that is comparably shallow. We chose `ResNet18` with roughly 11 million parameters.
- Do not freeze the weights of the pretrained model completely, but rather fine tune them during training (i.e. modify *all* weights by backpropagating through the entire network). We did not investigate this option further due to its high computational cost.
- Cut the pretrained model before the last layer with the hope that, at this point, very generic and widely applicable features of images are extracted. These features might in theory generalize better to our data. In practice, however, this option did not improve our results significantly.

It turned out that a *single* custom layer, mapping from 512 directly to a single neuron representing the scalar price prediction, was not expressive enough. The performance improved by appending a (small) Fully Connected Network at the end instead containing three layers and `ReLU` activation functions.

To ensure that the chosen design is not majorly flawed, we constructed a separate much smaller Convolutional Neural Network with only a handful of Convolutional Blocks as a benchmark model. Although the performance differences were not as large as desired, the pretrained `ResNet` indeed indicated more promising results.

Image Results
Using only the content of the available images, the pretrained `ResNet18` achieved a Mean Absolute Error of 579 NOK (approx. 58 Euros) on the Validation Set. In comparison, the *Null Model* of always predicting the mean price achieved an MAE of 630 NOK without a log-transformation of the price and a MAE of 569 NOK with a log-transformation. Thus, the raw predictive power of the images alone was very small.

However, the *correlation* of the CNN predictions with the true price was 0.41. This indicates some limitations of the correlation as useful metric on the one hand but at least positive tendencies of the CNN predictions on the other hand. In fact, the network struggled the most with capturing the wide *range* of prices and almost always predicted values close to

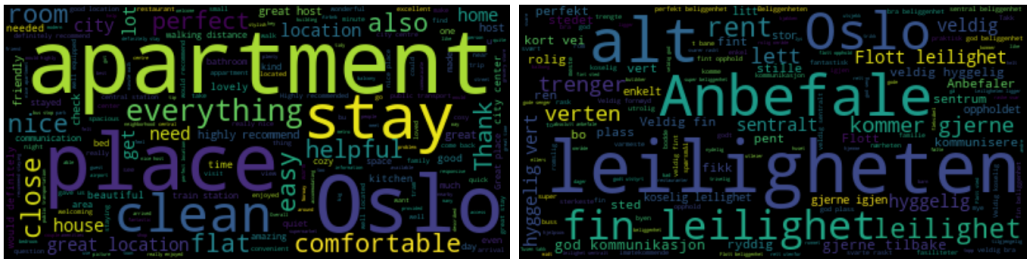

Figure 2: Wordclouds in English and Norwegian

the center of the (log) price distribution.

Although the general idea of categorizing images into price ranges based on image features sounds very appealing, taking a look at the actual input images reveals how challenging this task actually is. Figure 7 in the Appendix displays a random collection of input images. Considering the difficulty of the task it is actually highly doubtful that humans could provide much more accurate predictions.

2.1.2 Reviews

In order to extract the most important information from the reviews, we have performed the following analyses.

First, we used the langdetect package to determine the language of each review. With this information, we tried to get some insights about the internationality of the guests. Since English and Norwegian are the most commonly used languages, we then created two so-called word clouds in these languages to visualize the most frequently used words in the reviews and to give us a short overview of the given ratings. To just have the important words in this representation, a list of given stop words are used to extract them. What is most striking in Figure 2 is that in both predominantly positive words are printed. The largest printed and thus most frequently used words in English are *apartment*, *Oslo* and *place* and further *clean*, *comfortable*, *helpful* and *easy*. Also in the Norwegian plot there are mainly positive expressions like *anbefale* (engl. *recommend*), *fin* (engl. *fine*) and *flott leilighet* (engl. *great apartment*).

The results of the language detection are stored per *listing_id* in a new data frame called *reviews_features*. This data frame contains the number of reviews per listing, the median length of a review, the number of different languages as well as a list of languages in which the reviews for that apartment were written. The percentages of Norwegian and English reviews are also recorded for each listing id. Finally, this data frame was added to the *listings* data frame on which feature selection is performed later. (s. cite sentiment)

In addition, to obtain a more informed analysis of the reviews, we also performed a detailed sentiment analysis for each review. Sentiment analysis is used to detect the underlying emotion of a text. Therefore, it classifies the text as either positive or negative.

To do this, we used the `transformers` package, more specifically we used the `pipeline`

function with the `task` argument set to *sentiment-analysis*, which is used for classifying sequences according to positive or negative sentiments (s. documentation). The used model is DistlBERT (Sanh et al., 2020), a small, fast and light Transformer model, a distilled version of BERT (Devlin et al., 2019) algorithm, which achieves significantly faster results (s. documentation transformers).

Finally, we used the sentiment analysis to determine the ratio of negative reviews to the total number of reviews per listings id, which is also added to the *reviews_feature* data frame.

2.2 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 2.2.1 use input features as `numpy` arrays, the Neural Network in Section 2.2.2 first converts them to *Tensors*, the canonical data type across all popular Deep Learning Frameworks.

2.2.1 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 (Pedregosa et al., 2011) 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 (Chen and Guestrin, 2016) and LightGBM (Ke et al., 2017) 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 (Bergstra and Bengio, 2012) rather than the more common *Grid Search* approach. This allows for greater variation along more 'important' dimensions while still keeping the computation time feasible.

2.2.2 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. The following section highlights the key components of the model's architecture. A brief overview of additional (hyper-)parameters that were of minor importance can be found in Appendix A.4.1.

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 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 3. 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.

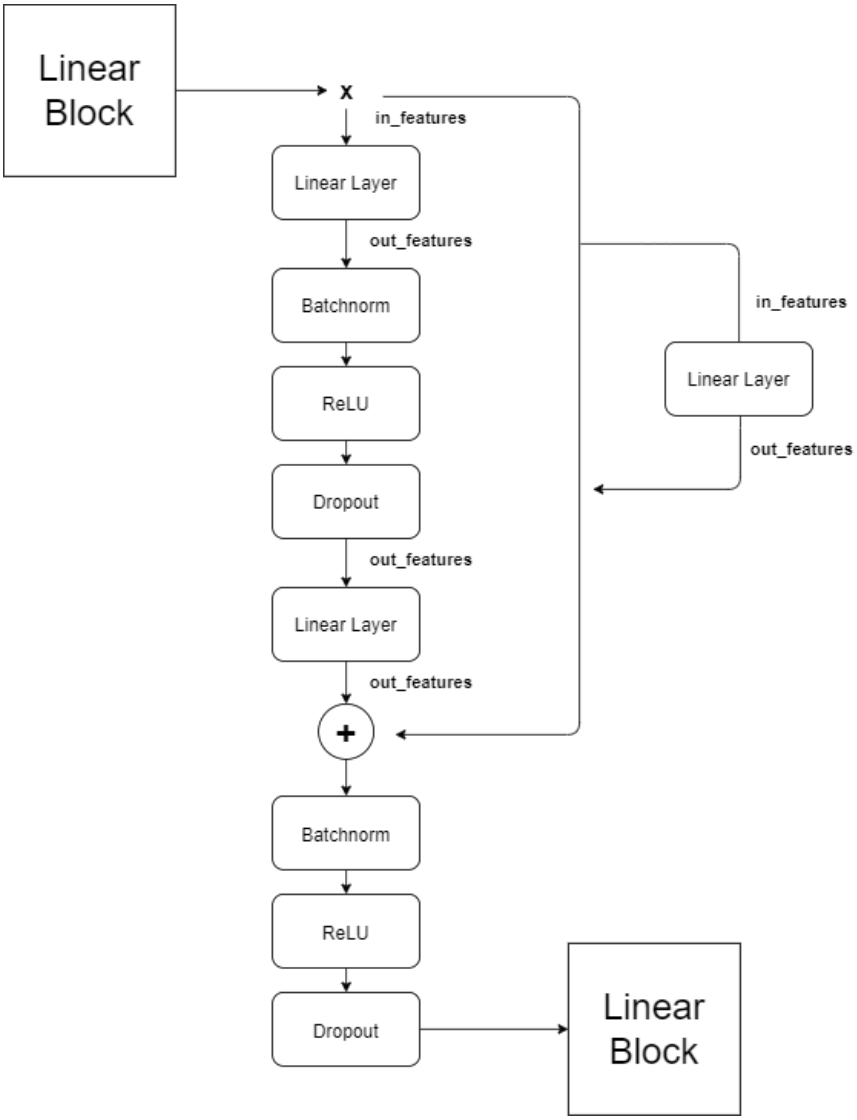

Figure 3: Architecture of each Block in the Fully Connected Neural Network

Batchnorm

`Batchnorm` Layers (Ioffe and Szegedy, 2015) 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 ex-

panding 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 (Srivastava et al., 2014) 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 9 in the Appendix 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 (Goodfellow et al., 2016): 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 Figure 3 is the *Residual Connection* which adds 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 (He et al., 2015) 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 3:

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 3, 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.

3 Results

3.1 Predictive Performance

Figure 4 shows performance metrics for all fitted models on both the Training Set and the Validation Set. Whereas we used the Mean Squared Error Loss for *training* the Neural Net due to its convenient differentiability with regards to backpropagation, we focused on the Mean Absolute Error and the R^2 value for *evaluating* all regression models. The MAE measures absolute deviations between true and predicted price and is therefore interpretable on the same scale as the original data. While the R^2 certainly has some inherent flaws, it provides a scale-independent and highly interpretable measure of overall model fit.

The different colors in Figure 4 indicate how many features were selected by the `RFE` algorithm for this particular fit with 59 total predictor variables. Unsurprisingly, one or two features (in case of the `RFE` the number of bedrooms and the (number of) accomodates) provide too little information to model the task appropriately. However, including merely 5 features (in this case adding the 30 day availability, the indicator variable for the *Frogner* neighbourhood and, notably, the price predictions from the Convolutional Net based on the image data) results in a very competitive performance for most models on the validation set.

The two subplots on the left displaying the MAE tell a very similar story to the subplots on the right that show the R^2: Generally, more features lead to better performance on training and validation set. In case of the flexible Histogram-based Gradient Boosting algorithm and, to some minor extent, for the Random Forest, a high number of input features lead to *overfitting* such that the performance on the training set is far superior to the values on out-of-sample data. In contrast, models with few parameters such as Linear Regression or Ridge Regression generalize very well to the validation set with virtually no performance drop.

One exception to this rule is the highly complex Neural Network which appears to per-

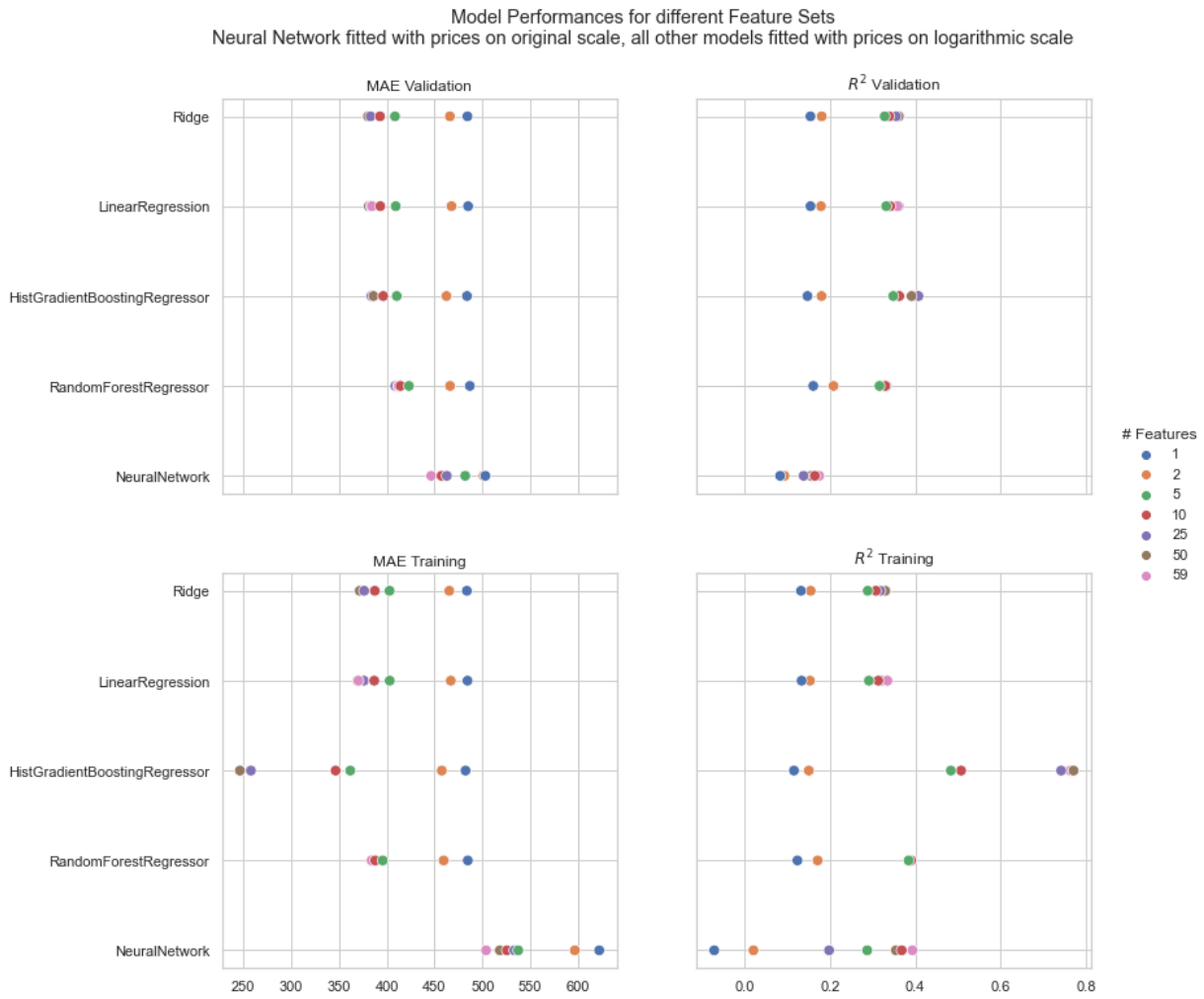

Figure 4: Performance Comparison of Classical Models with Neural Network

form *better* on out-of-sample data. This phenomenon can be explained to some extent by the different behaviour of Dropout and Batchnorm during training and inference and was already discussed in the previous chapter.

When comparing the models within each subplot, all classical Machine Learning models perform similarly on the validation data. This finding emphasizes two aspects:

1. The prediction task does **not** require overly complex models and linear models perform just fine.

2. We can expect predictions within a distance of roughly 400 NOK (40 Euros) on average to the true price. Further, a R^2 value of around 0.4 is the best we can hope for. These statements hold for fitting on the *entire* training set, Section 3.2.2 reveals how the performance radically improves when excluding some observations.

In comparison to all `scikit-learn` models our custom Neural Net appears to underper-

Model	MAE	R^2
Linear Regression	404.709	0.298
Ridge	405.932	0.294
Random Forest	444.166	0.268
HistGradientBoosting	412.243	0.387
Neural Network	402.24	0.333
Top2 Average	404.848	0.296
Top3 Average	399.315	0.343
Top4 Average	404.206	0.332
Top5 Average	408.116	0.27

Table 1: Test Set Performance of Classical Machine Learning Models, our custom Neural Network and Ensemble Predictions

form. We have to keep in mind, though, that evaluation on the validation set is overly optimistic for those models whose hyperparameters were tuned on this data during cross validation. The best simulation to performance on truly unseen data is thus provided by the *test* set that was not used in any step up to this point.

Table 1 compares test set metrics for the best performing models on the validation set within each class (i.e. the Neural Net version with all 59 features and the Random Forest model with only 25 features were selected). In addition, we added some simple ensemble models that predict the average estimate from the top 2, 3, 4 or all five models, respectively.

Quite surpringly, the Neural Net shows a significant performance boost on the Test Set and is now very competitive with all other models. The Random Forest Model generalizes worst and suffers from overfitting given its selected hyperparameter configuration. Averaging predictions from multiple algorithms indicates promising results: The Top 3 Average achieves the lowest out-of-sample Mean Absolute Error of slightly below 400 NOK.

3.2 Understanding and Interpretation

Interpreting the results of high-dimensional and complex statistical models is notoriously difficult. This section approaches the challenge from two opposite angles.

First, we reduce the *complexity* by fitting a simple and well-understood Linear Regression Model with the same features that were selected for the best-performing Neural Network and analyze the coefficients.

Second, we reduce the *dimensionality* by leveraging a different Neural Network and visualize the results in two-dimensional space.

3.2.1 Feature Importance

In order to draw conclusions about which features were most important for the Neural Network one could imagine to compute gradients of the single output node with respect

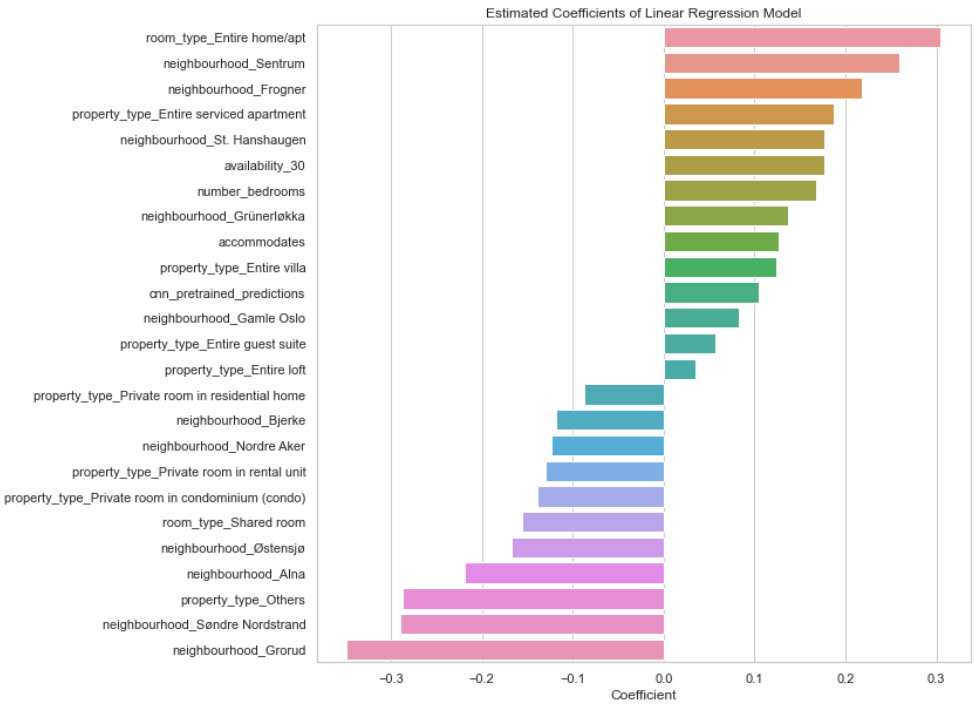

Figure 5: Estimated coefficients of a Linear Regression model for 25 preselected features

to each input node by backpropagating through the entire network. We chose to analyze the feature importance of an auxiliary Linear Regression model instead to provide some suggestions that could potentially apply to the Neural Network as well.

Therefore we preselected 25 of the network's input features with the `RFE` algorithm introduced in the previous chapter and compared the coefficient magnitudes. As noted before, all predictors were standardized such that a meaningful comparison is feasible. The results are shown in Figure 5.

It is worth noting that the two most important features based on various different feature selectors of the `scikit-learn` library were always the number of *bedrooms* and the (number of) *accomodates* in this order, both measuring the apartment's *capacity*.

The coefficient plot, however, is dominated by categorical features: In agreement with human intuition the *room* type, the *property* type and the *neighbourhood* strongly influence the predicted price. Unsurprisingly, the property types *entire home* and *entire villa* are connected with high prices, whereas the the room type *shared room* correlates with cheaper apartments. Notably, the price predictions from the Convolutional Network fitted on the image data indicates a significant positive effect, conditioned on all other selected features.

When analyzing the *marginal* effect of neighbourhood on price by e.g. ordering the neighbourhoods according to their median price, this order is nearly identical to the ranking in Figure 5 with *Frogner* at the top and *Grorud* at the bottom.

Interestingly, apartments in the *Sentrum* (central area) have a large positive coefficient in

Quantile Threshold	MAE	R^2
0.0	443.35	0.16
1.0	337.59	0.51
2.5	282.17	0.53
5.0	240.57	0.54
10.0	214.76	0.49

Table 2: Mean Absolute Error and R^2 value of the Neural Network on the validation set after removing the highest quantiles of the price distribution from the data set

the regression, yet the second to lowest median price. This finding indicates the presence of *confounders*: The city center might plausibly be connected to fewer rooms and smaller apartment sizes overall pulling the median price down. These confounding effects are controlled for in the regression model but not in the naive bivariate analysis.

3.2.2 Sensitivity to Outliers

Table 2 shows performance metrics for the Neural Network fitted on a subset of the data in the validation set. As indicated by the first column the dataset was reduced by successively cutting off observations from the top of the price distribution. More precisely, the first row refers to the entire data whereas the last row excludes the top 10% most expensive apartments.

By omitting just the top 1%, the MAE reduces by over 100 NOK (about 10 Euros) and the R^2 rapidly jumps to over 0.5. This sensitivity to outliers in the price distribution is **not** solved by log-transforming the price, all classical models from the previous chapter suffer from the same effect.

Clearly, the Neural Network lacks the ability to capture the entire price range accurately. There are two possible explanations for this phenomenon:

1. The model is flawed.
2. The model is faced with a nearly impossible task.

In order to discriminate the observations with the highest prices from all other listings, the corresponding feature combinations must be separable in the full 59-dimensional feature space. Since this high-dimensional space cannot be visualized, we approximate it with an two-dimensional *embedding* or *latent space*. If the price outliers are clearly separated from the rest in this embedded space, the network is faced with a feasible task. If, however, the outliers are located in the *middle* of the latent distribution, there is little hope to discriminate the most expensive apartments from any of their potentially much lower priced embedded neighbours.

Modern Machine Learning methods provide a large toolbox for low-dimensional embeddings. Since the project is mainly focused on Deep Learning we decided to use a *Variatonal*

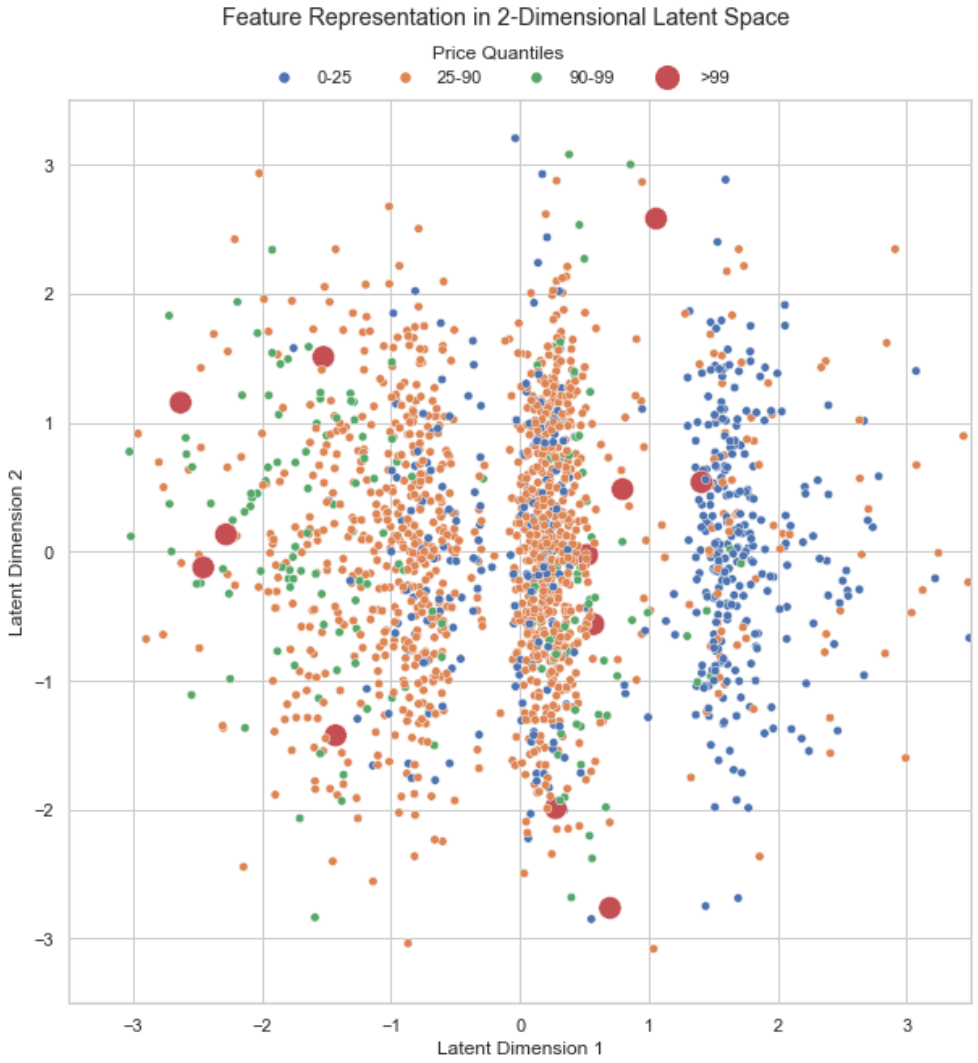

Figure 6: Feature Representation in a two-dimensional latent space embedding

Autoencoder (Kingma and Welling, 2014). In contrast to deterministic Autoencoders the VAE contains an additional loss term apart from the usual reconstruction loss that pulls the encoded latent space distribution towards an isotropic multivariate Gaussian distribution. For this reason, latent space visualizations of the VAE appear to be more spread out and output classes (that are not used for training the VAE) tend to be easier to identify.

Figure 6 visualizes the two-dimensional feature embedding. Already a *single* dimension seems so be sufficient for a general understanding of price segments. While some of the most expensive apartments are located on the far left, surrounded by highly-priced neighbours, the other half shares a similar feature representation to *medium to low* priced listings. Hence, the network likely maps this second half to comparably low price predictions resulting in large residuals with a high influence on the MAE and the R^2 value.

Assuming that the two-dimensional picture appropriately reflects the situation in the full

feature space there are two possible explanations for this lack of separability:

1. The collected data is simply not rich or expressive enough to capture all factors that contribute to very high prices.
2. Some apartments are listed at a price that does not represent their true value.

In the latter case it can be argued that the entire task of predicting Airbnb prices is flawed in the first place: In order to draw connections from features to outputs the process implicitly assumes observations whose listed price can be justified and explained by characteristics of the joint feature distribution.

Moreover, discriminating between these two options is nontrivial. One idea is to use features like availability or the number of reviews to approximate demand and classify price outliers with low demand as overpriced. However, it is entirely possible that an extremely expensive observation with very low demand is listed according to its true value, yet nobody is willing to pay such a high amount for an accomodation, regardless of its quality. As a consequence, a blind removal of such outliers is difficult to justify when obeying correct statistical methodology.

4 Conclusion

In order to predict Airbnb prices in Oslo we constructed our own Deep Neural Network and compared its performance to a collection of well-established Machine Learning algorithms. Since we extracted numeric features out of the unstructured image and text data as part of the preprocessing pipeline, the input data fed into the models was of tabular nature. Thus, classical Machine Learning models and in particular highly intepretable linear models indicated a competitive predictive performance on out-of-sample data.

Further, we emphasized the impact of using different feature set *sizes* both on the training and validation set and used a two-dimensional feature embedding to visualize and explain the sensitivity of the network's predictions with respect to outliers in the price distribution. The lowest *test* set error was achieved by a straightforward ensemble of multiple individual predictive models.

There are various options to extend our work. One particularly appealing idea for *understanding* the Neural Net's behaviour is the construction of *adversarial examples*. In the regression context one could try to leverage steep areas in the high-dimensional loss surface such that small perturbations of each input feature result in exploding price predictions.

Bounding the magnitude of each perturbation by a constant might not be directly transferable to regression tasks in presence of categorical features. As an alternative to the traditional approach borrowed from image classification which is often based on the *Fast Gradient Sign Method* (Goodfellow et al., 2015), a well-behaved Linear Regression model could again be used as a benchmark model. Then, any popular gradient *ascent* algorithm can be used to find feature combinations that maximize the distance between the Linear

Regression prediction and the Neural Net prediction. Since this approach does not naturally bound the input changes, the resulting loss function has to be heavily regularized to prevent pathological solutions.

A Appendix

A.1 Image Processing and Modeling

A.1.1 Webscraping

In the first step, the raw image links provided by the data set had to converted to an image format that the neural network is able to work with.

Therefore we first used the requests library to get the HTML Source Code of each listing's website. Next, the beautifulsoup library served as a convenient HTML parser to find and extract all embedded weblinks that lead to images located on the front page of the listings website. With this strategy we could extract the first 5 images for each apartment (if 5 or more were available) that could be directly accessed from the front page source code.

Finally, we used the `requests` module again in combination with the pillow package to decode the source content of all image adresses into two dimensional images.

A.1.2 Image Transformations

Before feeding these two dimensional images into the model, we performed some further preprocessing steps.

One very common technique when dealing with images is *Data Augmentation*. In contrast to classification tasks however, where Data Augmentation is used to expand the training set and improve simultaneously generalization, this approach is not immediately transferable to a regression context since we have to guarantee that the label (i.e. the price) remains unchanged for each image transformation. Thus, we decided against standard transformations such as rotating the images or manipulating the color composition.

We **did** use image *cropping*, however, which, in our opinion, is one of the few applicable augmentations in regression contexts. After resizing all images to 256×256 pixels we randomly cropped a square area of 224×224 out of each image in the training set and cropped an equally sized area out of the image **center** in the validation set to avoid any randomness during inference.

At a final step all images were normalized, separately for each color channel. In case of the pretrained model explained in the next section we used the same values for normilization that were used during training on the large `ImageNet` database (Russakovsky et al., 2015). The mean values and standard deviations for each color channel are provided in the PyTorch documentation.

A.2 Feature Selection

Not all features out of the new combined and extended feature set are equally valuable to the model in terms of predictive power. In fact, some of the features could not even be transformed to meaningful predictors due to containing e.g. *id* information. Others were

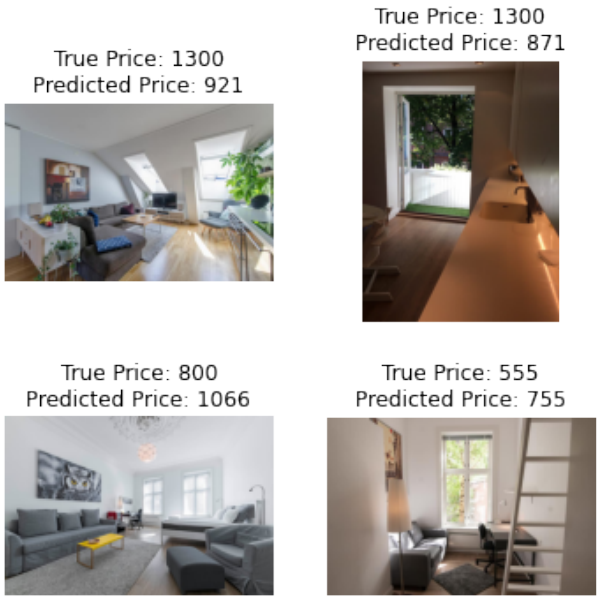

Figure 7: Images from Airbnb Apartments with true and predicted Prices

completely redundant in presence of a combination of original and/or manually constructed features and were thus dropped from the feature set.

Before starting the modeling we intended to reduce the feature set even further to avoid strong correlations among the input predictors and possibly improve generalization performance to out-of-sample data. Thus, we agreed on a two-step strategy.

First, we manually selected features based on three criteria:

1. The variable in consideration and the apartment price must have some connection based on human intuition and background knowledge. For instance, we included variables containing information about the apartment's *size* such as the number of *accomodates*, the number of *bathrooms* and the number of *bedrooms*.

2. There has to exist some correlation with the price in a bivariate visualization, e.g. a *barplot* in case of categorical predictors or a *scatterplot* in case of numeric features.

3. Since our dataset was comparably small with roughly 3000 observations, we had to take care about missing values. Therefore, each variable whose missing values could not be imputed in a meaningful and uncontroversial manner was either selected or dropped based on the trade-off of potential data reduction and its additional predictive value.

Up to this point the feature selection process was solely based on bivariate relationships and self-chosen (arguably arbitrary) selection criteria. There was a high chance of exluding important predictors that shine in combination with other variables rather than on their own.

Therefore, in a second step, we fitted an auxiliary Linear Regression model with *all* available features (except for the trivial ones such as the *picture url*) and analyzed the absolute magnitude of the coefficients. Since all variables are standardized, these magnitudes are within the same range and unit-independent, and can thus be compared.

Of course, even this approach is not perfect, as several highly correlated characteristics that have a strong influence on price could lead to rather small estimated individual coefficients due to their joint predictive/explanatory power. To circumvent this potential issue, we additionally used the *algorithmic* feature selectors provided by the scikit-learn library, which (ideally) are able to separate the effects of highly correlated features and select only a small subset of them.

Thereby we focused on two algorithms:

- Principal Component Analysis (PCA): Reduces dimensionality with the additional benefit of creating *uncorrelated* linear combinations of existing features.
- Recursive Feature Selection (RFE): Selects subset of original features by leveraging an external estimator (in our case a *Support Vector Regressor*). We chose the `RFE` algorithm for the main analysis since the selected features can be immediately interpreted and the performance on the selected feature set was slightly better compared to `PCA`.

The influence of the feature selector on the predictive and particularly the generalization performance is discussed in the *Results* section.

A.3 Price Distribution

One key aspect of exploratory data analysis is investigating the *distribution* of the outcome variable. In our case the price distribution is highly right-skewed with a few very expensive listings pulling the mean and median of the price distribution further away from each other.

Some statistical models such as *Linear Regression* tend to perform better when the outcome distribution is symmetric and approximately normal, whereas some very flexible algorithms like *Neural Networks* do not make any distributional assumptions and are capable of modeling any kind of distribution accurately. Figure 8 illustrates an approximate normal distribution can be achieved with a simple logarithmic distribution.

Whereas *all* of the classical models benefitted from the log-transformation resulting in lower error metrics, this was not the case for the Neural Network we used.

In fact, training turned out to be more challenging, since the *magnitude* of the losses by comparing true and predicted price on the logarithmic scale was drastically reduced, leading to smaller gradients and thus smaller weight updates. This issue could be mitigated to some extent with a larger learning rate. However, in contrast to the untransformed version, the network still suffered from *vanishing gradients* and *loss plateaus*.

Figure 8: Distribution of Apartment Prices on original and logarithmic Scale

A.4 Neural Network

A.4.1 Further (Hyper)-Parameters

Optimizer and Weight Decay
We used the popular `Adam` algorithm (Kingma and Ba, 2017) 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.

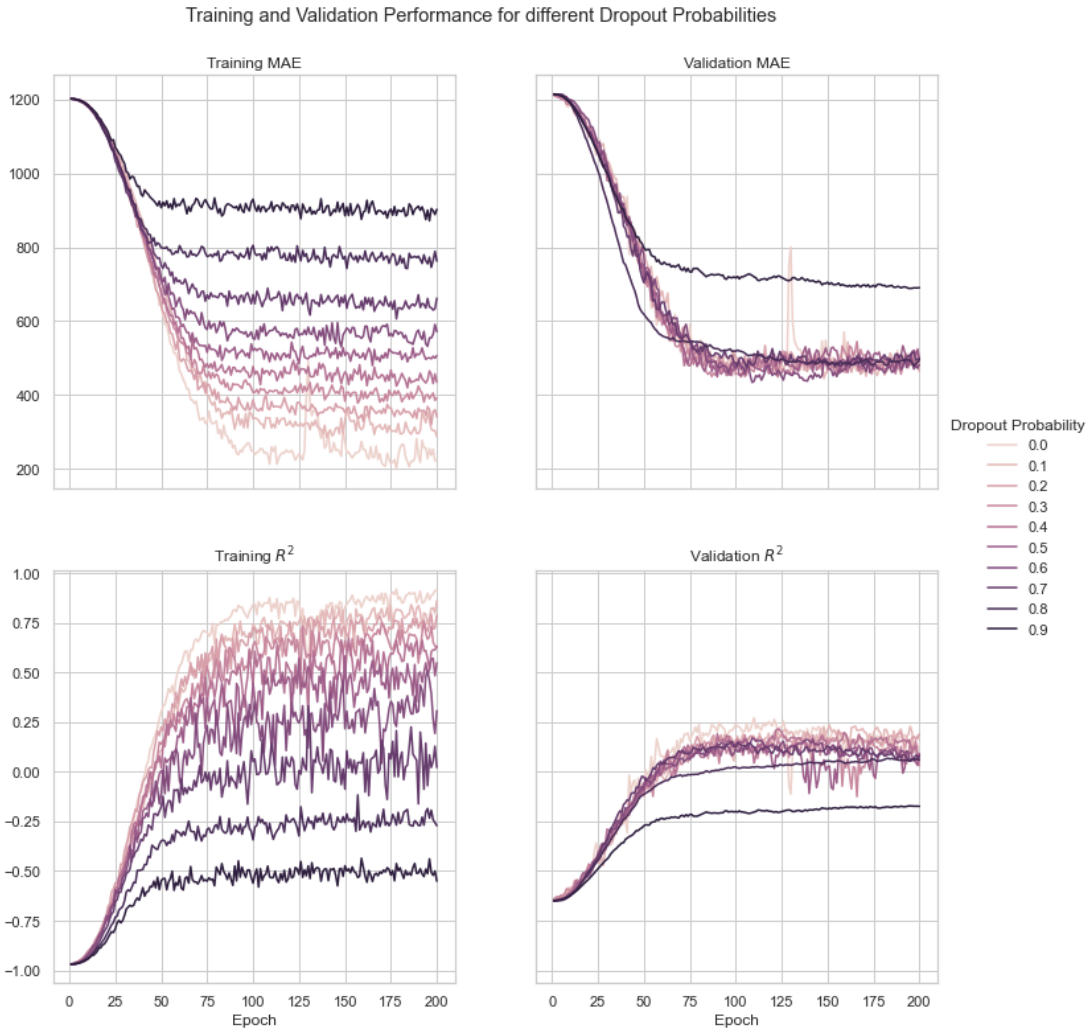

Figure 9: Impact of Dropout Probability on Training and Validation Performance

References

Bergstra, J. and Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.*, 13:281–305.

Chen, T. and Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs].*

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat].*

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs].*

Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs].*

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Kingma, D. P. and Ba, J. (2017). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs].*

Kingma, D. P. and Welling, M. (2014). Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat].*

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *arXiv:1409.0575 [cs].*

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv:1910.01108 [cs].*

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.