Erasmus University Rotterdam
Erasmus School of Economics
Bachelor Thesis Econometrie en Operationele Research

Deep Learning versus Gradient Boosted Decision Trees: A Comparison on Noise Robustness

Ruben Timmer (534559)

Supervisor:	Markus Mueller
Second assessor:	Carlo Cavicchia
Date final version:	2nd July 2023

Abstract

In recent years, deep learning and machine learning models have risen to popularity due to their incredible advancements on complex and unstructured data, such as text and images. However, advancements in the field of tabular data analysis have been relatively limited. One of the challenges surrounding tabular data is the presence of noise in the data. In light of identifying a baseline model that can be used for tabular data, this paper explores the impact of noise on the performance of Multi-Layer Perceptron (MLP), Transformer and Catboost models. Synthetic data with varying levels of noise is generated to evaluate the robustness against noise of the three models. The results reveal that Catboost outperforms MLP and Transformer models, particularly when little to no noise is present. However, this performance gap diminishes as noise increases, indicating that Catboost is less robust against high levels of noise than MLP and Transformer models. Furthermore, a simple out-of-the-box Transformer is shown to perform on-par with a tuned MLP model. However, due to its high computational cost it takes much longer to run than the relatively simple MLP model. The findings highlight the significance of addressing noise in data when utilizing neural networks or gradient-boosted decision tree models. Possible solutions include alternative model specifications and stronger regularisation techniques.

1 Introduction

In the last decade, deep learning (DL) and machine learning (ML) have gained a tremendous amount of attention, especially the models that are used to evaluate images and text. Since these data sources consist of complex and/or unstructured data, deep learning models were able to significantly improve the performance on predicting and classification tasks, up to a human-like level of image recognition and recently also a human-like level of conversational interaction with the rise of ChatGPT and other deep learning chat services (Tuli et al., 2021; Liu et al., 2023). But while these new techniques are slowly being adopted in business (George and George, 2023), the advancements in the field of tabular data have yet to take place. In the past years, there has been a growing interest to identify a baseline model that works well for any tabular data set (Gorishniy et al., 2021, 2022; Grinsztajn et al., 2022).

A good baseline model is of great use, since the data used in most business practices comes in tabular form. As shown with the rise of AI chat services like ChatGPT, businesses are keen on AI-powered applications and can adapt very quickly. It is for this reason that a potential game-changer in the field of tabular data should be studied extensively such that any possible unwanted effects can be minimised. Gorishniy et al. (2021) propose a novel model that uses a Transformer-like structure called Feature-Tokenizer Transformer (FT-Transformer). With the Transformer being a ground-breaking innovation compared to other neural network structures (Vaswani et al., 2017), the FT-Transformer is dubbed a very promising model. Among other DL models, they also test the performance of a vanilla Multi-Layer Perceptron (MLP), which is a relatively simple form of a neural network. Next to neural networks, Gradient-Boosted Decision Tree models (GBDT) such as Catboost and XGBoost (Prokhorenkova et al., 2018; Chen and Guestrin, 2016) have also been shown to achieve high performance (Gorishniy et al., 2021; Grinsztajn et al., 2022). But even though these models have been tested on a variety of data sets, there still is not one model that can consistently outperform the other models.

In the debate of identifying a proper model to use as a baseline for tabular data, all of the previously stated models can have their advantages and disadvantages, and testing these models in various controlled environments helps uncover these characteristics. In this study, we will investigate their ability to handle noise in the data. There are multiple ways for data to contain noise, but the most common cause of noise is measurement error, either from human error, faulty instruments, or limitations in the measurement techniques. Because noise is a very well-known phenomenon, it has been widely studied (Schennach, 2012; Hausman, 2001). As Schennach (2012) state, measurement errors are very common in economic data and can have a big impact on the results of a model.

In an attempt to identify a well-tested baseline model on tabular data, this paper focuses on the effect of noise in the data and tries to answer the following research question: *"How do MLP and Transformer models perform compared to Catboost when noise is added to the data, and what does this say about the generalisation ability of these models?"* Here, the MLP is simply a vanilla neural network, and the Transformer model used is a variation on the proposed FT-Transformer which is used in a later study Gorishniy et al. (2022). In light of identifying a proper tabular data baseline model, we also include a Catboost model and compare the results to see if neural networks are better at handling noise in tabular data than GBDT models. Because we investigate the effect of varying levels of noise in a controlled setting, it is impossible to use real-world data since it is not clear if this data contains any noise in the first place. Instead, we simulate the data using the same Data Generating Process (DGP) used in Gorishniy et al. (2022). More details on this are discussed in Section 3.

In our study we find that the Catboost model is superior in terms of accuracy compared to the MLP and Transformer model. This difference is quite big when little to no noise is present, but seems to diminish when a large amount of noise is present in the data. In general, we find that the performance of all models is strongly affected by noise, albeit large or small levels of noise. We show that the out-of-the-box Transformer model is on-par with the tuned MLP model when compared on their performance, and even performs better when no noise is present in the data. Still, it should be noted that the Transformer model is more computationally expensive than the MLP model. We also find the DL models to overfit more on the data than the Catboost model, and that creating ensembles of DL models significantly improves predictive accuracy. Overall, it can be stated that when using DL or GBDT models one should take the necessary precautions to deal with noise. These include using a modified Instrumental Variable approach from Hausman (2001) and the use of stronger regularisation techniques. Future research could look at the performance of other models such as XGBoost and SAINT (Chen and Guestrin, 2016; Somepalli et al., 2021) when challenged with noise.

We summarize the main contributions of our paper as follows:

- We show that the Catboost, MLP and Transformer model all seem to be impacted by noise, where only a small amount already leads to a substantial decrease in performance.
- We show that Catboost performs better than neural networks when little noise is present, but this advantage seems to diminish when a large amount of noise is present in the data.

- Finally, we show that the tuned MLP model is on-par with the default out-of-the-box Transformer model as proposed by Gorishniy et al. (2021) in terms of performance, but still performs worse when no noise is present.

This paper will proceed as follows: Section 2 discusses an overview of previous literature to provide some context, and in Section 3 we will describe the synthetic data generation process. Then the methods and models used in this study are discussed in Section 4, followed by all results and general inferences in Section 5. Finally, Section 6 contains a conclusion and discussion for future research.

2 Literature Review

Since the big advancements of Deep Learning (DL) models in the textual and visual domain, there has been a growing interest in DL models designed for tabular data (Klambauer et al., 2017; Popov et al., 2019; Arik and Pfister, 2021; Song et al., 2019; Wang et al., 2021; Badirli et al., 2020; Hazimeh et al., 2020; Huang et al., 2020; Somepalli et al., 2021; Kossen et al., 2021). These studies show that there is definitely potential for neural networks in the field of tabular data. At the same time, there is also a growing interest in tree-based models. Specifically, Gradient Boosted Decision Trees (GBDT) have developed a status of fast and effective methods to use when dealing with large amounts of tabular data, the so called "Big Data" that is a common term in business practices these days. Among others, the XGBoost and Catboost model show promising results (Chen and Guestrin, 2016; Prokhorenkova et al., 2018).

2.1 Deep learning vs GBDT

Gorishniy et al. (2021) and Grinsztajn et al. (2022) performed two very similar studies in which they test a range of DL and GBDT models on tabular data sets ranging from a couple of thousand to more than a million observations in an effort to identify a baseline model that works well for any tabular data set. Gorishniy et al. (2021) conclude that their own proposed model, the FT-Transformer, achieves the best results. This model is similar to the AutoInt model used for Click-Through-Rate (CTR) prediction (Song et al., 2019). They show that the FT-Transformer model outperforms all other DL and GBDT models. Indeed, this model has since been used for downstream tasks in a variety of sectors (Levin et al., 2022; Dang et al., 2022; Potts and Leontidis, 2023). But Grinsztajn et al. (2022) come to a different conclusion and state that GBDT models such as XGBoost perform better in almost all occasions. In an empirical investigation, they find that GBDT models work better on tabular data in general because neural network models are more biased towards smooth target functions and are more affected by uninformative features. To some extent, this is in contrast to the findings of Gorishniy et al. (2021). Thus, it can be stated that there is a disagreement on a proper baseline model.

In a later study, Gorishniy et al. (2022) state that the embedding structure of neural networks is underexplored. In the context of numerical features, an embedding layer is used to convert the features into a higher dimensional representation. This essentially creates more context as to how certain features relate to each other, and has been shown to improve the performance of neural networks (Arik and Pfister, 2021). Gorishniy et al. (2022) use multiple different embedding schemes and conclude that it is always beneficial to swap the single embedding layer for an embedding scheme as proposed in their paper. More details on the embedding schemes are discussed in Section 4. For the MLP model, when combined with an embedding scheme, the performance improves the most and is on-par with the Transformer model (where previously the difference in performance was quite large). Because there is big difference in training and tuning times for the Transformer and MLP models, both are considered in this study. Based on their investigation we combine the MLP and Transformer models with a piecewise linear embedding module. More details on this will be discussed in Section 4. Though, while these models perform better with the new embedding modules, the Catboost model still achieves comparable performance on most data sets. Because of its design, the Catboost model is specifically well-suited for categorical features, but even on data sets containing only numerical features it often achieves better prediction scores than the XGBoost model. Therefore, the Catboost model is also considered in this study.

2.2 When data becomes fuzzy

It is well-known that databases used in business practices can contain measurement errors, often referred to as noise (Klein and Rossin, 1999; Schennach, 2012; Bansal et al., 1993). As Schennach (2012) state, measurement errors can have a big impact on the prediction accuracy of a non-linear model. While there is an easy way of dealing with unknown noise for linear models, such as using Instrumental Variables (IV), it is not so easy to do that for non-linear models. Hausman (2001) describe a non-linear specification of the IV, but this is not a straightforward approach. While it might not be easy to deal with noise by changing the model definition, neural networks have still been shown to handle noise better than linear models. Bansal et al. (1993) compare neural networks to linear regression models and conclude that neural networks are more robust against noise than linear regression models. Building upon that, Klein and Rossin (1999) construct data where the fraction of the data containing noise and the magnitude of the noise in the data vary. They then investigate the effects of these two aspects of noise on the predictive performance of neural networks, and find that both the fraction of the data containing noise as well as the magnitude of the noise have an impact on the performance. But one could question if this result is still relevant, as neural networks have come a long way since then.

In this paper, we essentially try to do the same. We keep the fraction of data that contains noise the same (namely, all data points are contaminated with noise). This is done to magnify the effect noise has on the performance of the models. But we do investigate the influence of the magnitude of the noise on the performance of the MLP, Transformer and Catboost models. In particular we are interested if the high performance of the models maintains when the data quality worsens, and if any neural network is significantly more robust against noise than the other models.

3 Data

3.1 Synthetic Data Generation

Because we are interested solely in the effect of noise on the performance of the model, it is only fair to analyse this via a controlled experiment. Thus, it is impossible to use any real-world data sets since one cannot know if the data used contains no noise at all. If there would be noise in the first place, it could influence the results which would disrupt the controllability of the experiment. To this end, we will make use of synthetic data which is generated using 16 randomly initialised decision trees. This creates a very complex but still deterministic relation between explanatory variables and the target variable. First, the explanatory variables will be sampled from a standard normal distribution:

$$\boldsymbol{x}_j^* \sim N(0, \boldsymbol{I}). \tag{1}$$

Here, $\boldsymbol{x}_j^*$ corresponds to the variable j. This sampling is done for 10 independently drawn variables, all consisting of 30.000 instances. The noise is added by generating another series of variables in the same way as before:

$$\boldsymbol{w}_j \sim N(0, \boldsymbol{I}). \tag{2}$$

Every vector $\boldsymbol{w}_j$ serves as a vector of noise that will be added to explanatory variable $\boldsymbol{x}_j^*$. For a given noise level v, the final explanatory variables $\boldsymbol{x}_j$ which are used to construct the data sets are defined as

$$\boldsymbol{x}_j = \boldsymbol{x}_j^* + v\boldsymbol{w}_j \tag{3}$$

Here, the scaling factor v can be viewed as the standard deviation of the noise. To the best of our knowledge, there has not been a proper study on the magnitude of measurement error in tabular data. Thus, the levels of noise used in this study are more of an educated guess of what could be the levels of noise in a real-world example. For v, we take the following values: 0.00, 0.01, 0.02, 0.05, 0.10, 0.20 and 0.50. We denote the corresponding data sets of explanatory variables with $\boldsymbol{X}_v$. In total, we obtain seven data sets: $\boldsymbol{X}_{0.00}$, $\boldsymbol{X}_{0.01}$, $\boldsymbol{X}_{0.02}$, $\boldsymbol{X}_{0.05}$, $\boldsymbol{X}_{0.10}$, $\boldsymbol{X}_{0.20}$ and $\boldsymbol{X}_{0.50}$.

To generate target variables, we use only the "clean" data set $\boldsymbol{X}_{0.00}$ as input. We construct 16 independent randomly initialised decision trees, which act as separate complex functions f_k. The final target variable is then obtained by taking the average of the independently generated $\boldsymbol{y}_k$:

$$\begin{aligned} \boldsymbol{y}_k &= f_k(\boldsymbol{X}_{0.00}), \text{ for } k \in \{1, ..., 16\} \\ \boldsymbol{y} &= \sum_{k=1}^{16} \boldsymbol{y}_k \end{aligned} \tag{4}$$

The final data sets thus have a clear deterministic relation between $\boldsymbol{X}_v$ and $\boldsymbol{y}$ when v is zero, but this relation becomes more fuzzy when v increases. The generated data sets $\boldsymbol{X}_v$ all have a mean of approximately zero, and a variance that is approximately equal to $1 + v^2$. More details

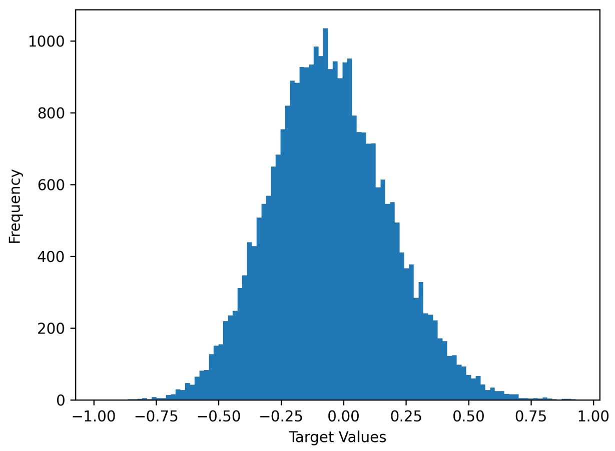


Figure 1: Distribution of synthetically generated target values

on the randomly initialized decision trees and the generated data sets can be found in Appendix A. Figure 1 shows a histogram of the synthetically generated target values. The histogram does not entirely match a Gaussian Distribution, but keep in mind that these values are just the average output of 16 randomly initialised decision trees.

When any tabular data is used for DL of GBDT models, it is common to transform it to follow a distribution that approximates the standard normal distribution. To adhere to the common practices, we also transform the data using quantile transformation before feeding it in the models.

3.2 Replication study

For this research, we use the code that was made available by Gorishniy et al. (2022) to generate the data and perform the tuning, training and testing of all models. To show that the implemented models work the same as in their paper, we conduct a replication study as well. In here, we use the MLP and FT-Transformer models as defined in their first study (Gorishniy et al., 2021), so without the newly implemented embedding modules of their second study. The data used in this replication study is the California Housing data set. This data set, first used by (Pace and Barry, 1997), is widely used in the field of machine learning and it contains real estate data. It consists of 20.640 observations of 8 numerical variables and concerns a regression task on one target variable. Due to its widespread use, it has been made available in the `sklearn` package, commonly used for simple DL implementations.

4 Methodology

In this section, we discuss various models and methods used to create the final results. First, the MLP, (FT-)Transformer and Catboost model are discussed, followed by an explanation of the embedding module from Gorishniy et al. (2022) that is used in combination with the DL models. Lastly, the implementation details will be discussed.

4.1 MLP

The vanilla neural network, also known as a Multi-Layer Perceptron model, proved to be a fast and effective way to achieve good performance when combined with certain embedding modules (Gorishniy et al., 2022). As a neural network, it consists interconnected neurons containing weights. Signals pass through these neurons, and are altered in every neuron by multiplying it with its weight. In addition, every layer contains a weight that is added separately as a bias. This weight is not connected with previous neurons and improves the model's flexibility. After multiplication with all the weights, the signal passes through an activation layer. This layer outputs the signals after they have been passed through an activation function. In this paper, we will use the ReLU (Rectified Linear Unit) function, which only passes the signal through if it is positive. This simple yet effective function introduces non-linearity and enables the neural network to learn complex non-linear relations in the data. After the activation layer, the signals are passed through a dropout layer. As Srivastava (2013) have shown, randomly dropping neurons from the neural network when training prevents it from overfitting too much on the training data, since the network now focuses more on general and robust features in the data. The use of a dropout layer has since become standard practice in neural networks. The MLP used in Gorishniy et al. (2022) consists of the following sequential structure:

$$\begin{aligned} \mathrm{MLP}(\boldsymbol{z}^{con}) &= \mathrm{Linear}(\mathrm{MLPBlock}(...(\mathrm{MLPBlock}(\boldsymbol{z}^{con})))) \\ \mathrm{MLPBlock}(\boldsymbol{z}^{con}) &= \mathrm{Dropout}(\mathrm{ReLU}(\mathrm{Linear}(\boldsymbol{z}^{con}))) \\ \boldsymbol{z}^{con} &= \mathrm{concat}[\boldsymbol{z}_1, ..., \boldsymbol{z}_k] \in \mathbb{R}^{kd_z} \end{aligned} \tag{5}$$

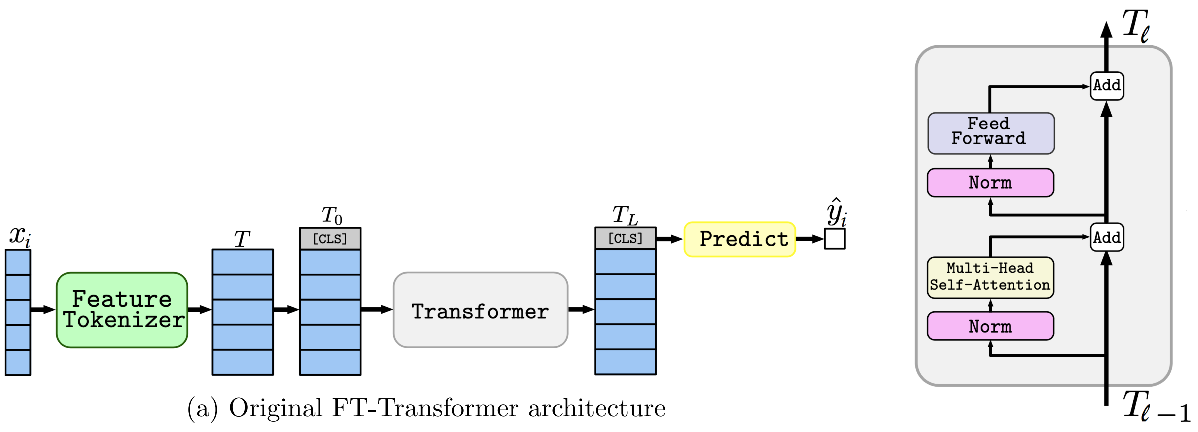
Here, $\boldsymbol{z}_k$ is the embedding vector of the k-th numerical feature and d_z is the dimension of embedding vectors. For the synthetically generated data the total number of features $k = 10$. The details on the embedding vectors are explained in Section 4.3. For the MLP model, the embedding vectors of all features must be concatenated to one long vector $\boldsymbol{z}^{con}$ in order to be passed to the model. This simply creates one long vector of all embedding vectors. The number of layers (MLPBlocks) in the neural network also have a big impact on its performance. While a network with more layers can capture more complex relationships in the data, they also become more prone to overfitting. When training, the model is optimised using backpropagation. This is an iterative process in which the weights in the neurons are adjusted slightly based on how close the prediction of the network is to the ground truth value. Because of this property, neural networks fall under supervised learning methods. Since backpropagation significantly increases training times, it is common practice to process the data in batches, rather than per individual observation. The predictions of all observations in the batch are first computed, after which the network calculates the vector of loss values for the entire batch. Then the aggregated loss value is used to update the network. Thus, the network does not update its neurons based on each observation, which enables the network to learn more generalized patterns and reduce the risk of overfitting.

4.2 (Feature-Tokenizer) Transformer

The Feature-Tokenizer Transformer, also known as FT-Transformer, was first proposed by Gorishniy et al. (2021) and is based on the original Transformer model as proposed by Vaswani et al. (2017). Figure 2a shows a schematic overview of the FT-Transformer model. Since the Feature-Tokenizer module at the beginning will be replaced by a different embedding module in our paper, we won't focus on that. But the output of the embedding module is essentially the same: a stack T of numerical feature embedding vectors $\boldsymbol{z}_1, ..., \boldsymbol{z}_k$ that expresses the input features of observation x_i in a higher dimension. The details on this are covered in Section 4.3.

4.2.1 Self-Attention

For the Transformer itself, the feature embedding stack T is appended with a classification token (CLS), which is used for the final prediction later on. The resulting stack T_0 is then passed as input for the first Transformer layer. A Transformer is differs from a normal neural network, since it uses self-attention. Self-attention takes the stack of feature embedding vectors T_0 and computes a new stack of feature vectors that contain information about its relation with other features. It accomplishes this by making use of attention weights. For every feature (and the CLS token), the attention weights are calculated using the similarity between that feature and another feature. Song et al. (2019) provide more details on the calculation of attention weights. Once calculated, these attention weights are used to generate a new representation of each feature that is inspired by the context of all other features as well, thus enabling the model to focus on relevant information and learn complex relations between features.



(a) Original FT-Transformer architecture

(b) One Transformer layer

Figure 2: FT-Transformer architecture and a Transformer layer (source: Gorishniy et al. (2021))

4.2.2 Multi-Head Self-Attention

The whole process of self-attention can be seen as a black-box: a stack of feature vectors gets transformed to a new stack of feature vectors, where the newly created vectors contain information about the relations between the features. Because there can be different relations and dependencies between the features, the Transformer layer uses multiple self-attention blocks simultaneously. This is called Multi-Head Self-Attention (MHSA). Here, every head serves as a self-attention block that can focus on a specific set of feature relations. In the field of textual data, where Transformer models are commonly used, this has a very intuitive explanation: one block could be related to grammar, the other to gender and a third block can be related to plurality. For tabular data this is less intuitive, but the idea still holds: learn about different high-order relations simultaneously. The newly created feature vectors of all heads are then concatened and processed by a linear layer to create a final stack of feature representations containing high-order relations.

Figure 2b shows how one Transformation layer looks like. After passing through a MHSA layer, the output is combined with the original stack of feature vectors via a residual connection. This is done to preserve previously learned relations between features. This is then used as input for a Feed-Forward Network (FFN), which is basically a MLP. This FFN consists of a linear layer, followed by an attention and dropout layer, and finally a second linear layer. The final output is again created by combining the output of the FFN with a residual connection. Before passing the stack of feature vectors to the MHSA and FNN blocks, the vectors are also normalised (PreNorm, Wang et al. (2019)) to achieve better performance. This is visualised in Figure 2b as the pink 'Norm' blocks. One Transformer layer converts a stack of feature vectors T_{l-1} to a new stack of feature vectors T_l, also changing the CLS token. In the Transformer model, a total of L layers are connected sequentially as shown in Figure 2a. As for MLP models, the Transformer is able to learn more complex behaviour when more layers are added. But this too comes with the risk of overfitting. Thus, the number of layers must be tuned or chosen carefully. Finally, the CLS token of the last Transformer layer T_L is used for the prediction of the target variable for observation i:

$$\hat{y}_i = \text{Linear}(\text{ReLU}(\text{LayerNorm}(T_L^{[CLS]}))) \tag{6}$$

Here, the LayerNorm is a normalisation layer that normalises the CLS token. The output of the normalisation layer then passes through an activation and linear layer, before being converted to a single prediction $\hat{y}_i$. The final prediction $\hat{y}_i$ can then be compared to the ground truth value y_i for backpropagation. Although Transformer models have been shown to be very accurate, they lack the ability to scale to large data sets containing many features. The MHSA modules has a quadratic complexity with respect to the number of features in the data, which makes it inefficient when dealing with these large data sets (Wang et al., 2020).

4.3 Embedding module

In neural networks, embeddings are commonly used to transform data before feeding it in the model itself. For example, when dealing with textual data it is common to pass the word tokens through an embedding layer that converts them into vectors. These alternative representations can then contain additional information about the similarity of words, which greatly helps the backbone structure of the networks to learn about important features. For tabular data, embedding structures have been used to convert categorical data into numerical data. In the TabTransformer model for example, categorical features are passed through a number of Transformer layers before being passed to a MLP (Huang et al., 2020). But Gorishniy et al. (2021) take this even a step further and also convert numerical features to embedding vectors using their Feature-Tokenizer. They do this such that the embeddings of the numerical features can be passed to the Transformer layers together with the categorical features. This approach has been shown to work very well and improve predictions significantly.

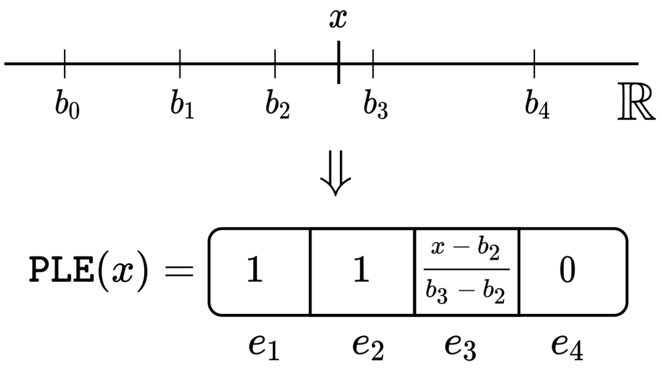


Figure 3: Graphic example of the piecewise linear encoding scheme, defined in equation 7 (source: Gorishniy et al. (2022))

4.3.1 Piecewise Linear Encoding

Interestingly, Gorishniy et al. (2022) find that converting the numerical features to embedding vectors not only helps Transformer models, but also other structures like MLP. In their study, they investigate the effect of different embedding structures on the performance of neural networks. They find that combining quantile feature binning with Piecewise Linear Encoding (PLE) to alter the input for the neural networks achieves higher performance on the benchmark. Analogous to one-hot encoding, where categorical features are transformed in numerical representations, quantile feature binning combined with PLE transforms numerical features into alternative numerical representations (See Figure 3 for a graphic example). Similar to one-hot-encoding, this method makes a trade-off between efficiency and expressivity.

To transform the numerical input into alternative representations, the numerical features are first split up in bins using uniformly chosen empirical quantiles of the corresponding distribution of each feature. After splitting the features up in T bins, the numeric values are transformed to the following encoding scheme:

$$\boldsymbol{z}_k = \text{PLE}(x) = [e_1, ..., e_T] \in \mathbb{R}^T$$
$$e_t = \begin{cases} 0, & \text{if } x < b_{t-1} \text{ and } t > 1 \\ 1, & \text{if } x \geq b_t \text{ and } t < T \\ \frac{x-b_{t-1}}{b_t-b_{t-1}} & \text{otherwise} \end{cases} \tag{7}$$

Here, $\boldsymbol{z}_k$ is the embedding vector of feature value x and b_t corresponds to the upper boundary of bin t. Essentially, for the encoding scheme, the boundary values of the bins are used to create an alternative numeric representation in the form of an array instead of a single value. If a value of a numeric feature falls in bin t, the $(t-1)$ preceding elements of the PLE representation of this value are all equal to 1, and all elements after position t are equal to 0. A graphic example with four bins is shown in Figure 3. For MLP-like neural networks, the resulting PLE representations must be concatenated to one flat vector in order for it to be processed by the neural network.

Note that right now, all features are converted to the same amount of bins. As Gorishniy et al. (2022) state, this might not be optimal as numerical features can still differ quite a bit in their distribution and characteristics. It is also important to note that the parameters of the embedding structures for different features are not shared, meaning that each feature gets embedded independently from other features. On top of the PLE layer, we add two differentiable components; first a linear layer (with bias) and then a ReLU layer, both of which are conventional differential layers. Also these layers are specific to each individual feature, meaning that they are not shared between features. This embedding module combined with the neural networks achieved one of the best performances, while being quite simple and easy to implement.

4.4 Catboost

The last model that is used in this paper is the Catboost model. This Gradient Boosted Decision Tree model (GBDT) was first proposed by Prokhorenkova et al. (2018) and it makes use of an effective way for creating ensembles of so-called 'weak learners'. It sequentially builds decision trees using the errors from the preceding tree, and for each new tree it tries to find the best splits to split the data on based on an ordering in the numerical features. Thus, it can capture complex relationships and interactions within the numerical features. Basically, the ensembles are created following a process of gradient descent based on the error of newly constructed trees. The Catboost model does this using efficient estimation methods, which speeds up the process. While other GBDT models like XGBoost also create sequences of trees in a similar manner, Catboost has been shown to be superior in terms of speed and accuracy. For more details on Catboost, we refer to Prokhorenkova et al. (2018).

4.5 Implementation details

In this subsection, all relevant details for the implementation are listed. Most of these remarks are similar to the implementation details as stated in Gorishniy et al. (2022), but for the sake of completeness the details are also given here.

In the field of neural networks, it is common practice to standardise the data when pre-processing. While the synthetically generated data did not exhibit any unusual patterns, the data was still pre-processed using quantile transformation from the Scikit-learn library (Pedregosa et al., 2011) to adhere to common practices. In addition, the targets are also standardised (meaning that the mean is subtracted and the targets are scaled with their standard deviation). For the tuning, training and evaluation, the data sets are all split up in the same way. For the train/validation/test set, two-third of the data is used for training (20.000 observations) and of the last part, two-fifth is used for validation (4.000 observations) while the remaining part is used for final evaluation (6.000 observations).

For tuning, we follow the same procedure as Gorishniy et al. (2022). In search for the best hyperparameters, we use the Optuna library from Akiba et al. (2019) where we let the Optuna program run for 40 trials per model-data set combination. It is important to note that due to limited resources, we are only capable of running the tuning and final training on CPU (8th Generation Intel Core i5). This poses a big limitation in terms of tuning, and thus we are only able to tune the Catboost and MLP model. For the MLP model, the tuning space was changed compared to Gorishniy et al. (2022). For the FT-Transformer model, we use the default settings as proposed by Gorishniy et al. (2021) with a slight modification to the learning rate during training. Previously, it was set to 0.0001, but we find 0.001 to work better for the data. In section 5, we discuss more details surrounding tuning/training times and model sizes. The hyperparameter tuning spaces for Catboost and MLP, as well as the final model parameters for all three models can be found in Appendix B.

After tuning on the validation set for the MLP and Catboost models, the best hyperparameter configurations are used for training on 15 different random seeds. For the FT-Transformer model, the default parameters are used to train on 15 different random seeds as well. After obtaining 15 separately trained versions of each model, we create 3 ensembles consisting of 5 models. For each ensemble, the individual predictions of the 5 models are averaged. Finally the Root Mean Squared Errors (RMSE) of all ensembles are calculated using predictions $\hat{y}_i$ and ground-truth values y_i, according to the following formula:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}}. \tag{8}$$

The implementation of the MLP, FT-Transformer and Catboost model are all taken from Gorishniy et al. (2022). For the optimisation of the models we make use of the AdamW optimizer (Loshchilov and Hutter, 2017). The batch size for all data sets is set to 512 and the performance of the models during tuning is evaluated against the validation set. We train until there are `patience + 1` consecutive epochs without improvements on the validation set. We set `patience = 10` for all models. For the tuning of the MLP model, we also tune the number of bins used in the embedding module.

5 Results

In this section, we will discuss the results. First, the results of the replication study are discussed. Then we will discuss the results of the MLP, Transformer and Catboost model on the test sets, as well as on the training sets, to draw conclusions about the relative performance of the models and if they are overfitting on the data. We then discuss the results of the ensembles of the models and finish with an overview of the total tuning and training times, as well as a comparison of the model sizes.

5.1 Replication

We start with the results of the replication study. In table 1, the results on the California Housing data set are summarised. For this replication study, only the MLP and FT-Transformer models are considered (without the new embedding schemes). The reported RMSE values are the results on the test set. It is clear that the results are nearly identical to those of the original study, which is as expected. Do note that the results of the replication study are obtained by running one model only, whereas the results of the original study are obtained by taking the average RMSE value of 15 models trained using different seeds.

Table 1: Results of the replication study. The results of the original study are taken directly from Gorishniy et al. (2021)

	Original study	Replication
MLP	0.499	0.496
FT-Transformer	0.459	0.460

5.2 Performance of the models after tuning

We now look at the performance of the models after tuning the models to find the optimal hyperparameters. As stated previously, this is only done for the Catboost and MLP model, since the Transformer model was computationally too expensive to tune. The tuning is done using the Optuna library, and we run 40 trials for each of the models that are tuned. This means that for each model, 40 different configurations of hyperparameters are tested using the RMSE as a performance metric. The results are shown in Table 2. Note that the all reported RMSE values are the average of 15 models trained on different seeds. Additionally, the results are shown in the graph in Figure 4. We summarise our observations:

- Most importantly, we observe that all models seem to perform increasingly worse for higher levels of noise. This is not only the case for high levels of noise - even when the added noise has a standard deviation that is equal to 5% of the standard deviation of the data, the RMSE already doubles for all models.
- It is immediately clear that the Catboost model outperforms both the MLP and the Transformer model. While it consistently scores the best for every noise level, we do notice that this advantage seems to diminish for higher noise levels. At a level of 0.5 the models score more or less the same. But at these high levels of noise, the small differences in performance can be neglected, as all models perform much worse with very noisy data.
- From Table 2, it is visible that the proportional increase in RMSE value of the Catboost model is much higher than those of the neural networks, especially for higher levels of noise. Thus, it can be stated that the Catboost model is the most affected by noise in the data.
- When no noise is present in the data, the default Transformer works quite well compared to the MLP model. But after adding a little noise this advantage seems to vanish. For all data sets containing noise, the MLP and Transformer model seem to perform almost on-par, and we cannot identify any of the two models as better in terms of handling noise. Do note that we compare the default Transformer against a tuned version of the MLP model.

Table 2: Results on the test set after tuning. The reported values are the RMSE values averaged over the 15 models trained on different seeds. Behind every RMSE value, we also report the proportional increase between that result and the result of the data set without noise (column 0.00).

	Noise Level						
Model	0.00	0.01	0.02	0.05	0.1	0.2	0.5
MLP	0.0414	0.0489 (×1.18)	0.0580 (×1.40)	0.0786 (×1.90)	0.0995 (×2.40)	0.1290 (×3.11)	0.1791 (×4.32)
Transformer	0.0339	0.0465 (×1.37)	0.0572 (×1.69)	0.0783 (×2.31)	0.1007 (×2.97)	0.1312 (×3.87)	0.1796 (×5.30)
Catboost	0.0247	0.0345 (×1.39)	0.0444 (×1.80)	0.0661 (×2.67)	0.0904 (×3.65)	0.1230 (×4.97)	0.1761 (×7.12)

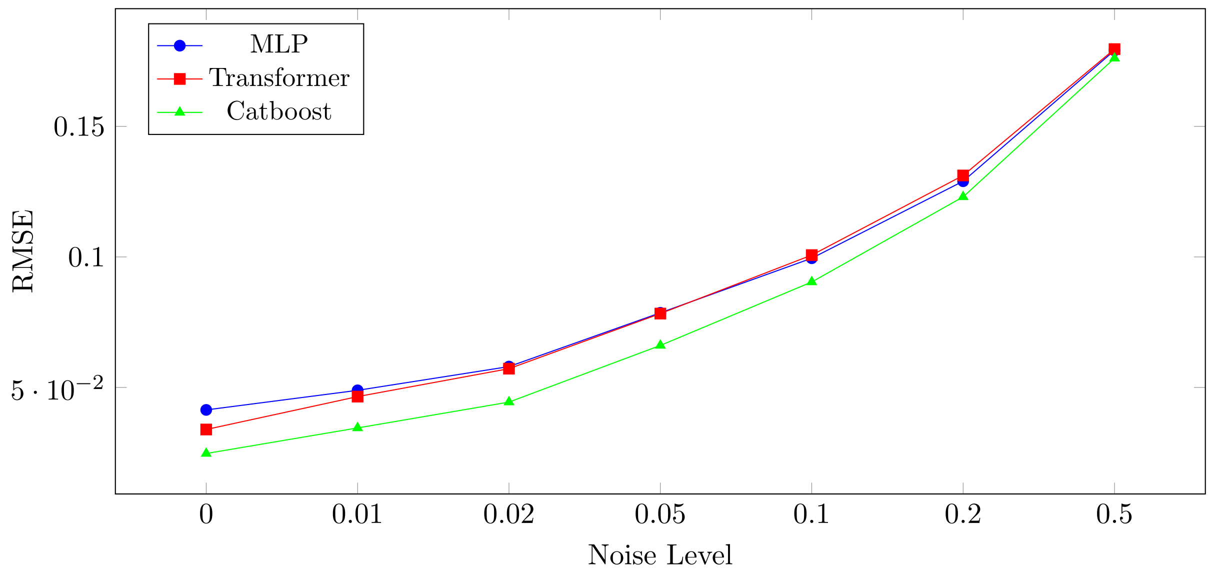


Figure 4: Graph of the RMSE values for each model after tuning. For the Transformer, the default model is used. All reported RMSE values are the average of 15 models trained of different seeds. Note that the distances between noise levels on the bottom axis are not scaled according to the level of noise.

5.3 Overfitting

When training the models, we observed that there were large differences between training and test RMSE values. Table 3 shows the results on the training set of all models, given as a percentage of the RMSE value of the same model on the test set. Thus, it is immediately clear how the performances on the train and test set compare to each other. It is normal to see models perform better on the training set than on unseen data, since they are adjusted based on that training data. But in general, the difference should not be that big.

It is clear from Table 3 that the MLP and Transformer models perform much better on the training data than on the test data. This is also the case for the Catboost model, but the differences are not as big as for the neural networks. This indicates that the neural networks could be overfitting on the training data. Especially for the lower noise levels the difference in RMSE values is quite large, to the point where the training set RMSE is four times lower than the RMSE value for the test set. For higher levels of noise we see that in general the difference is not very big, but this only logical since it is harder to predict and overfit on noisy data.

A possible solution to this overfitting problem could be to consider stonger regularisation options. For example, one could try out higher dropout proportions and weight decay for the neural network structures. This could result in models that generalise better to unseen data, at the cost of training efficiency. Additionally, one could invest more time and resources in tuning the models. Now, only the default Transformer hyperparameters as stated in Gorishniy et al. (2021) were used and the the tuning space for the MLP model was a bit limited compared to previous research Gorishniy et al. (2021). Lastly, one could use more data to train and evaluate the models. Due to computational constraints we used a data set containing of 30.000 observations, of which 20.000 were used for training. This could of course be the cause of the large difference in training and test RMSE, in which case the use of more data could be the solution.

Table 3: Results on the training set of every model, given as a percentage of the RMSE value of the same model on the test set. The absolute RMSE values on the training set are stated in brackets behind the percentages. All reported RMSE values are the average of 15 models trained of different seeds.

	Noise Level						
Model	0.00	0.01	0.02	0.05	0.1	0.2	0.5
MLP	24.0% (0.0099)	43.5% (0.0213)	47.9% (0.0277)	67.8% (0.0533)	79.9% (0.0795)	87.5% (0.1129)	94.8% (0.1697)
Transformer	63.0% (0.0214)	59.1% (0.0275)	62.3% (0.0356)	70.4% (0.0551)	80.6% (0.0811)	87.6% (0.1150)	96.4% (0.1730)
Catboost	74.0% (0.0183)	66.0% (0.0228)	73.9% (0.0328)	85.4% (0.0564)	87.8% (0.0794)	87.3% (0.1074)	90.2% (0.1589)

5.4 Making ensembles

For every model-data set combination, we use the 15 models trained on different seeds to create 3 ensembles of 5 models each. The predictions of each ensemble is computed by taking the average of the predictions from the individual models. The results are summarised in Table 4. The RMSE values stated in the table are the average RMSE value of the three ensembles for that model on the test set. Behind the RMSE values is the percentage reduction compared to the average RMSE values of the single models in Table 2.

The most important finding is that for the neural networks the ensemble technique greatly improves predictions, and can reduce the RMSE by almost 18 percent in some cases. For Catboost, the ensemble technique does not seem to help that much, but this can of course be dedicated to the fact that GBDT already makes use of an ensembling technique whereas the single neural networks are not using any ensembling. From Table 4 it is also clear that the ensembles offer a bigger advantage when little noise is present. As for the results of the single models, this is an indication that the higher levels of noise contain so much noise that making predictions is hard to do. For the noise levels 0.2 and 0.5 it seems that all models work bad, and the small differences in performance are not that relevant anymore.

Overall it can be stated that ensembling is a very useful technique to boost performances without the need to modify the underlying models. Especially the DL models benefit from ensembling, and we advise anyone using MLP or Transformer models to use this approach if they have the time and resources.

Table 4: Results of the ensembles on the test set. The reported RMSE values are the average of the three ensembles. Behind every RMSE value, we also report the percentage reduction compared to the RMSE value of the single models in Table 2.

	Noise Level						
Model	0.00	0.01	0.02	0.05	0.1	0.2	0.5
MLP	0.0364 (-12.14%)	0.0429 (-12.27%)	0.0529 (-8.76%)	0.0736 (-6.32%)	0.0957 (-3.80%)	0.1265 (-1.87%)	0.1774 (-0.95%)
Transformer	0.0278 (-17.92%)	0.0408 (-12.16%)	0.0512 (-10.43%)	0.0725 (-7.39%)	0.0960 (-4.63%)	0.1272 (-3.04%)	0.1778 (-1.00%)
Catboost	0.0240 (-2.82%)	0.0338 (-1.88%)	0.0438 (-1.40%)	0.0658 (-0.47%)	0.0901 (-0.39%)	0.1227 (-0.27%)	0.1760 (-0.10%)

5.5 Tuning times and model size

Of course, the accuracy of the models is not the only thing that plays a role in the discussion of which model performs best. It is important to also take tuning times and model sizes into account, since not everyone has access to fast processors. For this study too, the tuning as well as the training is limited by the limited access to proper resources. Since the only available processor is 1 CPU (8th Generation Intel Core i5), it is impossible to fully tune all the models like Gorishniy et al. (2022) do. But since the Catboost model is quite fast compared to the neural networks, we do use the same tuning space for that model as in their paper. For the MLP model we choose a smaller tuning space, which can be found in Appendix B. Because the tuning of only 1 hyperparameter configuration of the Transformer already takes more 40 minutes, it is impossible to tune this model for all datasets. Thus, only the default model as proposed by Gorishniy et al. (2021) is used.

In Table 5 summarises the average training and tuning times, as well as the model sizes of the neural networks. For the tuning times, we report the average tuning time over all data sets. The training time per model is taken as the average training time over all data sets as well. The total tuning of the Catboost model for 40 Optuna trials on one data set takes on average 15 minutes on the CPU. This is around 8 times as fast as the MLP model which takes on average almost 2 hours to finish tuning on one data set. The average training time of one MLP model also compares similarly to the average training time of one Catboost model. Here, we also see the enormous difference in training times between the Transformer and the other two models; training one MLP model is more than 20 times faster than one Transformer model, and one Catboost model is trained 200 times faster than the Transformer model. These differences in training and tuning times should also be taken into account when one considers using any of these models. Part of the longer training times of the Transformer model can be dedicated to the fact that it has twice as many parameters that need to be tweaked. This combined with the quadratic complexity of the MHSA modules in the Transformer with respect to the number of features explaines partly why the Transformer model takes so much longer to train.

Table 5: Total tuning time, training time for one model and model sizes.

	Total tuning time	Training time	Model size
MLP	01:57:51	00:01:55	724663
Transformer	-	00:41:09	1385281
Catboost	00:15:00	00:00:12	-

6 Conclusion and Discussion

In this work we discuss the robustness of various models when it comes to handling noise in the data. We test this for various levels of noise, and see how the performance of the models worsens as the level of noise increases. In light of identifying a good baseline model to use, it is hard to come to a clear conclusion. While the Catboost model shows superior performance, it must be noted that the data used in this study was generated using decision trees, and thus follows GBDT-friendly behaviour. Thus, we cannot make any claims about the generalisation ability of the Catboost model for widespread use when it comes to noise. Apart from the Catboost model, we see that the out-of-the-box Transformer model performs relatively similar to the MLP model, although it does outperform the MLP model when no noise is present in the data. It is important to note here that we are comparing a tuned MLP model to a default Transformer model that was not tuned in any way.

We also see that for the DL models, the difference in performance on the test and training set is quite large when compared to the Catboost model. This could indicate that the models overfit more on the data, and we advise to use stronger regularisation for the DL models. In addition, the performance of the models can strongly be improved when one makes use of ensembles. We organise multiple instances of every model that are trained using different seeds in three ensembles. When the performance of those ensembles is compared to the average performance of the single models, we often see a reduction in Root Mean Squared Error (RMSE) that often exceeds 10% for DL models. Lastly, it should be noted that the tuning and training time of the models varies a lot; training one Transformer model is about 200 times slower than training one Catboost model, and about 8 times slower than one MLP model. This is also the reason why the Transformer model is not tuned. This property of the Transformer model makes it hard to scale to larger data sets containing many features, as it is simply taking too long to run. Since tabular data sets vary a lot in size, one might question if Transformer models are well-suited for tabular data at all. However, the issue with the quadratic complexity of the model could be alleviated by using alternative specifications of the MHSA method (Wang et al., 2020).

Generally speaking, we see that the performance of all models is strongly affected by noise, albeit large or small levels of noise. This indicates that the models are not very robust against noise. Since measurement errors are quite common in tabular data, this is a troubling result. The simplest solution would be to eliminate noise as much as possible, but we do recognise that this advise is not very practical. When dealing with noise in the data, it could be helpful to also consider other models and methods that have been shown to handle noise well. For instance, one could use the proposed Instrumental Variable (IV) approach for non-linear models from Hausman (2001). In addition, one could use stronger regularisation methods to prevent the models from overfitting on the data, such as a high dropout proportion for neural networks or stronger L2-regularisation for Catboost models. While it is possible to implement these techniques, it is also important note that the models used in this paper have been used in other studies as well, where they show good performance on a range of real-world data sets (Gorishniy et al., 2021). Since these data sets could also contain noise, the levels of noise in the data sets used in this study might be higher than what is normally present in real-world data.

Although the results show a clear trend when it comes to robustness against noise, we are careful with drawing any conclusions, since the data used in this study was skewed towards being GBDT friendly. It is thus not surprising that the Catboost model performs best, and we recommend future research to also take other synthetically generated data into account. For example, one could use data generated by randomly initialised neural networks or other deep learning structures that lead to a complex relation between the explanatory variables and the target variable. In addition, future research could investigate what the impact of different frequency levels of noise is on performance. Realistically speaking, noise is often not present in all data points, but only a certain percentage of data points. In this study, all data points were contaminated with noise. This is done to magnify the effect noise has on the performance of the models, but one could also investigate if the results of this study hold even when not all of the data contains noise. Apart from different types of noise, future studies could also employ other models, such as XGBoost and SAINT (Chen and Guestrin, 2016; Somepalli et al., 2021), to investigate if there is a model that handles noise significantly better. Finally, we would recommend future studies that do have access to the proper hardware to also look at the performance of tuned versions of the Transformer, since only then one can make a fair comparison to other models and draw a conclusion about the generalisation ability of the model to other data sets.

References

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

S. Ö. Arik and T. Pfister. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6679–6687, 2021.

S. Badirli, X. Liu, Z. Xing, A. Bhowmik, K. Doan, and S. S. Keerthi. Gradient boosting neural networks: Grownet. *arXiv preprint arXiv:2002.07971*, 2020.

A. Bansal, R. J. Kauffman, and R. R. Weitz. Comparing the modeling performance of regression and neural networks as data quality varies: A business value approach. *Journal of Management Information Systems*, 10(1):11–32, 1993.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

C. Dang, T. Choi, S. Lee, S. Lee, M. Alam, M. Park, S. Han, J. Lee, and D. Hoang. Machine learning-based live weight estimation for hanwoo cow. *Sustainability*, 14(19):12661, 2022.

A. S. George and A. H. George. A review of chatgpt ai's impact on several business sectors. *Partners Universal International Innovation Journal*, 1(1):9–23, 2023.

Y. Gorishniy, I. Rubachev, V. Khrulkov, and A. Babenko. Revisiting deep learning models for tabular data. *Advances in Neural Information Processing Systems*, 34:18932–18943, 2021.

Y. Gorishniy, I. Rubachev, and A. Babenko. On embeddings for numerical features in tabular deep learning. *Advances in Neural Information Processing Systems*, 35:24991–25004, 2022.

L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

J. Hausman. Mismeasured variables in econometric analysis: problems from the right and problems from the left. *Journal of Economic perspectives*, 15(4):57–67, 2001.

H. Hazimeh, N. Ponomareva, P. Mol, Z. Tan, and R. Mazumder. The tree ensemble layer: Differentiability meets conditional computation. In *International Conference on Machine Learning*, pages 4138–4148. PMLR, 2020.

X. Huang, A. Khetan, M. Cvitkovic, and Z. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *arXiv preprint arXiv:2012.06678*, 2020.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *Advances in neural information processing systems*, 30, 2017.

B. Klein and D. Rossin. Data quality in neural network models: effect of error rate and magnitude of error on predictive accuracy. *Omega*, 27(5):569–582, 1999.

J. Kossen, N. Band, C. Lyle, A. N. Gomez, T. Rainforth, and Y. Gal. Self-attention between datapoints: Going beyond individual input-output pairs in deep learning. *Advances in Neural Information Processing Systems*, 34:28742–28756, 2021.

R. Levin, A. Aravkin, and M. Kim. Patient-specific quality assurance failure prediction with deep tabular models. *medRxiv*, pages 2022–10, 2022.

Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu, et al. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models. *arXiv preprint arXiv:2304.01852*, 2023.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

R. K. Pace and R. Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3): 291–297, 1997.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

S. Popov, S. Morozov, and A. Babenko. Neural oblivious decision ensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*, 2019.

R. L. Potts and G. Leontidis. Attention-based deep learning methods for predicting gas turbine emissions. 2023.

L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.

S. M. Schennach. Measurement error in nonlinear models: A review. 2012.

G. Somepalli, M. Goldblum, A. Schwarzschild, C. B. Bruss, and T. Goldstein. Saint: Improved neural networks for tabular data via row attention and contrastive pre-training. *arXiv preprint arXiv:2106.01342*, 2021.

W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, and J. Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1161–1170, 2019.

N. Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

S. Tuli, I. Dasgupta, E. Grant, and T. L. Griffiths. Are convolutional neural networks or transformers more like human vision? *arXiv preprint arXiv:2105.07197*, 2021.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao. Learning deep transformer models for machine translation. *arXiv preprint arXiv:1906.01787*, 2019.

R. Wang, R. Shivanna, D. Cheng, S. Jain, D. Lin, L. Hong, and E. Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the web conference 2021*, pages 1785–1797, 2021.

S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

A Synthetic Data generation

The use of decision trees is inspired by Gorishniy et al., who use them to test their models in multiple studies. A detailed description of the algorithm used to randomly initialise a oblivious decision tree can be found in Gorishniy et al. (2021). A oblivious decision tree is a special type of decision tree where at every depth level the same splitting criterion is used. This produces a balanced tree and is easy to compute. For the synthetic data set creation, we use 16 independent randomly initialised oblivious decision trees, all with a depth of 4. While Gorishniy et al. (2022) use a depth of 6, this study uses more shallow trees since a lot of noise is added later on. The splitting indices for each depth level are sampled randomly from 0-9, indicating the variable to split on. Then, from the range of values of that variable, a value is randomly chosen to serve as a threshold.

After randomly initialising all decision trees, the explanatory variables with noise are created as described in Section 3. We create a total of 30.000 observations, of which 20.000 are used for training. This might not seem like much for machine learning practices, but it was necessary to limit the amount of observations since the only processor available was one CPU. The target variables are created by feeding the data through the ensemble of decision trees, where the final target value is computed by taking the average of all tree outputs. This creates a very complex relation between the randomly generated explanatory variables and the target function, but it still is a deterministic function that can be reproduced. Note that this deterministic relation becomes more fuzzy when the noise level increases. Since the target variables are computed by an ensemble of decision trees, it can be stated that the data is GBDT-friendly. The data sets can be found in the repository under the name 'syn_{noise_level}', where the {noise_level} values are $0.00, 0.01, 0.02, 0.05, 0.10, 0.20$ and 0.50. For all models, the batch size of the data sets is set at 512.

Table 6 provides an overview of the randomly generated explanatory variables for different noise levels. The average value does not change that much, as is expected, but for higher noise levels the variance increases. This is as expected, since the variance of the explanatory variables should theoretically be equal to $1 + v^2$. Since the data sets only consist of 30.000 observations, this does not exactly match the variances as stated in Table 6, but for $v = 0.5$, it does approximately equal the theoretical variance of 1.25.

Table 6: Mean and variance of the synthetically generated variables. Since all variables are sampled for the standard normal distribution, the reported values for each noise level are taken as the average over the ten variables.

	Noise Level						
	0.00	0.01	0.02	0.05	0.1	0.2	0.5
Average	0.00097	0.00094	0.00091	0.00083	0.00069	0.00040	-0.00046
Variance	0.9961	0.9962	0.9965	0.9985	1.0058	1.0355	1.2438

B Hyperparameter Tuning

In this section, the hyperparameter tuning spaces for the MLP, Transformer and Catboost will be discussed, as well as the final set of hyperparameters that was chosen for the models. In addition, the hyperparameters for the MLP and FT-Transformer model used in the replication study will be shown.

B.1 MLP

Because the original hyperparameter tuning space for the MLP model used by Gorishniy et al. (2022) is too big to use, the tuning space for the MLP in this study was selected as a subset of the bigger tuning space. The subsets are all based on an educated guess of the parameters after inspecting the best selected hyperparamter values in Gorishniy et al. (2022). An overview of the hyperparameter tuning space is shown in Table 7. Note that the number of embedding bins refers to the number of bins used in the quantile feature binning procedure for the PLE embeddings.

In Table 8, the selected hyperparameter values per data set are shown. It is clear that in general, shallow MLP models are preferred over deep models, and often with no dropout at all. It is also visible that the tuning times differ significantly from each other, and in general decreases when the noise level increases. This is due to the fact that the less complex data sets with low noise levels are trained for more epochs (training iterations) than the complex data sets.

Table 7: MLP hyperparameter tuning space.

Parameter	Distribution
# Layers	UniformInt[2,4]
Layer size	UniformInt[256,1024]
Dropout	{0, Uniform[0,0.5]}
Learning rate	LogUniform[1e-4,1e-2]
Weight decay	{0, LogUniform[1e-5,1e-4]}
# Embedding bins	UniformInt[2,256]
# Optuna iterations	40

Table 8: Best MLP hyperparameters selected per data set.

	Noise Level						
Parameter	0.00	0.01	0.02	0.05	0.1	0.2	0.5
# Layers	2	2	2	2	2	2	3
First layer	748	287	748	773	542	546	940
Middle layer(s)	-	-	-	-	-	-	661
Last layer	366	462	366	556	313	641	599
Dropout	0	0	0	0	0	0.255	0.246
Learning rate	1.30e-3	8.70e-4	1.30e-3	1.15e-3	5.85e-4	7.00e-4	6.82e-4
Weight decay	2.86e-5	7.34e-5	2.86e-5	9.22e-5	3.98e-5	0	0
# Embedding bins	146	239	146	101	145	240	71
Total tuning time	03:07:57	02:27:51	02:23:08	01:37:20	01:19:13	01:26:54	01:22:36

B.2 Transformer

For the Transformer model, the default model hyperparameters as proposed by Gorishniy et al. (2021) are used in this study. Table 9 gives an overview of these default settings. Originally, the learning rate is set to 0.0001, but we found that 0.001 produced better results. For the quantile feature binning, the number of bins is set to 256. The FFN size factor is the ratio between the dimension of the embedding vectors and the dimension of the hidden layers in the Feed-Forward Network (FFN) module of the Transformer layers (see Figure 2b). It is set to $\frac{4}{3}$, which means that for the embedding size of 192, the FFN hidden dimension is 256.

Table 9: Transformer hyperparameters.

Parameter	Value
# Transformer layers	3
Feature embedding size	192
# MHSA heads	8
FFN size factor	$\frac{4}{3}$
Attention dropout	0.2
FFN dropout	0.1
Residual dropout	0.0
Learning rate	1e-3
Weight decay	1e-5
# Optuna iterations	40

B.3 Catboost

Since the Catboost model is very fast compared to the MLP and Transformer models, we did not select a subset of the hyperparameter tuning space of the original implementation Gorishniy et al. (2022), but rather used the whole tuning space as well in this study. An overview of the hyperparameter tuning space is shown in Table 10. We set and do not tune the following hyperparameters:

`early-stopping-rounds` = 50,

`od-pval` = 0.001,

`iterations` = 2000.

Additionally, the `task_type` parameter is set to "CPU". The values of the hyperparamters after tuning on each data set can be found in Table 11. It is clearly visible that the tuning times of the Catboost model are much lower than the MLP model, and more consistent.

Table 10: Catboost hyperparameter tuning space.

Parameter	Distribution
Max depth	UniformInt[1,10]
Learning rate	LogUniform[0.001,1]
Bagging temperature	Uniform[0,1]
L2 leaf reg	LogUniform[1,10]
Leaf estimation iterations	LogUniform[1,10]
# Optuna iterations	40

Table 11: Best Catboost hyperparameters selected per data set.

				Noise Level			
Parameter	0.00	0.01	0.02	0.05	0.1	0.2	0.5
Max depth	5	6	5	5	4	5	6
Learning rate	7.65e-2	6.98e-2	7.65e-2	3.96e-2	5.73e-2	5.03e-2	2.64e-2
Bagging temperature	0.426	0.104	0.426	0.172	0.949	0.193	0.935
L2 leaf reg	2.88	3.48	2.88	2.86	8.55	7.16	9.08
Leaf estimation iterations	3	2	3	3	5	4	9
Total tuning time	00:16:18	00:14:49	00:14:46	00:18:56	00:14:08	00:13:56	00:12:10

B.4 MLP and FT-Transformer for the replication study

Below, the hyperparameters of the MLP and FT-Transformer models used in the replication study can be found. They are taken directly from the original study (Gorishniy et al., 2021), for the California Housing data set.

Table 12: MLP hyperparameters for the replication study.

Parameter	Value
# Layers	3
First layer	510
Middle layer	263
Last layer	363
Dropout	0.197
Learning rate	1.30e-3
Weight decay	1.52e-4

Table 13: Transformer hyperparameters for the replication study.

Parameter	Value
# Transformer layers	3
Feature embedding size	272
# MHSA heads	8
FFN size factor	2.34
Attention dropout	0.452
FFN dropout	0.146
Residual dropout	0.0
Learning rate	9.23e-5
Weight decay	2.24e-6

C Running the code

In this section, we discuss detailed tutorials that explain how to run the code in the repository. For the replication, we mainly use the tutorial that is provided on the GitHub repository of the original research, which can be found here Gorishniy et al. (2021). For the main code, the tutorial is also inspired by the original research on embeddings for tabular DL, which can be found here Gorishniy et al. (2022).

C.1 Replication Tutorial

- The repository we will work in, is in the zip file under 'Replication Study' → 'tabular-dl-revisiting-models-main'. Open Anaconda Prompt in this directory. Make sure to have **conda** installed on your PC/laptop.
- Now type the following commands in this window:

```
set PROJECT_DIR= C:\Users\sates\Documents\Bachelor Thesis\Replication
Study\tabular-dl-revisiting-models-main
conda create -n revisiting-models python=3.8.8
conda activate revisiting-models
conda install pytorch==1.7.1 torchvision==0.8.2 cudatoolkit=10.1.243
numpy=1.19.2 -c pytorch -y
conda install cudnn=7.6.5 -c anaconda -y
```

```
pip install -r requirements.txt
conda install nodejs -y
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

- This will set up the virtual environment with the necessary packages. Now define the following environment variables (if these commands do not work, update conda):

  ```
  set PYTHONPATH=%PYTHONPATH%;%PROJECT_DIR%
  set LD_LIBRARY_PATH=%CONDA_PREFIX%\lib;%LD_LIBRARY_PATH%
  set CUDA_HOME=%CONDA_PREFIX%
  set CUDA_ROOT=%CONDA_PREFIX%
  conda deactivate
  conda activate revisiting-models
  ```

- Now we will copy the config files used in the original study into our own replication folder.

  ```
  mkdir replication\mlp
  mkdir replication\ft_transformer
  copy output\california_housing\mlp\tuned\0.toml replication\mlp\0.toml
  copy output\california_housing\ft_transformer\tuned\0.toml replication\
  ft_transformer\0.toml
  ```

- Now we will train the models on the California Housing data set. To run the training, simply run the following commands:

  ```
  python bin/mlp.py replication/mlp/0.toml
  python bin/ft_transformer.py replication/ft_transformer/0.toml
  ```

After training, the results can be found in `replication/mlp/0` for MLP and in `replication/ft_transformer/0` for FT-Transforemer. In these folders, go to the `stats.json` files and scroll down to the bottom. There you can see the RMSE scores on the train, validation and test set. For the replication, we compare the RMSE scores on the test set to the results in Table 2 of the original research "Revisiting Deep Learning Models for Tabular Data"

C.2 Main Code Tutorial

In this section, we discuss a step-by-step tutorial for replicating the main code of this study can be run. In this guide, we explain how to tune and train an instance of the MLP model, and create ensembles from it. In addition, we discuss how to generate a synthetic data set with a user-supplied noise variance via the command line. Lastly, we discuss how to collect all the results in excel sheets.The repository we will work in is in the zip file under 'tablular-dl-num-embeddings-main'. Open Anaconda Prompt in this directory. Make sure to have **conda** installed on your PC/laptop.
For replicating the training of the MLP model:

- Type the following commands in this window:

  ```
  set PROJECT_DIR=C:\Users\sates\Documents\Bachelor Thesis\tabular-dl-num-
  embeddings-main
  conda create -n num-embeddings python=3.9.7
  conda activate num-embeddings
  pip install torch==1.10.1+cu111 -f https://download.pytorch.org/whl/
  torch_stable.html
  pip install -r requirements.txt
  ```

- This will set up the virtual environment with the necessary packages. Now define the following environment variables (if these commands do not work, update conda):

  ```
  set PYTHONPATH=%PYTHONPATH%;%PROJECT_DIR%
  set LD_LIBRARY_PATH=%CONDA_PREFIX%\lib;%LD_LIBRARY_PATH%
  set CUDA_HOME=%CONDA_PREFIX%
  set CUDA_ROOT=%CONDA_PREFIX%
  conda deactivate
  conda activate revisiting-models
  ```

- Now we will copy the config file used in the original study into our own replication folder.

  ```
  mkdir replication/mlp
  copy exp/mlp-q-lr/syn_0.0/0_tuning.toml replication/mlp/0_tuning.toml
  ```

- Now we will tune the MLP model on the `‘syn_0.0’` data set. To run the tuning, simply run the following commands. This should take around 3 hours on a standard CPU

  ```
  python bin/tune.py replication/mlp/0_tuning.toml
  ```

  This will create a new folder called `‘0_tuning’`. After completing the tuning part, you can find the saved model with the best hyperparameters in the `‘report.json’` file. If you want to skip this long tuning process, delete the `0_tuning` folder that was created, and manually copy the folder `exp/mlp-q-lr/syn_0.0/0_tuning` to the `replication/mlp` directory.

- Now we will create 15 instances of this best model, trained on 15 different seeds:

  ```
  python bin/evaluate.py replication/mlp/0_tuning 15
  ```

  This creates a new folder `‘0_evaluation’` with the 15 different models in it. Note that training each model takes up to 5 minutes.

- Finally, we can run the ensemble script to create three ensembles, each containing 5 models:

  ```
  python bin/ensemble.py replication/mlp/0_evaluation
  ```

  After ensembling, the results of each ensemble can be found in `‘0_ensemble’` → `[ensemble number]` → `‘scores.json’`.

To create a new synthetic data set, we will use the `synthetic.py` script in the ‘`bin`’ folder. This script can take a user-specified parameter for the noise level as input from the command line. To create a synthetic data set with a noise level of 0.8, enter the following command:

```
python bin/synthetic.py --noise_variance 0.8
```

The resulting data set can be found in the ‘`data`’ folder, under the name ‘`syn_0.8`’. It is already processed such that the models can use it directly, but there is also an Excel file in this folder where the data can be viewed quickly. To use the new data set, you must modify the config files itself in order for it to be used. For example, to use the MLP model on the new data set, copy the config file ‘`0_tuning`’ from ‘`exp/mlp-q-lr/syn_0.0`’ to a new folder, such as ‘`exp/mlp-q-lr/syn_0.8`’. In this copied file, change only the ‘`path`’ variable to `\data/syn_0.8"`.

Lastly, we will discuss how to collect all the results in Excel files. Therefore, we will use the ‘`gather_results.py`’ file in the ‘`bin`’ folder. This script can also take multiple inputs via the command line:

- `split_name`, indicating for which split the results should be gathered (`train, val` or `test`)
- `single_models`, indicating whether to gather the results for the single models created by the `evaluate.py` script, or if the results of the ensembles should be gathered
- `n_seeds`, is an optional parameter to gather a specific subset of single models. If set to 5, it will only gather the results of the first five single models.

We will now gather the results of the ensembles of all models, for each data set on the test part of the data set:
`python bin/gather_results.py --split_name test --single_models No` The results that are gathered can be found in the ‘`exp/results/ensembles/test`’ folder. Do note that this command gather all results from the regular data sets used in the paper, so the new ‘`syn_0.8`’ data set won’t be in this result folder.
Although the California data set is only used in the replication study, not the main study, we still leave the California Housing data set in this repository for reference, should one want to replicate the original study ‘On Embeddings for Numerical Features in Tabular Deep Learning’.