Revisiting Deep Learning Models for Tabular Data

Yury Gorishniy[*†‡] **Ivan Rubachev**[†♣] **Valentin Khrulkov**[†] **Artem Babenko**[†♣]

† Yandex, Russia
‡ Moscow Institute of Physics and Technology, Russia
♣ National Research University Higher School of Economics, Russia

Abstract

The necessity of deep learning for tabular data is still an unanswered question addressed by a large number of research efforts. The recent literature on tabular DL proposes several deep architectures reported to be superior to traditional "shallow" models like Gradient Boosted Decision Trees. However, since existing works often use different benchmarks and tuning protocols, it is unclear if the proposed models universally outperform GBDT. Moreover, the models are often not compared to each other, therefore, it is challenging to identify the best deep model for practitioners.

In this work, we start from a thorough review of the main families of DL models recently developed for tabular data. We carefully tune and evaluate them on a wide range of datasets and reveal two significant findings. First, we show that the choice between GBDT and DL models highly depends on data and there is still no universally superior solution. Second, we demonstrate that a simple ResNet-like architecture is a surprisingly effective baseline, which outperforms most of the sophisticated models from the DL literature. Finally, we design a simple adaptation of the Transformer architecture for tabular data that becomes a new strong DL baseline and reduces the gap between GBDT and DL models on datasets where GBDT dominates. The source code is available at https://github.com/yandex-research/rtdl.

1 Introduction

Due to the tremendous success of deep learning for unstructured data like raw images, audio and texts (Goodfellow et al., 2016), there has been a lot of research interest to extend this success to problems with structured data stored in tabular form. In these problems, data points are represented as vectors of heterogeneous features, which is typical for industrial applications and ML competitions, where non-deep models, e.g., GBDT (Chen and Guestrin, 2016; Ke et al., 2017; Prokhorenkova et al., 2018) are currently the top-choice solutions. Along with potentially higher performance, using deep learning for tabular data is appealing as it would allow to construct multi-modal pipelines for problems, where only one part of the input is structured, and other parts include images, audio and other DL-friendly data. Such pipelines can then be trained end-to-end by gradient optimization for both structured and unstructured modalities. For these reasons, a large number of DL solutions were recently proposed, and new models continue to emerge (Arik and Pfister, 2020; Badirli et al., 2020; Hazimeh et al., 2020; Huang et al., 2020a; Klambauer et al., 2017; Popov et al., 2020; Song et al., 2019; Wang et al., 2017, 2020a).

Unfortunately, due to the lack of established benchmarks (such as ImageNet (Deng et al., 2009) for computer vision or GLUE (Wang et al., 2019a) for NLP) and the fast development of the field, existing papers use different datasets for evaluation, and proposed DL models are often not adequately compared to each other. Therefore, from the current literature, it is unclear whether GBDT is

[*]Correspondence to: yura.gorishniy@phystech.edu, ygorishniy@yandex-team.ru

Preprint. Under review.

surpassed by DL models or what DL model generally performs better than others. These obscurities impede the research process and make the observations from the papers not conclusive enough.

Given the increasing number of works on tabular DL, we believe it is timely to review the recent developments from the field to identify more justified conclusions that can serve as a basis for future studies. To this end, we perform a large-scale evaluation of many recent open-sourced DL models on a wide range of datasets with the same protocol of hyperparameter tuning. Our study has revealed several curious findings. First, we do not observe a definitive superiority of GBDT and DL models w.r.t. each other, and their relative performance largely depends on the particular task. For instance, GBDT is superior for data with heterogeneous features, but lags behind DL solutions on non-heterogeneous data and poorly scales to classification problems when the number of classes is large (1K+). Second, we investigate the relative performance of DL models to each other. Surprisingly, we discover that most recent models are inferior to a simple ResNet-like architecture. Given its high performance and simplicity, it can serve as a reasonable baseline for future works on tabular DL. Finally, we design a Transformer-based architecture (Vaswani et al., 2017) that outperforms the ResNet baseline on tasks with heterogeneous features and becomes a new strong DL baseline. Overall, we hope that our evaluation protocol and the described baselines will lay the foundation for more decisive studies on tabular DL.

We summarize the contributions of our paper as follows:

1. We thoroughly evaluate the main models for tabular DL on a wide range of problems to investigate their relative performance and for the appropriate comparison to GBDT.
2. We reveal that GBDT and deep models are superior for different types of problems.
3. We demonstrate that a simple ResNet-like architecture is an effective baseline for tabular DL, which was overlooked by existing literature. Given its simplicity, this baseline is necessary for comparison in future tabular DL works.
4. We introduce a simple adaptation of the Transformer architecture for tabular data that demonstrates state-of-the-art performance among DL solutions on tasks with heterogeneous data.

2 Related work

The "shallow" state-of-the-art for problems with tabular data is currently ensembles of decision trees, such as GBDT (Gradient Boosting Decision Tree) (Friedman, 2001), which are typically the top-choice in various ML competitions. At the moment, there are several established GBDT libraries, such as XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017), CatBoost (Prokhorenkova et al., 2018), which are widely used by both ML researchers and practitioners. While these implementations vary in detail, on most of the tasks, their performances do not differ much (Prokhorenkova et al., 2018).

During several recent years, a large number of deep learning models for tabular data have been developed (Arik and Pfister, 2020; Badirli et al., 2020; Hazimeh et al., 2020; Huang et al., 2020a; Klambauer et al., 2017; Popov et al., 2020; Song et al., 2019; Wang et al., 2017). Most of these models can be roughly categorized into three groups, which we briefly describe below.

Differentiable trees. The first group of models is motivated by the strong performance of decision tree ensembles for tabular data. Since decision trees are not differentiable and do not allow gradient optimization, they cannot be used as a component for pipelines trained in the end-to-end fashion. To address this issue, several works (Hazimeh et al., 2020; Kontschieder et al., 2015; Popov et al., 2020; Yang et al., 2018) propose to "smooth" decision functions in the internal tree nodes to make the overall tree function and tree routing differentiable. While the methods of this family can outperform GBDT on some tasks (Popov et al., 2020), in our experiments, they did not outperform ResNet.

Attention-based models. Due to the ubiquitous success of attention-based architectures for different domains (Dosovitskiy et al., 2021; Vaswani et al., 2017), several authors propose to employ attention-like modules for tabular DL as well (Arik and Pfister, 2020; Huang et al., 2020a; Song et al., 2019). In our experiments, we show that the properly tuned ResNet outperforms the existing attention-based models. Nevertheless, we identify an effective way to apply the Transformer architecture (Vaswani et al., 2017) to tabular data: the resulting architecture outperforms ResNet on most of the tasks.

Explicit modeling of multiplicative interactions. In the literature on recommender systems and click-through-rate prediction, several works criticize MLP since it is unsuitable for modeling multiplicative interactions between features (Beutel et al., 2018; Qin et al., 2021; Wang et al., 2017). Inspired by this motivation, some works (Beutel et al., 2018; Wang et al., 2017, 2020a) have proposed different ways to incorporate feature products into MLP. In our experiments, however, we did not find such methods to be superior to properly tuned baselines.

The literature also proposes some other architectural designs (Badirli et al., 2020; Klambauer et al., 2017) that cannot be explicitly assigned to any of the groups above. Overall, the community has developed a variety of models that are evaluated on different benchmarks and are rarely compared to each other. Our work aims to establish a fair comparison of them and identify the solutions that consistently provide high performance.

3 Comparing the models for tabular data

3.1 Preliminaries

Notation. In this work, we consider supervised learning problems. $D{=}\{(x_i,\ y_i)\}_{i=1}^{n}$ denotes a dataset, where $x_i{=}(x_i^{(num)},\ x_i^{(cat)}) \in \mathbb{X}$ represents numerical $x_{ij}^{(num)}$ and categorical $x_{ij}^{(cat)}$ features of an object and $y_i \in \mathbb{Y}$ denotes the corresponding object label. The total number of features is denoted as k. The dataset is split into three disjoint subsets: $D = D_{train} \cup D_{val} \cup D_{test}$, where D_{train} is used for training, D_{val} is used for early stopping and hyperparameter tuning and D_{test} is used for the final evaluation. We consider three types of tasks: binary classification $\mathbb{Y} = \{0,\ 1\}$, multiclass classification $\mathbb{Y} = \{1,\ \ldots,\ C\}$ and regression $\mathbb{Y} = \mathbb{R}$.

Scope of the comparison. In our work, we focus on the relative performance of different architectures and do not employ various model-agnostic DL practices, such as pretraining, additional loss functions, data augmentation, distillation, learning rate warmup, learning rate decay and many others. While these practices can potentially improve the performance, our goal is to evaluate the impact of inductive biases imposed by the different model architectures.

3.2 Baselines

First, we include two established "shallow" models that are currently trusted by practitioners as well as the simplest DL models that should serve as a natural sanity check for testing novel deep architectures against them:

- **XGBoost** (Chen and Guestrin, 2016). One of the most popular GBDT implementations.
- **CatBoost** (Prokhorenkova et al., 2018). GBDT implementation that uses oblivious decision trees (Lou and Obukhov, 2017) as week learners.
- **MLP**. The simplest feed-forward model consisting of `Linear-ReLU-Dropout` blocks (details are provided in supplementary).
- Additionally, we employ **ResNet** — an MLP-like architecture with skip connections (He et al., 2015b) and batch normalizations (Ioffe and Szegedy, 2015) (details are provided supplementary). Surprisingly, this simple baseline is typically overlooked in existing papers. Moreover, a similar solution has been recently reported to perform competitively on some NLP tasks (Sun and Iyyer, 2021), which additionally motivates us to include ResNet in comparison.

Second, we cover the following deep architectures that were specifically designed for tabular data:

- **SNN** (Klambauer et al., 2017). An MLP-like architecture with the SELU activation that enables training deeper models.
- **NODE** (Popov et al., 2020). A differentiable ensemble of oblivious decision trees.
- **TabNet** (Arik and Pfister, 2020). A recurrent architecture that alternates dynamical reweighing of features and conventional feed-forward modules.
- **GrowNet** (Badirli et al., 2020). Gradient boosted weak MLPs. The official implementation supports only classification and regression problems.

- **DCN V2** (Wang et al., 2020a). Consists of an MLP-like module and the feature crossing module (a combination of linear layers and multiplications).
- **AutoInt** (Song et al., 2019). Transforms features to tokens and applies self-attention on the token level.

All technical details are provided in supplementary.

3.3 Datasets and metrics.

Datasets. We use a diverse set of eleven public datasets (see supplementary for the detailed description). For each dataset, there is exactly one train-val-test split, so all algorithms use the same splits. We informally call the dataset's features "heterogeneous" if they describe different physical properties of the object and are of diverse units of measurements, which is somewhat opposite to homogeneous features, such as pixels for images or words for text data. We apply this term to California Housing (CA, real estate data, Kelley Pace and Barry (1997)), Covertype (CO, forest characteristics, Blackard and Dean. (2000)), Yahoo (YA, search queries, Chapelle and Chang (2011)) and Microsoft (MI, search queries, Qin and Liu (2013)). Non-heterogeneous datasets include Higgs (HI, physical particles, Baldi et al. (2014)), ALOI (AL, images, Geusebroek et al. (2005)), Epsilon (EP, simulated physics experiments) and Year (YE, audio features, Bertin-Mahieux et al. (2011)). We do not assign anonymous datasets (Helena (HE) and Jannis (JA), Guyon et al. (2019)) and datasets with few numerical features (Adult (AD), Kohavi (1996)) to either heterogeneous or non-heterogeneous. We follow the pointwise approach to learning-to-rank and treat ranking problems (Microsoft, Yahoo) as regression problems.
Datasets scope. In our work, dataset sizes vary from 20K to 1M+, and the number of features is significantly less than the number of samples.
Metrics. For binary classification problems, we report Area Under the ROC Curve (AUC). For multiclass problems, we report accuracy. For regression problems, we report Root-Mean-Square Error.

3.4 Implementation details

Data preprocessing. Data preprocessing is known to be vital for DNN models. For each dataset, the same preprocessing was used for all models for fair comparison. By default, we used the quantile transformation from the Scikit-learn library (Pedregosa et al., 2011). We applied standardization (mean subtraction and scaling) to Helena and ALOI. The latter one represents image data and standardization is a normal practice in computer vision. We do not preprocess the Epsilon dataset, since we observed it to be highly detrimental to deep models' performance. We apply standardization (mean subtraction and scaling) to regression targets for all algorithms.

Tuning. For every dataset, we carefully tune each model's hyperparameters. The best hyperparameters are the ones that perform best on the validation set, so the test set is never used for tuning. For most algorithms, we use the Optuna library (Akiba et al., 2019) to run Bayesian optimization (the Tree-Structured Parzen Estimator algorithm), which is reported to be superior to random search (Turner et al., 2021). For the rest, we iterate over a predefined set of configurations recommended by corresponding papers. For details, see supplementary.

Evaluation. For each tuned configuration, we run 15 experiments with different random seeds and report the performance on the test set. For some algorithms, we also report the performance of default configurations without hyperparameter tuning.

Neural networks. We minimize cross-entropy for classification problems and mean squared error for regression problems. For TabNet and GrowNet, we follow the original implementations and use the Adam optimizer (Kingma and Ba, 2017). For all other algorithms, we use the AdamW optimizer (Loshchilov and Hutter, 2019). We do not apply learning rate schedules. For each dataset, we use a predefined batch size for all algorithms unless special instructions on batch sizes are given in the corresponding papers (see supplementary). We continue training until there are $\texttt{patience} + 1$ consecutive epochs without improvements on the validation set; we set $\texttt{patience} = 16$ for all algorithms.

See supplementary for information on hardware and training time.

3.5 The comparison results

Table 1: Results for single models. See supplementary for standard deviations. For each dataset, top results for baseline neural networks are in **bold**, top results for baseline neural networks and FT-Transformer are in **blue**, the overall top results are in **red**. "Top" means "the gap between this result and the result with the best mean score is not statistically significant". Datasets with heterogeneous features are underlined. FT-Transformer is introduced in section 3.6. Notation: "d" ~ "default", ↓ ~ lower is better, ↑ ~ higher is better.

	CA ↓	AD ↑	HE ↑	JA ↑	HI ↑	AL ↑	EP ↑	YE ↓	CO ↑	YA ↓	MI ↓
Baseline Neural Networks											
SNN	0.507	**0.816**	0.3728	0.718	0.721	0.954	0.8970	8.881	0.9465	0.769	0.7521
TabNet	0.513	0.796	0.3782	0.724	0.717	0.954	0.8902	9.032	0.9335	0.819	0.7565
GrowNet	0.500	0.793	–	–	0.724	–	**0.8977**	8.866	–	0.775	0.7549
DCN2	0.486	0.784	0.3853	0.714	0.720	0.955	**0.8975**	8.939	0.9491	0.766	0.7500
AutoInt	0.479	0.801	0.3722	0.716	**0.726**	0.945	0.8948	8.875	0.9312	0.795	0.7517
MLP	0.494	0.796	0.3832	0.719	0.721	0.954	0.8968	8.861	0.9499	0.776	0.7521
NODE	**0.464**	0.791	0.3593	**0.726**	0.724	0.918	0.8958	**8.774**	0.9436	**0.762**	**0.7474**
ResNet	0.487	**0.816**	**0.3960**	**0.727**	**0.727**	**0.963**	0.8971	8.845	**0.9560**	0.766	0.7493
FT-Transformer											
FT-Transformer$_d$	0.470	0.799	0.3812	0.725	0.723	0.953	0.8959	8.869	0.9617	**0.758**	**0.7475**
FT-Transformer	**0.464**	0.807	0.3913	**0.731**	**0.728**	0.960	**0.8982**	8.820	**0.9641**	**0.758**	**0.7469**
GBDT											
CatBoost$_d$	**0.430**	0.797	0.3814	0.721	0.724	0.946	0.8882	8.913	0.9076	0.751	0.7454
CatBoost	**0.431**	0.791	0.3853	0.723	0.725	–	0.8880	8.877	0.9658	0.743	0.7429
XGBoost$_d$	0.463	0.775	0.3502	0.721	0.705	0.925	0.8803	9.446	0.9640	0.773	0.7719
XGBoost	0.433	0.796	0.3755	0.724	0.725	–	0.8857	8.947	**0.9695**	**0.736**	**0.7424**

Table 2: Results for ensembles. Notation follows Table 1, but top results are now defined by the mean score. Standard deviations are reported in supplementary.

	CA ↓	AD ↑	HE ↑	JA ↑	HI ↑	AL ↑	EP ↑	YE ↓	CO ↑	YA ↓	MI ↓
Baseline Neural Networks											
SNN	0.485	**0.822**	0.3804	0.722	0.725	0.962	0.8971	8.747	0.9542	0.762	0.7487
TabNet	0.492	0.809	0.3908	**0.734**	0.724	0.961	0.8952	8.773	0.9497	0.814	0.7505
GrowNet	0.483	0.794	–	–	0.731	–	**0.8986**	8.718	–	0.766	0.7504
DCN2	0.478	0.786	0.3884	0.721	0.721	0.960	0.8977	8.764	0.9575	0.762	0.7488
AutoInt	**0.461**	0.805	0.3822	0.728	**0.732**	0.959	0.8967	8.730	0.9519	0.782	0.7477
MLP	0.489	0.806	0.3902	0.722	0.726	0.960	0.8970	8.716	0.9547	0.768	0.7510
NODE	0.461	0.791	0.3609	0.728	0.725	0.921	0.8968	**8.710**	0.9579	**0.758**	**0.7464**
ResNet	0.478	0.818	**0.3981**	0.733	0.731	**0.966**	0.8978	8.733	**0.9607**	0.760	0.7467
FT-Transformer											
FT-Transformer$_d$	0.455	0.801	0.3948	0.735	0.730	0.966	0.8969	8.719	**0.9695**	**0.748**	**0.7429**
FT-Transformer	**0.450**	0.810	**0.3983**	**0.737**	0.731	**0.967**	0.8984	8.722	0.9692	**0.748**	0.7434
GBDT											
CatBoost$_d$	0.428	0.798	0.3863	0.724	0.726	0.948	0.8894	8.885	0.9096	0.7490	0.7440
CatBoost	**0.423**	0.794	0.3885	0.727	0.726	–	0.8899	8.837	0.9685	0.7401	**0.7413**
XGBoost$_d$	0.463	0.775	0.3502	0.721	0.705	0.925	0.8803	9.446	0.9640	0.7732	0.7719
XGBoost	0.431	0.796	0.3767	0.725	0.725	–	0.8880	8.819	**0.9696**	**0.7320**	0.7421

Table 1 reports the comparison of the deep models w.r.t. each other and GBDT. Since GBDT is essentially an ensembling technique, we also compare all the competitors in the ensembling mode. We build three ensemble predictions by splitting the 15 single model predictions into three disjoint

groups of size five each. As shown in Table 2, ensembling significantly improves the performance of deep architectures, which is consistent with existing literature on this phenomenon (Fort et al., 2020).

3.5.1 Which DNN architecture is the best?

Table 1 indicates that ResNet yields the most consistent performance among single models, as it is the best model for six datasets out of eleven and it is very close to the leader on Epsilon. On the remaining datasets, NODE is the only model that shows non-negligible improvement over ResNet. However, NODE has an ensemble-like structure (and requires significantly more resources for training), so it is not a truly "single" model, and the gap is mostly gone in the ensembling mode, see Table 2. Surprisingly, MLP is often on par or even better than some of the recently proposed models (SNN, TabNet, GrowNet, DCN V2).
The main take away: ResNet is a strong baseline that none of the recently proposed architectures can outperform and MLP is still a good sanity check.

3.5.2 How GBDT and DNN are compared?

Both Table 1 and Table 2 demonstrate that GBDT is strictly superior to DNN on heterogeneous data (see section 3.3), while on non-heterogeneous data situation is rather the opposite. We also faced slow training of GBDT on multiclass problems with a large number of classes (ALOI ~ 1000 classes) and failed to tune CatBoost and XGBoost due to this phenomenon. This is a well-known issue and sophisticated modifications (such as Si et al. (2017)) are required to alleviate it, but none of them are available in popular GBDT implementations, so DNN solutions should be preferred for this type of tasks.
The main take away: DL research efforts aimed at surpassing GBDT should cover a diverse set of heterogeneous datasets in addition to non-heterogeneous datasets, which should serve only as a sanity check.

3.6 FT-Transformer

In this section, we introduce FT-Transformer (**F**eature **T**okenizer + **Transformer**) — a simple adaptation of the Transformer architecture (Vaswani et al., 2017) for the tabular domain. Figure 1 demonstrates the main parts of FT-Transformer. In a nutshell, our model transforms all features (categorical and numerical) to tokens and runs a stack of Transformer layers over the tokens, so every Transformer layer operates on the *feature* level of *one* object. We compare FT-Transformer to conceptually similar AutoInt in section 4.2.

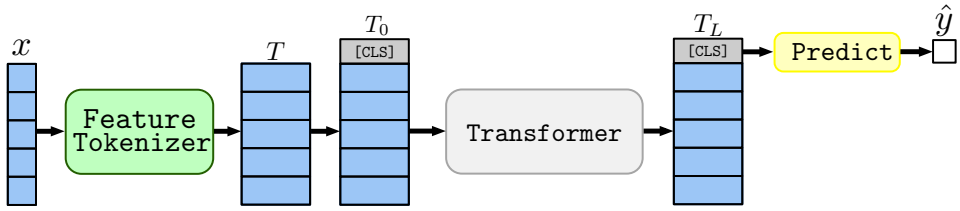

Figure 1: The FT-Transformer architecture. Firstly, Feature Tokenizer transforms features to tokens. The tokens are then processed by the Transformer module and the `[CLS]` token is used for prediction.

Feature Tokenizer. The Feature Tokenizer module (see Figure 2) transforms the input features x to tokens $T \in \mathbb{R}^{k \times d}$. The token for a given feature x_j is computed as follows:

$$T_j = b_j + f_j(x_j) \in \mathbb{R}^d \qquad f_j : \mathbb{X}_j \to \mathbb{R}^d.$$

where b_j is the j-th *feature bias*, $f_j^{(num)}$ is implemented as the element-wise multiplication with the *direction* $W_j^{(num)} \in \mathbb{R}^d$ and $f_j^{(cat)}$ is implemented as the lookup table $W_j^{(cat)} \in \mathbb{R}^{S_j \times d}$ for

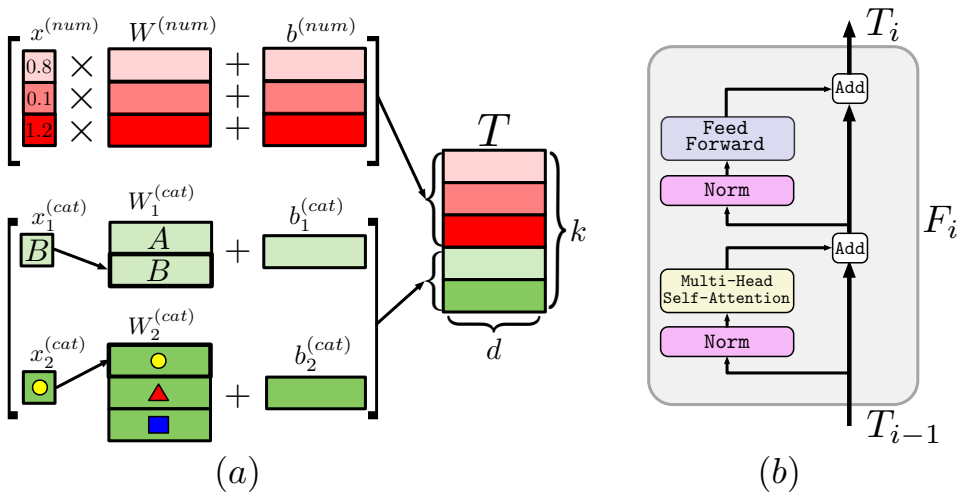

Figure 2: (a) Feature Tokenizer; in the example, there are three numerical and two categorical features; (b) One Transformer layer.

categorical features. Overall:

$$
\begin{aligned}
T_j^{(num)} &= b_j^{(num)} + x_j^{(num)} \cdot W_j^{(num)} && \in \mathbb{R}^d, \\
T_j^{(cat)} &= b_j^{(cat)} + e_j^T W_j^{(cat)} && \in \mathbb{R}^d, \\
T &= \texttt{stack}\left[T_1^{(num)}, \ \ldots, \ T_{k^{(num)}}^{(num)}, \ T_1^{(cat)}, \ \ldots, \ T_{k^{(cat)}}^{(cat)}\right] && \in \mathbb{R}^{k \times d}.
\end{aligned}
$$

where e_j^T is a one-hot vector for the corresponding categorical feature.

Transformer. At this stage, the `[CLS]` token (or "classification token", or "output token", see Devlin et al. (2019)) is appended to T and L Transformer layers $F_1, \ \ldots, \ F_L$ are applied:

$$
T_0 = \texttt{stack}\left[\texttt{[CLS]}, \ T\right] \qquad T_i = F_i(T_{i-1}).
$$

We use the PreNorm variant for easier optimization (Wang et al., 2019b), see Figure 2. In the PreNorm setting, we also found it to be necessary to remove the first normalization from the first Transformer layer to achieve good performance. See the original paper (Vaswani et al., 2017) for the background on Multi-Head Self-Attention (MHSA) and the Feed Forward module. See supplementary for details such as activations, placement of normalizations and dropout modules (Srivastava et al., 2014).

Prediction. The final representation of the `[CLS]` token is used for prediction:

$$
\hat{y} = \texttt{Linear}(\texttt{ReLU}(\texttt{LayerNorm}(T_L^{\texttt{[CLS]}}))).
$$

Limitations. FT-Transformer requires more resources (both hardware and time) for training than simple models such as ResNet and may not be easily scaled to datasets when the number of features is "too large" (it is determined by the available hardware and time budget). The main reason lies in the quadratic complexity of the vanilla MHSA with respect to the number of features. However, the issue can be alleviated by using efficient approximations of MHSA (Tay et al., 2020). Additionally, it is still possible to distill FT-Transformer into simpler architectures for better inference performance. We report training times and the used hardware in supplementary.

3.6.1 FT-Transformer vs baseline neural networks

Table 1 and Table 2 indicate the advantage of FT-Transformer

- FT-Transformer is superior to all recently proposed DL architectures.
- FT-Transformer is superior to ResNet on heterogeneous data and performs on par with ResNet on non-heterogeneous data (see section 3.3). We develop this intuition further in section 4.1.

- FT-Transformer strongly benefits from ensembling: in this mode, FT-Transformer closes the gap with ResNet on two datasets (Helena and ALOI).

Interestingly, FT-Transformer appears to be not very sensitive to hyperparameters:

- FT-Transformer allows for building powerful ensembles without tuning: ensembles of default FT-Transformers demonstrate similar performance to ensembles of tuned ones.
- On heterogeneous data, even the default configuration of FT-Transformer already performs well and beats most competitors.

3.6.2 FT-Transformer vs GBDT

In this section, we compare FT-Transformer with CatBoost and XGBoost.

We start with observations on the overall performance using Table 1 and Table 2.

- **Default hyperparameters.** Ensembled FT-Transformer is superior to GBDT, with California as the only exception. As for single models, FT-Transformer consistently demonstrates adequate off-the-shelf performance. The default XGBoost turns out to be uncompetitive (we observed extreme overfitting). CatBoost is more reliable in that regard, with only one exception (Covertype).
- **Tuned hyperparameters.** Once hyperparameters are properly tuned, GBDTs start dominating on the heterogeneous datasets, while FT-Transformer keeps leadership on DNN-friendly problems. Nevertheless, a small "success story" happens on one heterogeneous dataset (Covertype), where FT-Transformer becomes the only DNN that performs on par with GBDT (see Table 2).
- **Ensembling.** The benefit from ensembling for GBDT is clearly limited compared to FT-Transformer. It makes sense, since a single GBDT is already an ensemble-like model, which is not the case for FT-Transformer.

4 Analysis

4.1 When FT-Transformer is better than ResNet?

In this section, we make the first step towards understanding the difference in behavior between FT-Transformer and ResNet. To achieve that, we design a sequence of synthetic tasks where the difference between performance of the two models gradually changes from negligible to dramatic. Namely, we generate and *fix* objects $\{x_i\}_{i=1}^{n}$, perform the train-val-test split *once* and interpolate between two regression targets: f_{GBDT}, which is supposed to be "GBDT friendly" and f_{DNN}, which is expected to be easier for DNN. Formally, for one object:

$$x \sim \mathcal{N}(0, I_k),$$
$$y = \alpha \cdot f_{GBDT}(x) + (1 - \alpha) \cdot f_{DNN}(x).$$

where $f_{GBDT}(x)$ is an average prediction of 30 randomly constructed decision trees, and $f_{DNN}(x)$ is an MLP with three randomly initialized hidden layers. Both f_{GBDT} and f_{DNN} are generated once, i.e. the same functions are applied to all objects (see supplementary for details). The resulting targets are standardized before training. The results are visualized in Figure 3. ResNet and FT-Transformer perform similarly well on the DNN-friendly tasks and outperform CatBoost on those tasks. However, the ResNet's relative performance drops significantly when the target becomes more GBDT friendly. By contrast, FT-Transformer yields competitive performance across the whole range of tasks.

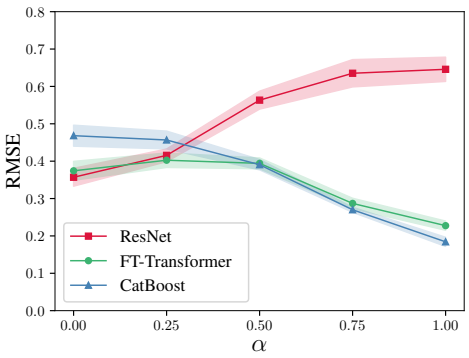

Figure 3: RMSE for the same test set with five different regression targets. The results are averaged over five seeds, shadows represent standard deviations.

The conducted experiment reveals a type of functions that are better approximated by FT-Transformer than by ResNet. Additionally, the fact that these functions are based on decision trees correlates with the results in Table 1 and Table 2, where FT-Transformer shows the most convincing improvements over other deep models exactly on those datasets where GBDT dominates.

4.2 Ablation study

Table 3: The results of the comparison between FT-Transformer and two attention-based alternatives: AutoInt and FT-Transformer without feature biases. Notation follows Table 1

	CA ↓	HE ↑	JA ↑	HI ↑	AL ↑	YE ↓	CO ↑	MI ↓
AutoInt	0.479	0.372	0.716	**0.726**	0.945	8.875	0.931	0.752
FT-Transformer (w/o feature biases)	0.472	0.381	0.724	0.723	0.958	**8.821**	0.962	0.751
FT-Transformer	**0.464**	**0.391**	**0.731**	**0.728**	**0.960**	**8.820**	**0.964**	**0.747**

In this section, we test some design choices of FT-Transformer.

First, we compare FT-Transformer with AutoInt (Song et al., 2019), since it is the closest competitor in its spirit. AutoInt also converts all features to tokens and applies self-attention on top of them. However, in its details, AutoInt significantly differs from FT-Transformer: its "tokenizer" does not include feature biases, its backbone significantly differs from the vanilla Transformer (Vaswani et al., 2017), and the inference mechanism does not use the `[CLS]` token.

Second, we check whether feature biases in Feature Tokenizer are essential for good performance.

We tune and evaluate FT-Transformer without feature biases following the same protocol as in section 3.4 and reuse the remaining numbers from section 3.5. The results averaged over 15 runs are reported in Table 3 and demonstrate both the superiority of the Transformer's backbone to that of Autoint and the necessity of feature biases.

4.3 Obtaining feature importances from attention maps

In this section, we evaluate attention maps as a source of information on feature importances for FT-Transformer for a given set of samples. For the i-th sample, we calculate the average attention map p_i for the `[CLS]` token from Transformer's forward pass. Then, the obtained individual distributions are averaged into one distribution p that represents the feature importances:

$$p = \frac{1}{n_{samples}} \sum_i p_i \qquad p_i = \frac{1}{n_{heads} \times L} \sum_{h,l} p_{ihl}.$$

where p_{ihl} is the h-th head's attention map for the `[CLS]` token from the forward pass of the l-th layer on the i-th sample. The main advantage of the described heuristic technique is its efficiency: it requires a single forward for one sample.

In order to evaluate our approach, we compare it with Integrated Gradients (IG, Sundararajan et al. (2017)), a general technique applicable to any differentiable model. We use permutation test (PT, Breiman (2001)) as a reasonable interpretable method which allows to establish a constructive metric, namely, rank correlation. We run all the methods on the train set and summarize results in Table 4. Interestingly, the proposed method yields reasonable feature importances and performs similarly to

Table 4: Rank correlation (takes values in $[-1,\ 1]$) between permutation test's feature importances ranking and two alternative rankings: Attention Maps (AM) and Integrated Gradients (IG). Means and standard deviations over five runs are reported.

	CA	HE	JA	HI	AL	YE	CO	MI
AM	0.81 (0.05)	0.77 (0.03)	0.78 (0.05)	0.91 (0.03)	0.84 (0.01)	0.92 (0.01)	0.84 (0.04)	0.86 (0.02)
IG	0.84 (0.08)	0.74 (0.03)	0.75 (0.04)	0.72 (0.03)	0.89 (0.01)	0.50 (0.03)	0.90 (0.02)	0.56 (0.02)

IG (note that this does not imply similarity to IG's feature importances). Given that IG can be orders of magnitude slower and the "baseline" in the form of PT requires $(n_{features} + 1)$ forward passes (versus one for the proposed method), we conclude that the simple averaging of attention maps can be a good choice in terms of cost-effectiveness.

5 Conclusion

In this work, we have investigated the status quo in the field of deep learning for tabular data. By the extensive experimental comparison of the recent DL models on various datasets, we have identified that most of them do not consistently outperform the simplest ResNet-like baseline as well as established GBDT implementations. Given these observations, we have proposed a new attention-based architecture that outperforms ResNet on many tasks. The code and all the details of the study are open-sourced [2], and we hope that our evaluation and two simple baselines (ResNet and FT-Transformer) will serve as a basis for further developments on tabular DL.

References

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *KDD*, 2019.

S. O. Arik and T. Pfister. Tabnet: Attentive interpretable tabular learning. *arXiv*, 1908.07442v5, 2020.

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv*, 1607.06450v1, 2016.

S. Badirli, X. Liu, Z. Xing, A. Bhowmik, K. Doan, and S. S. Keerthi. Gradient boosting neural networks: Grownet. *arXiv*, 2002.07971v2, 2020.

P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5, 2014.

T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

A. Beutel, P. Covington, S. Jain, C. Xu, J. Li, V. Gatto, and E. H. Chi. Latent cross: Making use of context in recurrent recommender systems. In *WSDM 2018: The Eleventh ACM International Conference on Web Search and Data Mining*, 2018.

J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151, 2000.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. In *Proceedings of the Learning to Rank Challenge*, volume 14, 2011.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *SIGKDD*, 2016.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv*, 1810.04805v2, 2019.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

S. Fort, H. Hu, and B. Lakshminarayanan. Deep ensembles: A loss landscape perspective. *arXiv*, 1912.02757v2, 2020.

[2] https://github.com/yandex-research/rtdl

J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

J. M. Geusebroek, G. J. Burghouts, , and A. W. M. Smeulders. The amsterdam library of object images. *Int. J. Comput. Vision*, 61(1):103–112, 2005.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

I. Guyon, L. Sun-Hosoya, M. Boullé, H. J. Escalante, S. Escalera, Z. Liu, D. Jajetic, B. Ray, M. Saeed, M. Sebag, A. Statnikov, W. Tu, and E. Viegas. Analysis of the automl challenge series 2015-2018. In *AutoML*, Springer series on Challenges in Machine Learning, 2019.

H. Hazimeh, N. Ponomareva, P. Mol, Z. Tan, and R. Mazumder. The tree ensemble layer: Differentiability meets conditional computation. In *ICML*, 2020.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015a.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv*, 1512.03385v1, 2015b.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, 2016.

X. Huang, A. Khetan, M. Cvitkovic, and Z. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *arXiv*, 2012.06678v1, 2020a.

X. S. Huang, F. Perez, J. Ba, and M. Volkovs. Improving transformer optimization through better initialization. In *ICML*, 2020b.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30: 3146–3154, 2017.

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

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv*, 1412.6980v9, 2017.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *NIPS*, 2017.

R. Kohavi. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *KDD*, 1996.

P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo. Deep neural decision forests. In *Proceedings of the IEEE international conference on computer vision*, 2015.

L. Liu, X. Liu, J. Gao, W. Chen, and J. Han. Understanding the difficulty of training transformers. In *EMNLP*, 2020.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

Y. Lou and M. Obukhov. Bdt: Gradient boosted decision tables for high accuracy and scoring efficiency. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

S. Narang, H. W. Chung, Y. Tay, W. Fedus, T. Fevry, M. Matena, K. Malkan, N. Fiedel, N. Shazeer, Z. Lan, Y. Zhou, W. Li, N. Ding, J. Marcus, A. Roberts, and C. Raffel. Do transformer modifications transfer across implementations and applications? *arXiv*, 2102.11972v1, 2021.

T. Q. Nguyen and J. Salazar. Transformers without tears: Improving the normalization of self-attention. In *IWSLT*, 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *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. In *ICLR*, 2020.

L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features. In *NeurIPS*, 2018.

T. Qin and T. Liu. Introducing LETOR 4.0 datasets. *arXiv*, 1306.2597v1, 2013.

Z. Qin, L. Yan, H. Zhuang, Y. Tay, R. K. Pasumarthi, X. Wang, M. Bendersky, and M. Najork. Are neural rankers still outperformed by gradient boosted decision trees? In *ICLR*, 2021.

N. Shazeer. Glu variants improve transformer. *arXiv*, 2002.05202v1, 2020.

S. Si, H. Zhang, S. S. Keerthi, D. Mahajan, I. S. Dhillon, and C. Hsieh. Gradient boosted decision trees for high dimensional sparse output. In *ICML*, 2017.

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 *CIKM*, 2019.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1): 1929–1958, 2014.

S. Sun and M. Iyyer. Revisiting simple neural probabilistic language models. In *NAACL*, 2021.

M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *ICML*, 2017.

Y. Tay, M. Dehghani, D. Bahri, and D. Metzler. Efficient transformers: A survey. *arXiv*, 2009.06732v1, 2020.

R. Turner, D. Eriksson, M. McCourt, J. Kiili, E. Laaksonen, Z. Xu, and I. Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. *arXiv*, https://arxiv.org/abs/2104.10201v1, 2021.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, 2017.

A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *ICLR*, 2019a.

Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao. Learning deep transformer models for machine translation. In *ACL*, 2019b.

R. Wang, B. Fu, G. Fu, and M. Wang. Deep & cross network for ad click predictions. In *ADKDD*, 2017.

R. Wang, R. Shivanna, D. Z. Cheng, S. Jain, D. Lin, L. Hong, and E. H. Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. *arXiv*, 2008.13535v2, 2020a.

S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *arXiv*, 2006.04768v3, 2020b.

N. Wies, Y. Levine, D. Jannai, and A. Shashua. Which transformer architecture fits my data? a vocabulary bottleneck in self-attention. In *ICLM*, 2021.

F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80, 1945.

T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Huggingface's transformers: State-of-the-art natural language processing. *arXiv*, 1910.03771v5, 2020.

Y. Yang, I. G. Morillo, and T. M. Hospedales. Deep neural decision trees. *arXiv*, 1806.06988v1, 2018.

Supplementary materials

A Data

A.1 Datasets

Table 5: Datasets description

Name	Abbr	# Train	# Validation	# Test	# Num	# Cat	Task type	Batch size
California Housing	CA	13209	3303	4128	8	0	Regression	256
Adult	AD	26048	6513	16281	6	8	Binclass	256
Helena	HE	41724	10432	13040	27	0	Multiclass	512
Jannis	JA	53588	13398	16747	54	0	Multiclass	512
Higgs Small	HI	62752	15688	19610	28	0	Binclass	512
ALOI	AL	69120	17280	21600	128	0	Multiclass	512
Epsilon	EP	320000	80000	100000	2000	0	Binclass	1024
Year	YE	370972	92743	51630	90	0	Regression	1024
Covtype	CO	371847	92962	116203	54	0	Multiclass	1024
Yahoo	YA	473134	71083	165660	699	0	Regression	1024
Microsoft	MI	723412	235259	241521	136	0	Regression	1024

A.2 Preprocessing

For regression problems we standardize the target values:

$$y_{new} = \frac{y_{old} - \texttt{mean}(y_{train}))}{\texttt{std}(y_{train})} \tag{1}$$

The feature preprocessing for DNNs is described in the main text. Note that before performing the quantile transformation, we add noise to all numerical features *once* (i.e. *before* training) as a workaround for features with few distinct values. See the source code for the exact implementation. We do not preprocess features for GBDTs, since this family of algorithms is insensitive to feature shifts and scaling.

B Models

In this section, we describe the implementation details for all models.

B.1 XGBoost

Implementation. We fix and do not tune the following hyperparameters:

- `booster` = "gbtree"
- `early-stopping-rounds` = 50
- `n-estimators` = 2000

In Table 6, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 6: XGBoost hyperparameter space. Here (A) = {CA, AD, HE, JA, HI} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
Max depth	(A) $\mathrm{UniformInt}[3, 10]$, (B) $\mathrm{UniformInt}[6, 10]$
Min child weight	(A,B) $\mathrm{LogUniform}[1e\text{-}8, 1e5]$
Subsample	(A,B) $\mathrm{Uniform}[0.5, 1]$
Learning rate	(A,B) $\mathrm{LogUniform}[1e\text{-}5, 1]$
Col sample by level	(A,B) $\mathrm{Uniform}[0.5, 1]$
Col sample by tree	(A,B) $\mathrm{Uniform}[0.5, 1]$
Gamma	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}8, 1e2]\}$
Lambda	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}8, 1e2]\}$
Alpha	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}8, 1e2]\}$
# Iterations	100

B.2 CatBoost

Implementation. We fix and do not tune the following hyperparameters:

- `early-stopping-rounds` $= 50$
- `od-pval` $= 0.001$
- `iterations` $= 2000$

In Table 7, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019). We set the `task_type` parameter to "GPU" (the tuning was unacceptably slow on CPU).

Table 7: CatBoost hyperparameter space. Here (A) = {CA, AD, HE, JA, HI} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
Max depth	(A) $\mathrm{UniformInt}[3, 10]$, (B) $\mathrm{UniformInt}[6, 10]$
Learning rate	(A,B) $\mathrm{LogUniform}[1e\text{-}5, 1]$
Bagging temperature	(A,B) $\mathrm{Uniform}[0, 1]$
L2 leaf reg	(A,B) $\mathrm{LogUniform}[1, 10]$
Leaf estimation iterations	(A,B) $\mathrm{UniformInt}[1, 10]$
# Iterations	100

Evaluation. We set the `task_type` parameter to "CPU", since for the used version of the CatBoost library it is crucial for performance in terms of target metrics.

B.3 MLP

Architecture.

$$\texttt{MLP}(x) = \texttt{Linear}\left(\texttt{MLPBlock}\left(\ldots\left(\texttt{MLPBlock}(x)\right)\right)\right)$$
$$\texttt{MLPBlock}(x) = \texttt{Dropout}(\texttt{ReLU}(\texttt{Linear}(x)))$$

Implementation. Ours, see the source code.

In Table 8, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019). Note that the size of the first and the last layers are tuned and set separately, while the size for "in-between" layers is the same for all of them.

Table 8: MLP hyperparameter space. Here (A) = {CA, AD, HE, JA, HI, AL} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
# Layers	(A) $\mathrm{UniformInt}[1, 8]$, (B) $\mathrm{UniformInt}[1, 16]$
Layer size	(A) $\mathrm{UniformInt}[1, 512]$, (B) $\mathrm{UniformInt}[1, 1024]$
Dropout	(A,B) $\{0, \mathrm{Uniform}[0, 0.5]\}$
Learning rate	(A,B) $\mathrm{LogUniform}[1e\text{-}5, 1e\text{-}2]$
Weight decay	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}6, 1e\text{-}3]\}$
Category embedding size	({AD}) $\mathrm{UniformInt}[64, 512]$
# Iterations	100

B.4 ResNet

Architecture. We tested several configurations and observed measurable difference in performance between all of them. We found the ones with "clear main path" (i.e. with normalizations placed only in residual branches as in He et al. (2016) or Wang et al. (2019b)) to perform better. As expected, it is also easier for them to train deeper configurations. We found the block design inspired by Transformer (Vaswani et al., 2017) to perform better or on par with the one inspired by the ResNet from computer vision (He et al., 2015b). The following equations describe the architecture that was referred to as "ResNet" throughout the main text:

$$\begin{aligned}
\texttt{ResNet}(x) &= \texttt{Prediction}\left(\texttt{ResNetBlock}\left(\ldots\left(\texttt{ResNetBlock}\left(\texttt{Linear}(x)\right)\right)\right)\right) \\
\texttt{ResNetBlock}(x) &= x + \texttt{Dropout}(\texttt{Linear}(\texttt{Dropout}(\texttt{ReLU}(\texttt{Linear}(\texttt{BatchNorm}(x)))))) \\
\texttt{Prediction}(x) &= \texttt{Linear}\left(\texttt{ReLU}\left(\texttt{BatchNorm}\left(x\right)\right)\right)
\end{aligned}$$

We observed that in the "optimal" configurations (the result of the hyperparameter optimization process) the inner dropout rate (not the last one) of one block was usually set to higher values compared to the outer dropout rate. Moreover, the latter one was set to zero in many cases.

Implementation. Ours, see the source code.

In Table 9, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 9: ResNet hyperparameter space. Here (A) = {CA, AD, HE, JA, HI, AL} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
# Layers	(A) $\mathrm{UniformInt}[1, 8]$, (B) $\mathrm{UniformInt}[1, 16]$
Layer size	(A) $\mathrm{UniformInt}[64, 512]$, (B) $\mathrm{UniformInt}[64, 1024]$
Hidden factor	(A,B) $\mathrm{Uniform}[1, 4]$
Hidden dropout	(A,B) $\mathrm{Uniform}[0, 0.5]$
Residual dropout	(A,B) $\{0, \mathrm{Uniform}[0, 0.5]\}$
Learning rate	(A,B) $\mathrm{LogUniform}[1e\text{-}5, 1e\text{-}2]$
Weight decay	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}6, 1e\text{-}3]\}$
Category embedding size	({AD}) $\mathrm{UniformInt}[64, 512]$
# Iterations	100

B.5 SNN

Implementation. Ours, see the source code.

In Table 10, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 10: SNN hyperparameter space. Here (A) = {CA, AD, HE, JA, HI, AL} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
# Layers	(A) UniformInt$[2, 16]$, (B) UniformInt$[2, 32]$
Layer size	(A) UniformInt$[1, 512]$, (B) UniformInt$[1, 1024]$
Dropout	(A,B) $\{0, \text{Uniform}[0, 0.1]\}$
Learning rate	(A,B) LogUniform$[1e\text{-}5, 1e\text{-}2]$
Weight decay	(A,B) $\{0, \text{LogUniform}[1e\text{-}5, 1e\text{-}3]\}$
Category embedding size	({AD}) UniformInt$[64, 512]$
# Iterations	100

B.6 NODE

Implementation. We used the official implementation: https://github.com/Qwicen/node.

Tuning. We iterated over the parameter grid from the original paper (Popov et al., 2020) plus the default configuration from the original paper. For multiclass datasets, we set the tree dimension being equal to the number of classes. For the Helena and ALOI datasets there was no tuning since NODE does not scale for large number of classes (for example, the minimal non-default configuration of NODE contains 600M+ parameters on the Helena dataset), so the reported results for these datasets are obtained with the default configuration.

B.7 TabNet

Implementation. We used the official implementation: https://github.com/google-research/google-research/tree/master/tabnet. We always set `feature-dim` equal to `output-dim`. We also fix and do not tune the following hyperparameters (let A = {CA, AD}, B = {HE, JA, HI, AL}, C = {EP, YE, CO, YA, MI}):

- `virtual-batch-size` $=$ (A) 2048, (B) 8192, (C) 16384
- `batch-size` $=$ (A) 256, (B) 512, (C) 1024

In Table 11, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 11: TabNet hyperparameter space.

Parameter	Distribution
# Decision steps	UniformInt$[3, 10]$
Layer size	$\{8, 16, 32, 64, 128\}$
Relaxation factor	Uniform$[1, 2]$
Sparsity loss weight	LogUniform$[1e\text{-}6, 1e\text{-}1]$
Decay rate	Uniform$[0.4, 0.95]$
Decay steps	$\{100, 500, 2000\}$
Learning rate	Uniform$[1e\text{-}3, 1e\text{-}2]$
# Iterations	100

B.8 GrowNet

Implementation. We used the official implementation: https://github.com/sbadirli/GrowNet. Note that it does not support multiclass problems, hence the gaps in the main tables

for multiclass problems. We use no more than 40 small MLPs, each MLP has 2 hidden layers, boosting rate is learned – as suggested by the authors.

Tuning. We use grid search to tune the learning rate and hidden layer size of the small MLP with the grid Table 12.

Table 12: GrowNet tunnig grid

Parameter	Values
Layer size	$\{32, 64, 128\}$
Learning rate	$\{0.0001, 0.001, 0.01\}$

B.9 DCN V2

Architecture. There are two variats of DCN V2, namely, "stacked" and "parallel". We tuned and evaluated both and did not observe strong superiority of any of them. We report numbers for the "parallel" variant as it was slightly better on large datasets.

Implementation. Ours, see the source code.

In Table 13, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 13: DCN V2 hyperparameter space. Here (A) = {CA, AD, HE, JA, HI, AL} and (B) = {EP, YE, CO, YA, MI}

Parameter	(Datasets) Distribution
# Cross layers	(A) $\mathrm{UniformInt}[1, 8]$, (B) $\mathrm{UniformInt}[1, 16]$
# Hidden layers	(A) $\mathrm{UniformInt}[1, 8]$, (B) $\mathrm{UniformInt}[1, 16]$
Layer size	(A) $\mathrm{UniformInt}[64, 512]$, (B) $\mathrm{UniformInt}[64, 1024]$
Hidden dropout	(A,B) $\mathrm{Uniform}[0, 0.5]$
Cross dropout	(A,B) $\{0, \mathrm{Uniform}[0, 0.5]\}$
Learning rate	(A,B) $\mathrm{LogUniform}[1e\text{-}5, 1e\text{-}2]$
Weight decay	(A,B) $\{0, \mathrm{LogUniform}[1e\text{-}6, 1e\text{-}3]\}$
Category embedding size	({AD}) $\mathrm{UniformInt}[64, 512]$
# Iterations	100

B.10 AutoInt

Implementation. Ours, see the source code. We mostly follow the original paper (Song et al., 2019), however, it turns out to be necessary to introduce some modifications such as normalization in order to make the model competitive. We fix $n_{heads} = 2$ as recommended in the original paper.

In Table 14, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019).

Table 14: AutoInt hyperparameter space. Here (A) = {CA, AD, HE, JA, HI} and (B) = {AL, YE, CO, MI}

Parameter	(Datasets) Distribution
# Layers	(A,B) $\mathrm{UniformInt}[1, 6]$
Token size	(A,B) $\mathrm{UniformInt}[8, 64]$
Residual dropout	(A) $\{0, \mathrm{Uniform}[0.0, 0.2]\}$, (B) $\mathrm{Const}(0.0)$
Attention dropout	(A,B) $\mathrm{Uniform}[0.0, 0.5]$
Learning rate	(A) $\mathrm{LogUniform}[1e\text{-}5, 1e\text{-}3]$, (B) $\mathrm{LogUniform}[3e\text{-}5, 3e\text{-}4]$
Weight decay	(A,B) $\mathrm{LogUniform}[1e\text{-}6, 1e\text{-}3]$
# Iterations	(A) 100, (B) 50

C FT-Transformer

In this section, we formally describe the details of FT-Transformer its tuning and evaluation. Also, we share additional technical experience and observations that were not used for final results in the paper, but may be of interest to researchers and practitioners.

C.1 Architecture

Formal definition.

$$\begin{aligned}
\texttt{FT-Transformer}(x) &= \texttt{Prediction}(\texttt{Block}(\ldots(\texttt{Block}(\texttt{FeatureTokenizer}(x))))) \\
\texttt{Block}(x) &= \texttt{ResidualPreNorm}(\texttt{FFN},\ \texttt{ResidualPreNorm}(\texttt{MHSA},\ x)) \\
\texttt{ResidualPreNorm}(\texttt{Module},\ x) &= x + \texttt{Dropout}(\texttt{Module}(\texttt{Norm}(x))) \\
\texttt{FFN}(x) &= \texttt{Linear}(\texttt{Dropout}(\texttt{Activation}(\texttt{Linear}(x))))
\end{aligned}$$

We use `LayerNorm` (Ba et al., 2016) as the normalization. See the main text for the description of `Prediction` and `FeatureTokenizer`. For MHSA, we set $n_{heads} = 8$ and do not tune this parameter.

Activation. Throughout the whole paper we used the ReGLU activation, since it is reported to be superior to the usually used GELU activation (Narang et al., 2021; Shazeer, 2020). However, we did not observe strong difference between ReGLU and ReLU in preliminary experiments.

Dropout rates. We observed that the attention dropout is always beneficial and FFN-dropout is also usually set by the tuning process to some non-zero value. As for the final dropout of each residual branch, it is rarely set to non-zero values by the tuning process.

PreNorm vs PostNorm. We use the PreNorm variant of Transformer, i.e. normalizations are placed at the beginning of each residual branch. The PreNorm variant is known for better optimization properties as opposed to the original Transformer, which is a PostNorm-Transformer (Liu et al., 2020; Nguyen and Salazar, 2019; Wang et al., 2019b). The latter one may produce better models in terms of target metrics (Liu et al., 2020), but it usually requires additional modifications to the model and/or the training process, such as learning rate warmup or complex initialization schemes (Huang et al., 2020b; Liu et al., 2020). While the PostNorm variant can be an option for practitioners seeking for the best possible model, we use the PreNorm variant in order to keep the optimization simple and same for all models. Note that in the PostNorm formulation the `LayerNorm` in the "Prediction" equation (see the section "FT-Transformer" in the main text) should be omitted.

C.2 The default configuration(s)

Table 15 describes the configuration of FT-Transformer referred to as "default" in the main text. Note that it includes hyperparameters for both the model and the optimization. In fact, the configuration is a result of an "educated guess" and we did not invest much resources in its tuning.

where "FFN size factor" is a ratio of the `FFN`'s hidden size to the token size.

Table 15: Default FT-Transformer used in the main text.

Layer count	3	
Token size	192	
Head count	8	
Activation & FFN size factor	(`ReGLU`, $4/3$)	
Attention dropout	0.2	
FFN dropout	0.1	
Residual dropout	0.0	
Initialization	`Kaiming`	(He et al., 2015a)
Parameter count	929K	The value is given for 100 numerical features
Optimizer	`AdamW`	
Learning rate	$1e{-4}$	
Weight decay	$1e{-5}$	0.0 for Feature Tokenizer, `LayerNorm` and biases

We also designed a heuristic scaling rule to produce "default" configurations with the number of layers from one to six. We applied it on the Epsilon and Yahoo datasets in order to reduce the number of tuning iterations. However, we did not dig into the topic and our scaling rule may be suboptimal, see Wies et al. (2021) for a theoretically sound scaling rule.

In Table 16, we provide hyperparameter space used for Optuna-driven tuning (Akiba et al., 2019). For Epsilon and Yahoo, however, we iterated over several "default" configurations using a heuristic scaling rule, since the full tuning procedure turned out to be too time consuming.

Table 16: FT-Transformer hyperparameter space. Here (A) = {CA, AD, HE, JA, HI} and (B) = {AL, YE, CO, MI}

Parameter	(Datasets) Distribution
# Layers	(A) $\mathrm{UniformInt}[1, 4]$, (B) $\mathrm{UniformInt}[1, 6]$
Token size	(A,B) $\mathrm{UniformInt}[64, 512]$
Residual dropout	(A) $\{0, \mathrm{Uniform}[0, 0.2]\}$, (B) $\mathrm{Const}(0.0)$
Attention dropout	(A,B) $\mathrm{Uniform}[0, 0.5]$
FFN dropout	(A,B) $\mathrm{Uniform}[0, 0.5]$
FFN factor	(A) $\mathrm{Uniform}[2/3, 8/3]$, (B) $\mathrm{Const}(4/3)$
Learning rate	(A) $\mathrm{LogUniform}[1e\text{-}5, 1e\text{-}3]$, (B) $\mathrm{LogUniform}[3e\text{-}5, 3e\text{-}4]$
Weight decay	(A,B) $\mathrm{LogUniform}[1e\text{-}6, 1e\text{-}3]$
# Iterations	(A) 100, (B) 50

C.3 Training

On the Epsilon dataset, we scale FT-Transformer using the technique proposed by Wang et al. (2020b) with the "headwise" sharing policy; we set the projection dimension to 128. We follow the popular "transformers" library (Wolf et al., 2020) and do not apply weight decay to Feature Tokenizer, biases in linear layers and normalization layers.

D Experiments

To report statistical significance in the main text we use the one-sided Wilcoxon (1945) test with $p = 0.01$.

Table 17 and Table 18 are more detailed versions of Table 1 and Table 2 from the main text.

Table 17: Results for single models with standard deviations. Notation follows Table 1 from the main text.

	CA ↓	AD ↑	HE ↑	JA ↑	HI ↑	AL ↑	EP ↑	YE ↓	CO ↑	YA ↓	MI ↓
Baseline Neural Networks											
SNN	0.507±6.6e-3	**0.816±6.5e-3**	0.3728±2.9e-3	0.718±2.4e-3	0.721±1.7e-3	0.954±1.7e-3	0.8970±3.1e-4	8.881±2.9e-2	0.9465±1.8e-3	0.769±1.4e-3	0.7521±4.5e-4
TabNet	0.513±1.5e-2	0.796±1.3e-2	0.3782±1.8e-3	0.724±2.2e-3	0.717±1.8e-3	0.954±1.1e-3	0.8902±1.9e-3	9.032±7.0e-2	0.9335±7.8e-3	0.819±4.0e-3	0.7565±2.3e-3
GrowNet	0.500±6.0e-3	0.793±6.5e-3	–	–	0.724±1.8e-3	–	**0.8977±4.8e-4**	8.866±2.1e-2	–	0.775±2.7e-3	0.7549±1.2e-3
DCN2	0.486±3.6e-3	0.784±5.6e-3	0.3853±3.1e-3	0.714±1.7e-3	0.720±1.3e-3	0.955±1.2e-3	**0.8975±4.3e-4**	8.939±4.0e-2	0.9491±1.8e-3	0.766±5.0e-4	0.7500±2.9e-4
AutoInt	0.479±4.1e-3	0.801±6.6e-3	0.3722±2.6e-3	0.716±2.3e-3	**0.726±2.3e-3**	0.945±1.4e-3	0.8948±7.2e-4	8.875±3.7e-2	0.9312±4.0e-3	0.795±5.7e-3	0.7517±6.1e-4
MLP	0.494±3.6e-3	0.796±5.9e-3	0.3832±2.6e-3	0.719±1.7e-3	0.721±1.7e-3	0.954±1.4e-3	0.8968±2.0e-4	8.861±1.5e-2	0.9499±2.0e-3	0.776±1.9e-3	0.7521±2.0e-4
NODE	**0.464±1.9e-3**	0.794±6.8e-3	0.3593±2.1e-3	**0.726±1.5e-3**	0.724±1.1e-3	0.918±5.5e-3	0.8958±4.5e-4	**8.774±1.5e-2**	0.9436±2.5e-3	**0.762±3.5e-4**	**0.7474±2.9e-4**
ResNet	0.487±4.7e-3	**0.816±1.6e-3**	**0.3960±1.8e-3**	**0.727±1.3e-3**	**0.727±1.9e-3**	**0.963±7.6e-4**	0.8971±3.7e-4	8.845±3.5e-2	**0.9560±1.9e-3**	0.766±8.8e-4	0.7493±5.7e-4
FT-Transformer											
FT-Transformer$_d$	0.470±3.6e-3	0.799±5.1e-3	0.3812±2.5e-3	0.725±1.4e-3	0.723±2.5e-3	0.953±1.2e-3	0.8959±4.7e-4	8.869±4.4e-2	0.9617±8.7e-4	**0.758±9.7e-4**	**0.7475±5.7e-4**
FT-Transformer	**0.464±5.1e-3**	0.807±6.8e-3	0.3913±1.2e-3	**0.731±1.4e-3**	**0.728±1.9e-3**	0.960±1.1e-3	**0.8982±2.9e-4**	8.820±2.6e-2	**0.9641±7.2e-4**	**0.758±9.7e-4**	**0.7469±7.9e-4**
GBDT											
CatBoost$_d$	**0.430±7.4e-4**	0.797±1.8e-3	0.3814±1.5e-3	0.721±1.2e-3	0.724±6.2e-4	0.946±9.3e-4	0.8882±4.0e-4	8.913±5.5e-3	0.9076±2.4e-4	0.751±2.0e-4	0.7454±2.4e-4
CatBoost	**0.431±1.6e-3**	0.791±2.7e-3	0.3853±1.2e-3	0.723±1.6e-3	0.725±1.1e-3	–	0.8880±5.1e-4	8.877±6.1e-3	0.9658±2.8e-4	0.743±2.4e-4	0.7429±2.1e-4
XGBoost$_d$	0.463± –	0.775± –	0.3502± –	0.721± –	0.705± –	0.925± –	0.8803± –	9.446± –	0.9640± –	0.773± –	0.7719± –
XGBoost	0.433±1.6e-3	0.796±1.1e-3	0.3755±8.3e-4	0.724±6.5e-4	0.725±5.7e-4	–	0.8857±3.8e-4	8.947±8.5e-3	**0.9695±1.8e-4**	**0.736±2.2e-4**	**0.7424±1.4e-4**

Table 18: Results for ensembles with standard deviations. Notation follows Table 1 from the main text.

	CA ↓	AD ↑	HE ↑	JA ↑	HI ↑	AL ↑	EP ↑	YE ↓	CO ↑	YA ↓	MI ↓
Baseline Neural Networks											
SNN	0.485±3.0e-3	**0.822±3.1e-3**	0.3804±1.2e-3	0.722±1.2e-3	0.725±1.0e-4	0.962±2.9e-4	0.8971±1.3e-4	8.747±9.9e-3	0.9542±2.7e-4	0.762±8.5e-4	0.7487±1.1e-4
TabNet	0.492±5.7e-3	0.809±5.5e-3	0.3908±3.2e-4	**0.734±7.3e-4**	0.724±9.3e-4	0.961±2.9e-4	0.8952±7.9e-4	8.773±4.8e-3	0.9497±1.2e-3	0.814±2.7e-3	0.7505±1.0e-3
GrowNet	0.483±4.9e-4	0.794±5.0e-4	–	–	0.731±3.5e-4	–	**0.8986±1.1e-4**	8.718±2.9e-3	–	0.766±5.0e-4	0.7504±4.9e-4
DCN2	0.478±1.8e-3	0.786±4.5e-3	0.3884±1.5e-3	0.721±4.8e-4	0.721±8.4e-4	0.960±4.2e-4	0.8977±8.6e-5	8.764±1.2e-2	0.9575±6.6e-4	0.762±1.9e-4	0.7488±3.1e-4
AutoInt	**0.461±1.8e-3**	0.805±3.6e-3	0.3822±3.8e-4	0.728±1.7e-3	**0.732±1.1e-3**	0.959±1.7e-4	0.8967±1.4e-4	8.730±4.0e-3	0.9519±1.0e-3	0.782±3.6e-3	0.7477±2.9e-4
MLP	0.489±7.6e-4	0.806±2.7e-3	0.3902±1.5e-3	0.722±1.8e-3	0.726±3.8e-4	0.960±3.3e-4	0.8970±1.6e-4	8.716±3.4e-3	0.9547±5.4e-4	0.768±2.4e-4	0.7510±6.9e-5
NODE	0.461±7.3e-4	0.797±3.8e-3	0.3609±8.0e-4	0.728±9.7e-4	0.725±4.5e-5	0.921±1.7e-3	0.8968±1.9e-4	**8.710±3.8e-3**	0.9579±3.8e-4	**0.758±2.7e-4**	**0.7464±8.3e-5**
ResNet	0.478±2.2e-3	0.818±4.9e-4	**0.3981±7.3e-4**	0.733±4.2e-4	0.731±3.9e-4	**0.966±4.9e-4**	0.8978±9.7e-5	8.733±1.8e-2	**0.9607±4.0e-4**	0.760±4.6e-4	0.7467±6.8e-5
FT-Transformer											
FT-Transformer$_d$	0.455±2.5e-3	0.801±9.0e-4	0.3948±9.5e-4	0.735±3.8e-4	0.730±1.8e-3	0.966±3.9e-4	0.8969±2.1e-4	8.719±1.3e-2	**0.9695±1.7e-4**	**0.748±6.7e-4**	**0.7429±2.1e-4**
FT-Transformer	**0.450±3.2e-4**	0.810±1.9e-3	**0.3983±4.4e-4**	**0.737±9.6e-4**	0.731±6.3e-4	**0.967±4.8e-4**	0.8984±1.7e-4	8.722±1.2e-2	0.9692±3.1e-4	0.748±6.7e-4	0.7434±3.7e-4
GBDT											
CatBoost$_d$	0.428±4.6e-5	0.798±8.5e-4	0.3863±1.0e-3	0.724±4.9e-4	0.726±3.1e-4	0.948±9.3e-4	0.8894±1.2e-4	8.885±1.9e-3	0.9096±3.0e-4	0.749±1.2e-4	0.7440±4.4e-5
CatBoost	**0.423±9.0e-4**	0.794±1.7e-3	0.3885±2.7e-4	0.727±6.5e-4	0.726±5.9e-4	–	0.8899±1.3e-4	8.837±3.3e-3	0.9685±2.3e-5	0.740±1.8e-4	**0.7413±7.4e-5**
XGBoost$_d$	0.463± –	0.775± –	0.3502± –	0.721± –	0.705± –	0.925± –	0.8803± –	9.446± –	0.9640± –	0.773± –	0.7719± –
XGBoost	0.431±3.7e-4	0.796±6.9e-4	0.3767±2.7e-4	0.725±1.2e-4	0.725±1.5e-4	–	0.8880±1.4e-4	8.819±4.0e-3	**0.9696±6.8e-5**	**0.732±5.4e-5**	0.7421±1.8e-5

D.1 Software and hardware

All the experiments were conducted in the same conditions in terms of software versions. For almost all experiments the used hardware can be found in the source code. The average workflow for each model-dataset pair is as follows:

- tune the model on any suitable hardware
- evaluate model on one or more NVidia Tesla V100 32Gb

The described approach allows to eliminate differences in the final performance caused by difference in hardware or software. There were *rare* exceptions from the described approach in terms of hardware, but never in terms of software (for instance, CatBoost was evaluated on CPU because of CatBoost's limitations).

D.2 Training times

Table 19: Training times in seconds averaged over 15 runs.

	CA	AD	HE	JA	HI	AL	EP	YE	CO	YA	MI
ResNet	156	33	363	126	218	933	347	573	3322	294	591
FT-Transformer	107	233	537	344	244	2864	802	1119	2483	12876	4251
Overhead	x0.7	x7.1	x1.5	x2.7	x1.1	x3.1	x2.3	x2.0	x0.7	x43.8	x7.2

For most experiments, training times can be found in the source code. In Table 19, we provide the comparison between ResNet and FT-Transformer in order to "visualize" the overhead introduced by FT-Transformer compared to the main DL baseline. The huge difference on the Yahoo dataset is expected because of the large number of features (700).

E Analysis

E.1 When FT-Transformer is better than ResNet?

Data. Train, validation and test set sizes are $500\,000$, $50\,000$ and $100\,000$ respectively. One object is generated as $x \sim \mathcal{N}(0, I_{100})$. For each object, the first 50 features are used for target generation and the remaining 50 features play the role of "noise".

f_{DNN}. The function is implemented as an MLP with 3 hidden layers, each of size 256. Weights are initialized with Kaiming initialization (He et al., 2015a), biases are initialized with the uniform distribution $\mathcal{U}(-a,\ a)$, where $a = d_{input}^{-0.5}$. All the parameters are fixed after initialization and are not trained.

f_{GBDT}. The function is implemented as an average prediction of 30 randomly constructed decision trees. The construction of one random decision tree is demonstrated in algorithm 1. The inference process for one decision tree is the same as for ordinary decision trees.

CatBoost. We use the default hyperparameters.

FT-Transformer. We use the default hyperparameters. Parameter count: $930K$.

ResNet. Residual block count: 4. Embedding size: 256. Dropout rate inside residual blocks: 0.5. Parameter count: $820K$.

E.2 Ablation study

Table 20 is a more detailed version of the corresponding table from the main text.

Algorithm 1: Construction of one random decision tree.

Result: Random Decision Tree
set of leaves $L = \{\texttt{root}\}$;
`depths` - mapping from nodes to their depths;
`left` - mapping from nodes to their left children;
`right` - mapping from nodes to their right children;
`features` - mapping from nodes to splitting features;
`thresholds` - mapping from nodes to splitting thresholds;
`values` - mapping from leaves to their associated values;
$n = 0$ - number of nodes;
$k = 100$ - number of features;
while $n < 100$ **do**
  randomly choose leaf z from L s.t. $\texttt{depths}[z] < 10$;
  $\texttt{features}[z] \sim \texttt{UniformInt}[1, \ldots, k]$;
  $\texttt{thresholds}[z] \sim \mathcal{N}(0, 1)$;
  add two new nodes l and r to L;
  remove z from L;
  unset $\texttt{values}[z]$;
  $\texttt{left}[z] = l$;
  $\texttt{right}[z] = r$;
  $\texttt{depths}[l] = \texttt{depths}[r] = \texttt{depths}[z] + 1$;
  $\texttt{values}[l] \sim \mathcal{N}(0, 1)$;
  $\texttt{values}[r] \sim \mathcal{N}(0, 1)$;
  $n = n + 2$;
end
return Random Decision Tree as $\{L, \texttt{left}, \texttt{right}, \texttt{features}, \texttt{thresholds}, \texttt{values}\}$.

Table 20: The results of the comparison between FT-Transformer and two attention-based alternatives: AutoInt and FT-Transformer without feature biases. Means and standard deviations over 15 runs are reported. Notation follows Table 1 from the main text.

	CA ↓	HE ↑	JA ↑	HI ↑	AL ↑	YE ↓	CO ↑	MI ↓
AutoInt	$0.479_{\pm 0.004}$	$0.372_{\pm 0.003}$	$0.716_{\pm 0.002}$	$\mathbf{0.726_{\pm 0.002}}$	$0.945_{\pm 0.001}$	$8.875_{\pm 0.037}$	$0.931_{\pm 0.004}$	$0.752_{\pm 0.001}$
FT-Transformer (w/o feat. bias.)	$0.472_{\pm 0.006}$	$0.381_{\pm 0.002}$	$0.724_{\pm 0.003}$	$0.723_{\pm 0.002}$	$0.958_{\pm 0.001}$	$\mathbf{8.821_{\pm 0.022}}$	$0.962_{\pm 0.001}$	$0.751_{\pm 0.001}$
FT-Transformer	$\mathbf{0.464_{\pm 0.005}}$	$\mathbf{0.391_{\pm 0.001}}$	$\mathbf{0.731_{\pm 0.001}}$	$\mathbf{0.728_{\pm 0.002}}$	$\mathbf{0.960_{\pm 0.001}}$	$\mathbf{8.820_{\pm 0.026}}$	$\mathbf{0.964_{\pm 0.001}}$	$\mathbf{0.747_{\pm 0.001}}$