nanoTabPFN: A Lightweight and Educational Reimplementation of TabPFN

Alexander Pfefferle[*1,2] **Johannes Hog**[*2] **Lennart Purucker**[2] **Frank Hutter**[3,1,2]
[1]ELLIS Institute Tübingen [2]University of Freiburg [3]Prior Labs
pfeffera@cs.uni-freiburg.de

Abstract

Tabular foundation models such as TabPFN have revolutionized predictive machine learning for tabular data. At the same time, the driving factors of this revolution are hard to understand. Existing open-source tabular foundation models are implemented in complicated pipelines boasting over 10 000 lines of code, lack architecture documentation or code quality. In short, the implementations are hard to understand, not beginner-friendly, and complicated to adapt for new experiments. We introduce nanoTabPFN, a simplified and lightweight implementation of the TabPFN v2 architecture and a corresponding training loop that uses pre-generated training data. nanoTabPFN makes tabular foundation models more accessible to students and researchers alike. For example, restricted to a small data setting it achieves a performance comparable to traditional machine learning baselines within **one minute of pre-training on a single GPU** (160 000× faster than TabPFN v2 pretraining). This eliminated requirement of large computational resources makes pre-training tabular foundation models accessible for educational purposes. Our code is available at `https://github.com/automl/nanoTabPFN`.

1 Introduction

The field of tabular data has recently been undergoing significant changes with the introduction of Tabular Foundation models. This revolution was started with the introduction of TabPFN [Hollmann et al., 2023] and continued with newer foundation models such as TabDPT [Ma et al., 2024], TabICL [Qu et al., 2025], LimiX [Zhang et al., 2025a] and TabPFN v2 [Hollmann et al., 2025]. TabPFN v2 significantly improved over TabPFN by introducing a new architecture, a new prior for training data, and many small tricks in the inference pipeline. This improvement comes with a significant increase in the complexity of its implementation, with the official repository currently boasting over 10 000 lines of Python code. Such complexity presents a substantial hurdle for researchers and students who want to understand, modify, or build upon TabPFN.

We solve this issue by introducing nanoTabPFN, a simplified and lightweight implementation of the TabPFN v2 architecture and a training pipeline in under 500 lines of code. We also provide an interface to load pre-generated training data. Our code can be used to pre-train nanoTabPFN for small tabular prediction tasks within minutes. The lightweight and modular design of our code allows users to quickly familiarize themselves with tabular foundation models and allows for fast iterations of research ideas for the prior, training pipeline, or architecture. We believe that nanoTabPFN will serve as a first step in the journey of students towards TFMs and will make this field of research more accessible.

The contributions of our work are: (1) our repository itself, containing a simplified reimplementation of the TabPFN v2 architecture, along with an in-depth explanation and (2) experiments showing

[*]Equal contribution.

AI for Tabular Data workshop at EurIPS 2025

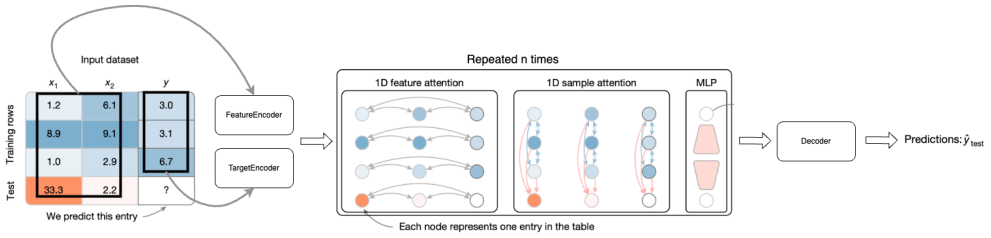

Figure 1: **nanoTabPFN Architecture.** The architecture consists of the `FeatureEncoder`, which normalizes and embeds the features, the `TargetEncoder`, which pads up the labels to the full length of rows and embeds the Targets, followed by a repeated `TransformerEncoderStack`, and the `Decoder`, which maps the high-dimensional embeddings to our predictions. Adapted from Figure 1 of Hollmann et al. [2025].

that nanoTabPFN achieves a performance comparable to traditional machine learning baselines within one minute of pre-training on a single GPU in a small data setting.

2 Related Work

nanoTabPFN is inspired by the success of minGPT [Karpathy, 2020] and nanoGPT [Karpathy, 2022]. minGPT provides a minimal educational implementation of GPT-2 [Radford et al., 2019], while nanoGPT advances this work with a performance-centric reimplementation. nanoGPT is one of the most popular GPT-2 implementations, with over 47 000 stars on GitHub currently. It enabled many students and researchers to learn about large language models and bootstrap research, such as research on optimizers, including Muon [Jordan et al., 2024] or Adam-mini [Zhang et al., 2025b].

3 nanoTabPFN

nanoTabPFN consists of two parts: the model architecture and its training loop. In this section, we provide an in-depth explanation of both of these parts, as well as a small code example and a description of the differences to the original TabPFN v2 implementation.

3.1 Model Architecture

Figure 1 illustrates the architecture of nanoTabPFN, which consists of four parts: the `FeatureEncoder`, which normalizes the features and creates an embedding for each cell in the feature-part of the table; the `TargetEncoder`, which initializes the unknown test targets cells, and creates an embedding for each cell in the target-column of the table; multiple `TransformerEncoderLayers`, the main transformer layers adapting the embeddings of all cells in the table; and the `Decoder`, which maps the high-dimensional embedding of the unknown test targets to the predictions.

On an abstract level, TabPFN v2 works by alternating between applying attention between features and attention between datapoints on the table, as illustrated in Figure 2. To do this, we must first create an embedding for each cell in the table. We create the embeddings for all the features (`X_train` and `X_test`) using the `FeatureEncoder`. The `FeatureEncoder` normalizes each feature based on the mean and standard deviation of the training set, and removes outliers by clipping features that are too large or too small. Then it applies a linear layer to map the scalar values in the table to high-dimensional feature embeddings. Note that we use no positional embedding in our reimplementation since we want our model to be permutation-invariant with respect to datapoints and features. The `TargetEncoder` creates the embeddings for the target column; it extends `y_train` with its mean values to create entries for `y_test`, which we try to predict. Afterwards, it also applies a linear layer.

Now that we have an embedding for each cell in the table we apply multiple `TransformerEncoderLayers` sequentially, each of which applies bi-attention on the embeddings of the cells in our table (attention between features followed by attention between datapoints). In the

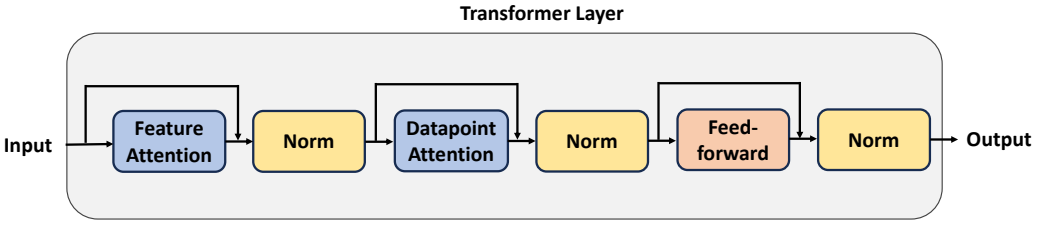

Figure 2: Transformer Layer. The Transformer Layer consists of Feature Attention, Datapoint Attention and a 2-layer MLP at the end. We have skip connections around each of the attention blocks and the MLP. A Layer Norm follows each skip connection.

attention between datapoints, the training data can only attend to itself and not the test data, while the test data can only attend to the training data. This restriction ensures that the test data is not attended to, thereby guaranteeing that adding or removing datapoints to `X_test` does not change the predictions for other datapoints.

The `TransformerEncoderLayer` has separate skip connections [He et al., 2016] around feature and datapoint attention, followed by layer normalizations [Ba et al., 2016]. After bi-attention the embeddings are further adapted by a cell-wise MLP, consisting of two feed-forward layers. This is surrounded by a skip connection and followed by a Layer Norm, see Figure 2.

The last part of our architecture is the `Decoder`, which takes the embeddings of the `y_test` cells as input and applies a 2-layer MLP. We treat the output of the MLP as logits for classification.

3.2 Training

We provide a simple training loop that trains the model on the data given by a dataloader, using the scheduler-free [Defazio et al., 2024] version of the AdamW [Loshchilov and Hutter, 2019] optimizer without weight decay. We support loading datasets that have been pre-generated and saved to the HDF5 [The HDF Group] format on disk. Enabling the loading of datasets from a file dump allows faster prototyping of the architecture and training code, since generating a new batch of data from a prior is quite expensive, and thus loading a pre-generated version drastically reduces the training time.

3.3 Code Example

On the right, we show a small example code for pretraining a 3-layer nanoTabPFN model on a pre-generated list of 80 000 datasets with exactly 150 datapoints, 5 features, and 2 classes. We later report results precisely for this code.

```python
from model import NanoTabPFNModel
from train import PriorDumpDataLoader
from train import train

model = NanoTabPFNModel(
    embedding_size=96,
    num_attention_heads=4,
    mlp_hidden_size=192,
    num_layers=3,
    num_outputs=2
)

prior = PriorDumpDataLoader(
    "300k_150x5_2.h5",
    num_steps=2500,
    batch_size=32,
)

model, _ = train(
    model,
    prior,
    lr=4e-3,
)
```

Figure 3: Code example showing how to train nanoTabPFN.

3.4 Differences to TabPFN v2

We intend nanoTabPFN to be an easier-to-understand and more lightweight version of TabPFN, and as such, we only include its core features. We do not include functionality of TabPFN that allows for combining neighboring pairs of features when creating the feature embedding, as it substantially increases code complexity, reduces interpretability and destroys permutation invariance of the features. Another feature we do not include adds a column filled with row hashes to the table to differentiate between datapoints. Finally, to keep our implementation minimal, we do not inherently

handle categorical features and missing values. If a user wants to evaluate nanoTabPFN on datasets containing categorical features or missing values, they have to preprocess them beforehand.

4 Results

We evaluate nanoTabPFN in a small data setting for educational use, demonstrating strong performance within minutes of pretraining.

We pretrained a small version of nanoTabPFN with 3 layers, 4 attention heads, an embedding size of 96, and a hidden layer size of 192 on 80 000 synthetically-generated datasets with exactly 150 datapoints and 5 features and 2 classes each, with a batch size of 32, using the example code from Figure 3. Training converged after one minutes on a single NVIDIA GeForce RTX 2080 Ti GPU (11GB VRAM), whereas TabPFN v2 was pretrained on eight of these GPUs for two weeks. This is more than 160 000× faster (14*24*60*8=161 280), substantially lowering the barrier to entry.

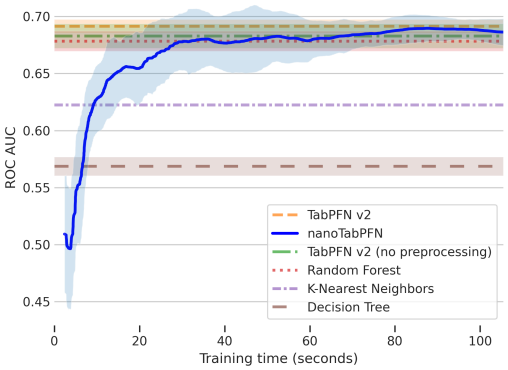

Figure 4: Within 60 seconds of pretraining on one consumer GPU, nanoTabPFN achieves average ROC AUC on a subset of subsampled datasets from TabArena comparable to traditional machine learning baselines.

The traditional baselines we use are k-nearest neighbors, a decision tree, and a random forest, in their default scikit-learn [Pedregosa et al., 2011] configuration. For TabPFN we evaluate two configurations, one is the default configuration and the other disables ensembling and pre-processing. The latter configuration more closely aligns with the feature set of nanoTabPFN, enabling a fairer comparison of the trained models rather than the surrounding pipelines. For more detailed information about our experimental setup and the evaluation strategy, including the datasets we used, we refer to Appendix A.

Figure 4 shows nanoTabPFN's average ROC-AUC over its training time. Within 60 seconds of training, nanoTabPFN reaches a higher ROC AUC than all traditional machine learning baselines, thereby demonstrating its effectiveness. After only a few more seconds, the similar scale of prior and the evaluation setting allowed nanoTabPFN to quickly learn in this restricted setting, outperforming the more generally trained TabPFN configuration without preprocessing. We present per-dataset results in Appendix B.

5 Conclusion

In this paper, we introduced nanoTabPFN, a small and lightweight implementation of the TabPFN v2 architecture. nanoTabPFN includes the core features of TabPFN, resulting in an implementation consisting of less than 500 lines of code, compared to the over 10 000 lines of code of the TabPFN repository. This allows us to train a model within minutes that performs comparably to traditional machine learning algorithms on small datasets. nanoTabPFN's fast training speed, combined with our lightweight implementation, enables researchers and students to understand the inner workings of TabPFN more easily and enables faster prototyping of new research ideas, in the same way minGPT and later on nanoGPT did for the space of large language models.

Since nanoTabPFN focuses on simplicity and educational value, we did not include all features of TabPFN v2 which results in performance limitations. For example, we do not include code for regression, handling missing values and ensembling across different pre-processings of the data. Despite these limitations we are able to achieve good performance on small datasets. We intentionally focus on small-scale data, as the repository is aimed at educational value with small-scale compute. Lastly, while nanoTabPFN's repository contains code for its architecture and training, it lacks a simplified prior implementation, which we leave to future work.

To conclude, with nanoTabPFN, we take an important first step towards democratizing the field of TFMs, lowering the barrier to entry, and accelerating research on TFMs. We look forward to nanoTabPFN being used in many university courses to teach TFMs.

Disclaimer

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

Acknowledgement

We thank the reviewers for their feedback, which contributed to improving this work. Additional we would also like to thank Jake Robertson for his input regarding our prior selection. This paper was supported by European Union's Horizon Europe research and innovation programme under grant agreement number 101214398 (ELLIOT). Johannes Hog and Lennart Purucker acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under SFB 1597 (SmallData), grant number 499552394. Frank Hutter acknowledges the financial support of the Hector Foundation.

References

Noah Hollmann, Samuel Müller, Katharina Eggensperger, and Frank Hutter. Tabpfn: A transformer that solves small tabular classification problems in a second, 2023. URL https://arxiv.org/abs/2207.01848.

Junwei Ma, Valentin Thomas, Rasa Hosseinzadeh, Hamidreza Kamkari, Alex Labach, Jesse C. Cresswell, Keyvan Golestan, Guangwei Yu, Maksims Volkovs, and Anthony L. Caterini. Tabdpt: Scaling tabular foundation models, 2024. URL https://arxiv.org/abs/2410.18164.

Jingang Qu, David Holzmüller, Gaël Varoquaux, and Marine Le Morvan. Tabicl: A tabular foundation model for in-context learning on large data. *arXiv preprint arXiv:2502.05564*, 2025.

Xingxuan Zhang, Gang Ren, Han Yu, Hao Yuan, Hui Wang, Jiansheng Li, Jiayun Wu, Lang Mo, Li Mao, Mingchao Hao, et al. Limix: Unleashing structured-data modeling capability for generalist intelligence. *arXiv preprint arXiv:2509.03505*, 2025a.

Noah Hollmann, Samuel Müller, Lennart Purucker, Arjun Krishnakumar, Max Körfer, Shi Bin Hoo, Robin Tibor Schirrmeister, and Frank Hutter. Accurate predictions on small data with a tabular foundation model. *Nature*, 637(8045):319–326, 2025.

Andrej Karpathy. minGPT. https://github.com/karpathy/minGPT, 2020.

Andrej Karpathy. nanoGPT. https://github.com/karpathy/nanoGPT, 2022.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P. Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more, 2025b. URL https://arxiv.org/abs/2406.16793.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Aaron Defazio, Xingyu Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. The road less scheduled, 2024.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

The HDF Group. Hierarchical Data Format, version 5. URL https://github.com/HDFGroup/hdf5.

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.

Nick Erickson, Lennart Purucker, Andrej Tschalzev, David Holzmüller, Prateek Mutalik Desai, David Salinas, and Frank Hutter. Tabarena: A living benchmark for machine learning on tabular data, 2025. URL https://arxiv.org/abs/2506.16791.

A Detailed Experimental Setup

Hyperparameter Optimization We have chosen our prior, model, and training configuration based on a random search where we sampled 200 configurations. The training time was limited to two minutes and we evaluated the performance on a synthetic prior consisting of 1600 datasets. For generating training and evaluation data, we relied on TabICL's prior implementation. The sampling was done on-the-fly and not included in the runtime measurement since we used pre-generated data when rerunning our best configuration. The search space and best configuration can be seen in Table 1. For our comparison in Section 4, we rounded the configuration values before training.

Table 1: Hyperparameter Search Space and Optimal Configuration

Hyperparameter	Search Space	Optimal Value
`lr`	$[10^{-4}, 5 \times 10^{-2}]$ (log scale)	0.003892
`weight_decay`	$[10^{-9}, 10^{-4}]$ (log scale)	1.00×10^{-7}
`effective_batch_size`	{8, 16, 32, 64}	32
`num_features`	[3, 13]	4
`num_datapoints_max`	[50, 300]	154
`num_attention_heads`	{2, 4, 8}	4
`embedding_size`	{64, 80, 96, 112, 128, 144, 160, 176, 192}	96
`mlp_multiple`	{2, 4}	2

Experimental Setup We limited our evaluation to the binary classification datasets, which contain no missing values, in TabArena [Erickson et al., 2025] with at most 10 features and subsampled the number of datapoints to 200. The evaluation employed stratified 5-fold cross-validation with 20 repetitions. We only measured the accumulated training time and excluded the evaluation time at each step. Our baselines used scikit-learn version 1.6.1 and tabpfn version 2.2.1.

B Additional Results

Figure 5 shows the ROC AUC during pretraining (Section 4) on the individual evaluation datasets.

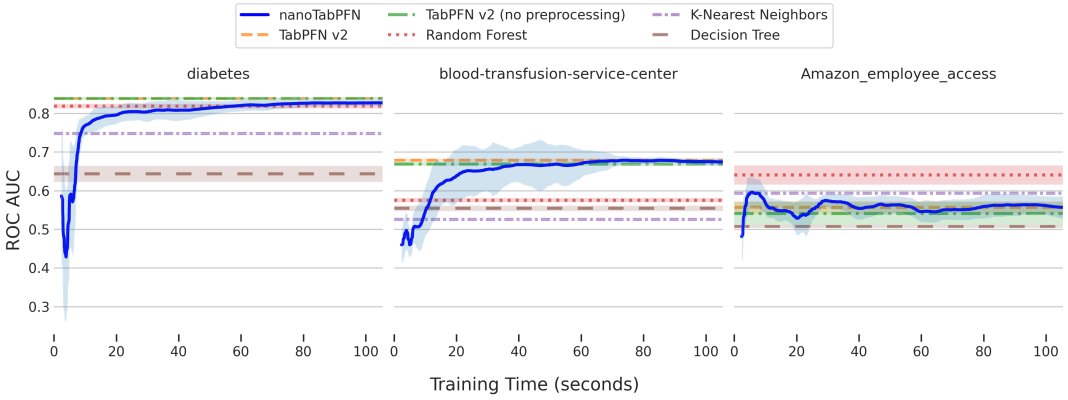

Figure 5: Per dataset results