PyTorch Frame: A Modular Framework for Multi-Modal Tabular Learning

Weihua Hu[1], **Yiwen Yuan**[1], **Zecheng Zhang**[1], **Akihiro Nitta**[1], **Kaidi Cao**[1], **Vid Kocijan**[1], **Jinu Sunil**[1], **Jure Leskovec**[1,2], **Matthias Fey**[1]

[1]Kumo AI, [2]Stanford University

Abstract

We present PyTorch Frame, a PyTorch-based framework for deep learning over multi-modal tabular data. PyTorch Frame makes tabular deep learning easy by providing a PyTorch-based data structure to handle complex tabular data, introducing a model abstraction to enable modular implementation of tabular models, and allowing external foundation models to be incorporated to handle complex columns (*e.g.*, LLMs for text columns). We demonstrate the usefulness of PyTorch Frame by implementing diverse tabular models in a modular way, successfully applying these models to complex multi-modal tabular data, and integrating our framework with PyTorch Geometric, a PyTorch library for Graph Neural Networks(GNNs), to perform end-to-end learning over relational databases.

1 Introduction

Deep learning has revolutionized many application domains, such as computer vision (He et al., 2016), natural language processing (Brown et al., 2020), audio processing (Oord et al., 2016), and graphs (Kipf & Welling, 2017). Yet, one critical domain that has yet to see big success is the *tabular domain*—a powerful and ubiquitous representation of data via heterogeneous columns. In the tabular domain, many existing studies (Shwartz-Ziv & Armon, 2022; Grinsztajn et al., 2022) have reported that Gradient-Boosted Decision Trees (GBDT) (Chen & Guestrin, 2016) is still a dominant paradigm.

However, GBDT has notable limitations. First, GBDT models are primarily focused on numerical and categorical features and cannot effectively handle raw multi-modal features, such as texts, sequences, images, and embeddings. Second, their end-to-end integration with downstream deep learning models, such as Graph Neural Networks (GNNs), is highly non-trivial since GBDT models are neither differentiable nor producing embeddings (Ivanov & Prokhorenkova, 2021). As such, GBDT falls short on complex applications, such as prediction over modern relational databases Fey et al. (2023).

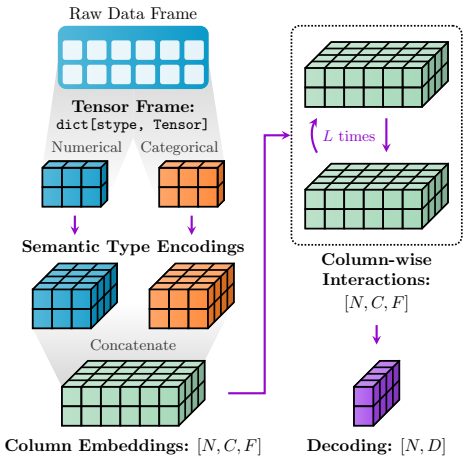

Figure 1: **Overview of PyTorch Frame's architecture**, consisting of a **(1)** Tensor Frame materialization stage, **(2)** semantic type-wise model encodings, **(3)** column-wise interaction blocks, and a final **(4)** readout decoder head.

Table Representation Learning Workshop at NeurIPS 2024

Tabular deep learning is a promising paradigm to resolve the challenges. In fact, the community has come up with many deep tabular models in an attempt to outperform GBDT (Huang et al., 2020; Gorishniy et al., 2021, 2022, 2024; Chen et al., 2023a; Arik Sercan O., 2021; Somepalli et al., 2021; Zhu et al., 2023; Popov et al., 2020; Abutbul et al., 2021; Chen et al., 2023b). While significant progress has been made, these models have only been evaluated on conventional numerical/categorical features. What is missing is a systematic exploration of model architectures and their capabilities in handling complex columns with general multi-modal data.

Here we introduce *PyTorch Frame*, a new PyTorch-based framework for tabular deep learning. Our goal is to facilitate research in tabular deep learning and realize its full potential. First, realizing the limited expressiveness of vanilla PyTorch to hold multi-modal data, we introduce *Tensor Frame*, an expressive Tensor-based data structure to handle arbitrary complex columns in an efficient way. Second, we introduce a general framework for learning on tabular data that abstracts the commonalities between the most promising existing deep learning models for tabular data. Our framework is illustrated in Figure 1 and shares a similar spirit to the message passing framework (Gilmer et al., 2017) that has propelled the field of graph learning. Given that many strong tabular models follow our general framework, we believe the community can further advance modeling with it more easily.

Under our framework, it is easy to incorporate external foundation models to handle complex multi-modal columns. They can be used to either generate embeddings or be finetuned end-to-end with deep tabular models. Moreover, models implemented with our framework can be easily integrated with other PyTorch models. For instance, by integrating with GNNs from PyTorch Geometric (Fey & Lenssen, 2019), we can achieve deep learning over relational databases (Fey et al., 2023). Finally, we demonstrate the usefulness of our framework by showing promising results on complex tabular data (*i.e.* multi-modal columns, multiple tables), in addition to conventional numerical/categorical datasets.

2 Related Work

Our framework follows the modular encoder-combiner-decoder framework (Molino et al., 2019), while being explicit about modeling multi-layer column interactions notable in modern deep tabular models (Chen et al., 2023a; Gorishniy et al., 2021; Chen et al., 2023b; Huang et al., 2020). Our framework is also related to PyTorch Tabular (Joseph, 2021), an open-source tabular learning framework built on top PyTorch. While PyTorch Tabular has primarily focused on supporting existing tabular models, our PyTorch Frame offers enhanced flexibility for exploring and building novel tabular learning approaches while still providing access to established models. PyTorch Frame further distinguishes itself through support for a wider variety of column modalities and streamlined integration with LLMs.

3 PyTorch Frame

PyTorch Frame [1] provides a unified framework for efficient deep learning over tabular data $\mathbf{T} = [(v_1, \ldots, v_C)]_{n=1}^{N}$, which holds data across C columns for every of its N rows. We denote $T[i, j]$ as the raw value of column j in row i. We also use standard NumPy notations (Harris et al., 2020), such as $T[:, j]$, $T[i, :]$, $T[[i_1, ..., i_k], :]$, and $T[:, [j_1, ..., j_k]]$.

Semantic Type. Modern tabular data is complex, consisting of a variety of multi-modal columns. To effectively handle such data, PyTorch Frame introduces a *semantic type* that specifies the "modality" of each column. A variety of semantic types are supported to handle diverse columns, such as:

- `numerical` type can be used to handle numerical values, such as price and age columns.
- `categorical` type can be used to handle categorical values, such as gender and educational-level columns.
- `multicategorical` type can be used to handle multi-hot categories, such as a movie genres columns.
- `timestamp` type can be used to handle time columns, such as columns storing the date of events.

[1] `https://github.com/pyg-team/pytorch-frame`

- Both `text_embedding` and `text_tokenized` types can be used to handle text data, such as columns storing product descriptions. The former pre-encode text into embedding vectors, while the latter enables fine-tuning text model parameters.
- `embedding` type can be used to handle columns storing embedding data, such as pre-computed image embedding vectors.

Given tabular data $\mathbf{T}$, PyTorch Frame assumes a semantic type being specified for each of C columns. It can be either inferred based on some heuristics or manually specified by users. We let $\phi(s)$ denote the mapping from a semantic type $s \in \mathcal{S}$ to the list of column indices specified as s.

As shown in Figure 1, PyTorch Frame learns representation vectors of $\mathbf{T}$ in the following four stages:[2]

1. **Materialization** groups column data according to their semantic type and converts the grouped raw values $\mathbf{T}[:, \phi(s)]$ into a tensor-friendly data $\mathbf{F}_s$ of shape $[N, |\phi(s)|, *]$, where the last dimension $*$ depends on the specific semantic type s. We refer the dictionary $\{s : \mathbf{F}_s\}_{s \in \mathcal{S}}$ as a *Tensor Frame* representation of $\mathbf{T}$.
2. **Encoding** independently embeds each column value into a F-dimensional vector. Specifically, for each semantic type s, it embeds input tensor data $\mathbf{F}_s$ of shape $[N, |\phi(s)|]$ into embedding $\mathbf{X}_s$ of shape $[N, |\phi(s)|, F]$. Then, it concatenates $\{\mathbf{X}_s\}_{s \in \mathcal{S}}$ to obtain the column embedding vector $\mathbf{X}$ of shape $[N, C, F]$.
3. **Column-wise Interaction** performs multiple layers of column-wise message passing to enrich each column's representation by the knowledge of other columns. For each layer $\ell = 0, ..., L-1$, we update the embedding of column j as follows for each row i:

$$\mathbf{X}^{(\ell+1)}[i, j, :] \leftarrow f_\theta \left(\mathbf{X}^{(\ell)}[i, j, :], \{\mathbf{X}^{(\ell)}[i, c, :]\}_{1 \leq c \leq C} \right), \tag{1}$$

   where $\mathbf{X}^{(0)} \leftarrow \mathbf{X}$. The last-layer column embedding $\mathbf{X}^{(L)}[i, :, :]$ for each row i captures high-order interactions among columns within the row.
4. **Decoding** summarizes the last-layer column embeddings $\mathbf{X}^{(L)}[i, :, :]$ to obtain row embeddings $\mathbf{Z}[i, :] = g_\theta(\mathbf{X}^{(L)}[i, :, :])$ of shape $[D,]$, where D is the output dimensionality. The output row embedding $\mathbf{Z}$ can be either sent directly to a prediction head for row-wise prediction or used as input to downstream deep learning models, such as GNNs.

Usually, the materialization is performed as a pre-processing step, and the subsequent three stages have parameters to be learned during training. In what follows, we describe each step in more detail.

3.1 Materialization

Data materialization takes care of converting the raw input data in $\mathbf{T}$ into a *Tensor Frame*, a tensor-friendly format that can be efficiently processed in a deep learning pipeline.

The key step is to transform $\mathbf{T}[:, \phi(s)]$ with N rows and $|\phi(s)|$ columns into a tensor data $\mathbf{F}_s$ of shape $[N, |\phi(s)|, *]$, for each semantic type $s \in \mathcal{S}$. Below we show examples for some representative semantic types.

`numerical`. $\mathbf{T}[:, \phi(s)]$ is already in numerical form, so it can be directly transformed into a standard floating tensor $\mathbf{F}_s$ of shape $[N, |\phi(s)|]$. We model missing values as `NaN`.

`categorical`. $\mathbf{T}[:, \phi(s)]$ usually consists of strings, *e.g.*, "male", "female", "non-binary" in the case of a gender column. For each column, we map elements into an non-negative contiguous indices, *e.g.*, "male"$\mapsto$ 0, "female"$\mapsto$ 1, "non-binary"$\mapsto$ 2. Applying this fixed mapping, we transform $\mathbf{T}[:, \phi(s)]$ into a standard integer tensor $\mathbf{F}_s$ of shape $[N, |\phi(s)|]$. We model missing values as -1.

`multicategorical`. Each cell in $\mathbf{T}[:, \phi(s)]$ consists of a list of multiple categories, *e.g.*, ["comedy", "romance", "drama"] for a movie genre column. We can similarly map each category into an integer index *e.g.*, "comedy"$\mapsto$ 0, "romance"$\mapsto$ 1, "drama"$\mapsto$ 2 so that each cell in $\mathbf{T}[:, \phi(s)]$ can be mapped to a list of integers, *e.g.*, [0, 1, 2] in the case above. The challenge in the implementation lies in the varying sizes of the lists for different cells. To handle such data, PyTorch Frame supports its own tensor format called `MultiNestedTensor` based on a *ragged tensor layout* as illustrated in Figure 2. We use it to transform $\mathbf{T}[:, \phi(s)]$ into `MultiNestedTensor` $\mathbf{F}_s$ of shape $[N, |\phi(s)|, \cdot]$.

[2] In practice, we consider a mini-batch of rows $\mathbf{T}[[i_1, ..., i_k], :]$, but its extension is straightforward.

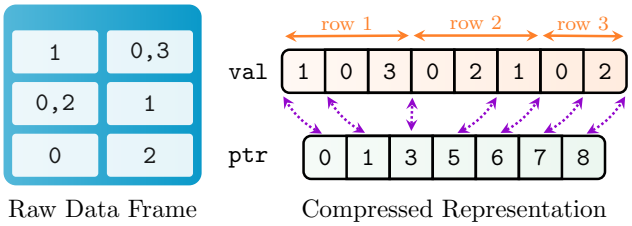

Figure 2: `MultiNestedTensor` **based on compressed ragged tensor layout.** Our ragged layout describe tensors of shape $[N, C, \cdot]$, where the size of the last dimension can vary across both rows and columns. Internally, data is stored in an efficient compressed format `(val, ptr)`, where `val` holds data in a flattened vector and `ptr` holds cumulated offsets of rows and columns. $T[i, j]$ can be accessed via `val[ptr[C*i+j]:ptr[C*i+j+1]]`, which allows for efficient slicing and indexing along the row dimension.

`text_tokenized`. Each cell of $\mathbf{T}[:, \phi(s)]$ consists of a piece of text, which can be tokenized into a list of integers of varying length. Hence, we can transform $\mathbf{T}[:, \phi(s)]$ into `MultiNestedTensor` $\mathbf{F}_s$ of shape $[N, |\phi(s)|, \cdot]$, similar to `multicategorical`.

`text_embedded`. Similar to `text_tokenized`, each cell of $\mathbf{T}[:, \phi(s)]$ consists of a piece of text. Different from `text_tokenized`, we use external text embedding models (Neelakantan et al., 2022; Reimers & Gurevych, 2019) to pre-compute text vectors. Concretely, each single column $\mathbf{T}[:, j], j \in \phi(s)$ is pre-encoded by a column-specific text model into an embedding tensor of shape $[N, 1, D_j]$, where D_j can be different for different j. To handle multiple text columns simultaneously, PyTorch Frame introduces its own `MultiEmbeddingTensor` layout, where each column of $\mathbf{T}[:, \phi(s)]$ is pre-embedded into D_j-dimensional vectors, which are stacked to produce `MultiEmbeddingTensor` $\mathbf{F}_s$ of shape $[N, |\phi(s)|, \mathbf{D}]$. We also support `image_embedded` in the similar way, whereas each cell contains the path of the image data.

`embedding`. A table may contain pre-computed embeddings, such as those created by other teams (Hu et al., 2022). Specifically, $\mathbf{T}[:, j], j \in \phi(s)$ directly stores D_j-dimensional embeddings. Similar to `text_embedded`, we can transform $\mathbf{T}[:, \phi(s)]$ into a `MultiEmbeddingTensor` $\mathbf{F}_s$ of shape $[N, |\phi(s)|, \mathbf{D}]$.

In summary, PyTorch Frame uses specialized tensor-based data structures to efficiently handle complex tabular data with different semantic types. In addition, the materialization stage computes basic statistics for each column, such as mean and standard deviation for `numerical` columns, or the count of category elements for `categorical` and `multicategorical` columns. These statistics are stored and supplied to the subsequent encoding stage to normalize data or impute missing values.

3.2 Encoding

PyTorch Frame encoders receive a Tensor Frame with N rows as input[3] and map their columns into a shared embedding space $\mathbf{X}$ of shape $[N, C, F]$. All columns within the same semantic type are embedded in parallel, ensuring maximum throughput. More concretely, for each semantic type s, its tensor data $\mathbf{F}_s$ of shape $[B, |\phi(s)|, *]$ is embedded into $\mathbf{X}_s$ of shape $[B, |\phi(s)|, F]$, where F is the dimensionality of column embeddings. Then, $\{\mathbf{X}_s\}_{s \in \mathcal{S}}$ are concatenated to produce the final shared embedding $\mathbf{X}$ of shape $[N, C, F]$. In mapping to $\mathbf{X}_s$, encoders perform feature normalization and column embeddings, as detailed below.

Feature normalization. The tensor data $\mathbf{F}_s$ can contain missing and values with arbitrary scales, making them not suitable as input to machine learning models. To resolve these issues, encoders first normalize the features based on the statistics calculated from the materialization stage. As an example, for `numerical` type, one can impute missing values with the mean value. Then, each column can be normalized to have zero-mean and unit-variance. The feature normalization is performed at the encoding stage (instead of at the materialization stage), which allows our users to test various imputation/normalization strategies without the need to re-materialize the data.

[3]In mini-batch training, they receive a Tensor Frame of $B \leq N$ rows.

Column Embeddings. After feature normalization, the encoders embed $\mathbf{F}_s$, representing $|\phi(s)|$ columns, into F-dimensional column embeddings $\mathbf{X}_j$ of shape $[N, |\phi(s)|, F]$. Different modeling choices are possible. For example, `numerical` columns can either be transformed either via a linear layer or can be first converted into piecewise linear or periodic representations Gorishniy et al. (2022) before a linear layer. For `categorical` columns, one can transform them via shallow embeddings learnable for each category. The `text_tokenized` columns can be transformed into embeddings via language models that take the sequences of tokens as input.

3.3 Column-wise Interaction

Given the column embeddings $\mathbf{X}$, where all columns are embedded in a shared F-dimensional embedding space, we proceed to model the interactions between columns in the embedding space. Specifically, an embedding of each column are iteratively updated based on those of the other columns, as shown in Eq. 1. After L iterations, we obtain $\mathbf{X}^{(L)}$ of shape $[N, C, F]$, capturing higher-order interactions among column values within each row.

Many existing works can be cast under this framework. For example, Gorishniy et al. (2021) applied a permutation-invariant Transformer (Vaswani et al., 2017) to model column interactions, while Huang et al. (2020) used a Transformer with positional column encoding. Chen et al. (2023a) also followed a Transformer architecture except that it sorts the features by mutual information and used diagonal attention in the Transformer block. Chen et al. (2023b) used cross attention between column embeddings and learnable prompt embeddings to model column interactions.

3.4 Decoding

Finally, we apply a decoder on $\mathbf{X}^{(L)}$ to obtain D-dimensional row-wise embedding $\mathbf{Z}$ of shape $[N, D]$, which can be directly used for prediction over tabular rows or as input to subsequent deep learning models.

The decoder can be, *e.g.*, a weighted sum of column embeddings, where the weights are either uniform or learned attention weights (Chen et al., 2023b). Huang et al. (2020) modeled the decoder by applying an MLP over the flattened column embeddings of length $C \times F$. Gorishniy et al. (2021) added a "CLS" column embedding (Devlin et al., 2018) in $\mathbf{X}$ and directly read out "CLS" column embeddings in $\mathbf{X}^{(L)}$, similar to BERT (Devlin et al., 2018).

3.5 Accommodating Diverse Tabular Models

While our abstraction framework covers many existing tabular models, not all models fit within our framework. Some models are simple to accommodate, *e.g.*, ResNet (Gorishniy et al., 2021) does not have the column-wise interaction stage, so we can simply omit the stage. Other models are harder to accommodate. For example, TabNet (Arik Sercan O., 2021) operates on 2-dimensional tensors (instead of the 3-dimensional tensor layout of $\mathbf{X}$) and applies a series of attention-based transformation over it. Notably, those models can still be implemented by taking Tensor Frame as input; PyTorch Frame supports those models by directly implementing the model architecture without following the modular framework.

4 Integration

Integration with Foundational Models. For modeling complex columns like text and images, it is best to incorporate external large pre-trained foundation models. PyTorch Frame supports seamless integration with external models via semantic types like `text_embedded`, `text_tokenized`, and `image_embedded`.

For example, for `text_embedded`, users only need to specify embedding models to map a list of texts into embedding tensors, which can be achieved via the OpenAI embedding API[4] or any sentence transformer (Reimers & Gurevych, 2019). Then, at the materialization stage, PyTorch Frame automatically applies the embedding models to generate a Tensor Frame with text embeddings. Note that the materialization can be expensive since it uses LLMs to embed all text columns. To avoid

[4] `https://platform.openai.com/docs/guides/embeddings`

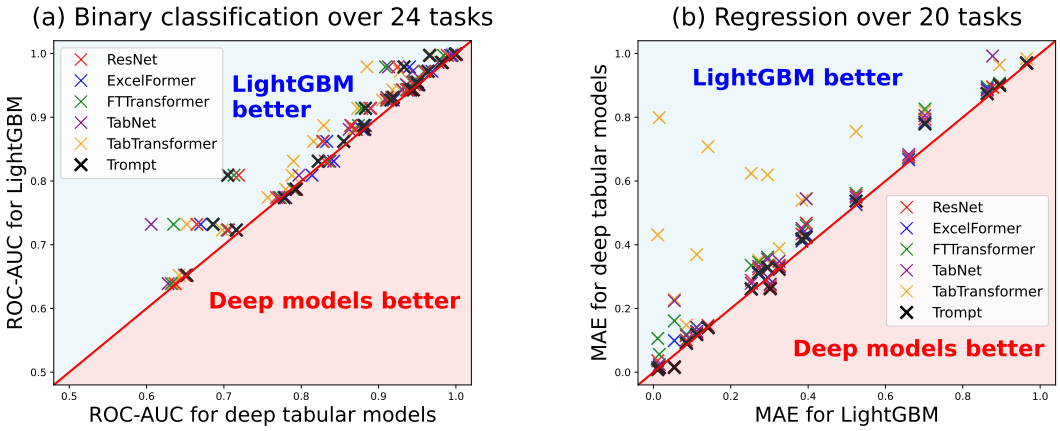

Figure 3: **Scatter plot comparison between deep tabular models and LightGBM across datasets with *only numerical and categorical features*.** Here each "x" represents a single predictive task, and its position represents the predictive performance of a deep tabular model compared against LightGBM. When "x" lies above (resp. below) the diagonal line, it means the LightGBM outperforms (resp. underperforms) the corresponding deep tabular model on the respective task. Overall, LightGBM is still dominating the existing deep tabular models on the conventional numerical/categorical datasets, although the recent Trompt model (Chen et al., 2023b) is getting close.

repeated materialization, PyTorch Frame *caches* the materialized data. The cached Tensor Frame can be reused in subsequent runs, avoiding expensive re-materialization.

Integration with PyTorch Geometric. We have so far discussed single-table tabular learning, but many practical applications involve data stored in a *relational format* (Codd, 1970), where tabular data is connected with each other via primary-foreign key relationships. Combining tabular deep learning with Graph Neural Networks (GNNs) has proven to be promising to handle such relational datasets Fey et al. (2023). PyTorch Frame integrates natively with PyTorch Geometric (PyG) Fey & Lenssen (2019), a popular PyTorch library for GNNs. PyTorch Frame enhances PyG by learning embedding vectors of nodes and edges with complex multi-modal features. The node and edge embeddings are subsequently fed as input to GNNs by PyG. Crucially, tabular deep learning models by PyTorch Frame and GNNs by PyG can be jointly trained to optimize for downstream task performance.

5 Experimental Study

Here we demonstrate the usefulness of PyTorch Frame in handling conventional single-table data as well as more complex tabular data with text columns and relational structure.

5.1 Handling single-table data

First, we focus on the traditional tabular machine learning setting with only numerical and categorical columns. We collected datasets from diverse resources (Grinsztajn et al., 2022; Gorishniy et al., 2021; Blake, 1998), totalling 23 tasks for binary classification and 19 tasks for regression. Following Hu et al. (2020), we make all these datasets and and their data split available through PyTorch Frame package so that it is easy to compare models in a standardized manner.

Using PyTorch Frame, we implemented six deep tabular models: ResNet (Gorishniy et al., 2021), ExcelFormer (Chen et al., 2023a), FTTransformer (Gorishniy et al., 2021), TabNet (Arik Sercan O., 2021), TabTransformer (Huang et al., 2020). PyTorch also seamlessly integrated GBDT models, XGBoost (Chen & Guestrin, 2016), CatBoost (Prokhorenkova et al., 2018), and LightGBM (Ke et al., 2017), that operate on Tensor Frame. For each model, we used Optuna to perform a hyper-parameter search with 20 trials (Akiba et al., 2019).

Figure 3 shows the comparison between each of the six deep learning models and LightGBM, which we found to perform the best among the GBDT models. Similar to previous studies (Shwartz-Ziv & Armon, 2022; Grinsztajn et al., 2022), we found that deep tabular models are coming close to

Table 1: **Results on tabular datasets with text columns.** We report binary classification ROCAUC and training time (excluding text pre-encoding). For each dataset, bolded values represent the best in each category, with * indicating the best overall. Details of text models: ♠ `all-roberta-large-v1` (Sentence Transformer) (Reimers & Gurevych, 2019). ♣ `text-embedding-3-large` (OpenAI) (Neelakantan et al., 2022). ♦ RoBERTa-large (Liu et al., 2019). ♥ best model from Shi et al. (2020) (RoBERTa-large or ELECTRA (Clark et al., 2020)). For LightGBM†, text embeddings are treated as numerical features.

Method		**fake**		**jigsaw**		**kick**	
Text Model	**Tabular Model**	ROC-AUC	Time	ROC-AUC	Time	ROC-AUC	Time
`RoBERTa`♠	ResNet	0.934	7.3s	0.883	36.1s	0.753	27.8s
	FTTransformer	0.936	19.7s	0.882	100.8s	0.747	77.4s
	Trompt	**0.958**	18.8s	**0.885**	480.4s	0.756	581.9s
	LightGBM†	0.954	15.5s	0.865	571.1s	**0.767**	1931.9s
`OpenAI`♣	ResNet	0.923	10.4s	0.945	56.5s	0.807	107.1s
	FTTransformer	0.911	23.6s	0.945	337.4s	0.807	168.9s
	Trompt	**0.976**	40.8s	**0.947**	4285.1s	**0.810***	538.0s
	LightGBM†	0.966	131.0s	0.926	1732.9s	0.809	1924.3s
`RoBERTa`♦	ResNet	**0.979***	5.5h	**0.970***	>1 day	**0.786**	>1 day
	FTTransformer	0.960	5.5h	0.968	>1 day	0.775	>1 day
Best single model (Shi et al., 2021)♥		0.967	-	0.967	-	0.794	-

LightGBM, but not outperforming it. Among the six deep tabular models, we found the Trompt model to give the closest performance to LightGBM, but Trompt is also the most expensive deep tabular model with nearly 100 to 1000 times more training time compared to LightGBM even with GPU. Given the simplicity and efficiency of GBDT models, they may remain a practical choice for conventional tabular learning datasets.

Next, we shift our evaluation to more modern tabular datasets that come with text columns and multiple tables.

5.2 Handling text data

In this section, we demonstrate the capability of PyTorch Frame in utilizing external text models to achieve strong performance on tabular datasets with text columns.

PyTorch Frame provides two options to handle text columns: `text_embedded` and `text_tokenized`. The `text_embedded` option pre-encodes text into embedding vectors at the materialization stage, while `text_tokenized` option only tokenizes text during materialization, allowing text models to be jointly trained with deep tabular models at training time.

For `text_embedded`, we consider two kinds of text embedding models: The `all-roberta-large-v1` model from the Sentence Transformer (Liu et al., 2019; Reimers & Gurevych, 2019) and the more recent OpenAI embedding model, `text-embedding-3-large`, available through API (Neelakantan et al., 2022).[5] For `text_tokenized`, we used the original RoBERTa-large model (Liu et al., 2019), to align with the setting in Shi et al. (2021). We trained strong deep tabular models[6] and LightGBM to make the final label prediction. The hyper-parameters of LightGBM are tuned with Optuna (Akiba et al., 2019) with 3 trials, while those of deep tabular models are tuned manually.

The results are shown in Table 1. Overall, we find that the best results from PyTorch Frame significantly improve over the best single-model results from Shi et al. (2021), demonstrating the promise of PyTorch Frame in handling tabular data with text columns.

[5]Note that we need to be aware that the OpenAI embedding model may be trained on the experimented tabular data.

[6]We did not include the Trompt model in our `text_tokenized` experiment since the model architecture requires applying the text model in each layer, which is very GPU memory intensive.

Table 2: **Results on multi-tabular relational datasets**.

Method	rel-stackex-engage ROCAUC ↑	rel-stackex-votes MAE ↓
LightGBM	0.618	0.422
PyG-HeteroSAGE+ PyTorch Frame-ResNet	**0.854**	**0.373**

Comparing among models with `text_embedded`, we see clear benefit of using the advanced OpenAI embedding model as opposed to the less advanced `RoBERTa` model. Moreover, the Trompt model often provides the best performance among the tabular models, collaborating our finding in Section 5.1.

Comparing between `text_embedded` and `text_tokenized` options with the same base text model (*i.e.*, RoBERTa), we see that `text_tokenized` gives a substantially better predictive performance. This is expected since `text_tokenized` allows the text model to be specifically fine-tuned on predictive tasks of interest. However, `text_tokenized` is orders-of-magnitude slower than `text_embedded` due to the expensive fine-tuning of text models at training time. Nonetheless, by using the more advanced OpenAI embeddings, `text_embedded` gives significantly better performance that is comparable to that of the `text_tokenized` option, while being much faster than `text_tokenized` in terms of training time. With the faster and cheaper text embedding API available, the `text_embedded` option becomes a promising choice to achieve good performance on tabular datasets with text columns.

5.3 Handling relational data

Finally, we show the benefit of tabular deep learning by integrating PyTorch Frame models with PyG (Fey & Lenssen, 2019) to make predictions over relational databases.

We consider `rel-stackex`, a Stack Exchange dataset from Fey et al. (2023). It consists of 7 tables that store users, posts, comments, votes, post links, badge records, and post history records. Within the dataset, two practically relevant prediction tasks are defined. The `rel-stackex-engage` aims to predict if the user will make any contribution, defined as vote, comment, or post, in the next 2 years. The `rel-stackex-votes` task aims to predict the popularity of a question post in the next 2 years, where the popularity is defined as the number of upvotes the post will receive. `rel-stackex-engage` is a binary classification task, while `rel-stackex-votes` is a regression task.

Following the relational deep learning approach (Fey & Lenssen, 2019), we use deep tabular models to encode table rows into node embeddings, which are then fed into GNNs to update the embeddings based on primary-foreign key relations. Crucially, the deep tabular models and GNNs are *jointly trained* to optimize for the task performance. As a specific instantiation, we adopted ResNet from PyTorch Frame for row encoding and heterogeneous GraphSAGE from PyG for updating node embeddings. We compare our model against a LightGBM that is trained on a single table data. As we see in Table 2, the relational deep learning approach enabled by the combination of PyTorch Frame and PyG provides superior performance compared to LightGBM that can be only trained on single-table data.

6 Conclusions

We presented PyTorch Frame to facilitate deep learning research on tabular data. We introduced Tensor Frame, a new tensor-based data structure to efficiently handle multi-modal tabular data. Then, we built a general model abstraction on top of Tensor Frame and implemented state-of-the-art deep tabular models under the modular framework. We empirically demonstrate the usefulness of PyTorch Frame on modern tabular learning settings involving text columns and multiple tables. Overall, we hope PyTorch Frame helps pushing tabular deep learning to enable accurate prediction over complex multi-modal tabular data.

References

Abutbul, A., Elidan, G., Katzir, L., and El-Yaniv, R. Dnf-net: A neural architecture for tabular data. In *International Conference on Learning Representations (ICLR)*, 2021.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 2623–2631, 2019.

Arik Sercan O., Pfister, T. TabNet: Attentive interpretable tabular learning. In *AAAI Conference on Artificial Intelligence*, 2021.

Blake, C. L. Uci repository of machine learning databases. *http://www. ics. uci. edu/˜ mlearn/ML-Repository. html*, 1998.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. volume 33, pp. 1877–1901, 2020.

Chen, J., Yan, J., Chen, D. Z., and Wu, J. Excelformer: A neural network surpassing gbdts on tabular data. *arXiv preprint arXiv:2301.02819*, 2023a.

Chen, K.-Y., Chiang, P.-H., Chou, H.-R., Chen, T.-W., and Chang, D. T.-H. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning (ICML)*, 2023b.

Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 785–794, 2016.

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, 13 (6):377–387, 1970.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Fey, M., Hu, W., Huang, K., Lenssen, J. E., Ranjan, R., Robinson, J., Ying, R., You, J., and Leskovec, J. Relational deep learning: Graph representation learning on relational databases. *arXiv preprint arXiv:2312.04615*, 2023.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, pp. 1273–1272, 2017.

Gorishniy, Y., Ivan, R., Khrulkov, V., and Babenko, A. Revisiting deep learning models for tabular data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Gorishniy, Y., Ivan, R., and Babenko, A. On embeddings for numerical features in tabular deep learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Gorishniy, Y., Rubachev, I., Kartashev, N., Shlenskii, D., Kotelnikov, A., and Babenko, A. Tabr: Tabular deep learning meets nearest neighbors. In *International Conference on Learning Representations (ICLR)*, 2024.

Grinsztajn, L., Oyallon, E., and Varoquaux, G. Why do tree-based models still outperform deep learning on typical tabular data? volume 35, pp. 507–520, 2022.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. Array programming with numpy. *Nature*, 585(7825): 357–362, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Hu, W., Bansal, R., Cao, K., Rao, N., Subbian, K., and Leskovec, J. Learning backward compatible embeddings. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 3018–3028, 2022.

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

Ivanov, S. and Prokhorenkova, L. Boost then convolve: Gradient boosting meets graph neural networks. *arXiv preprint arXiv:2101.08543*, 2021.

Joseph, M. Pytorch tabular: A framework for deep learning with tabular data. *arXiv preprint arXiv:2104.13638*, 2021.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. volume 30, 2017.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Molino, P., Dudin, Y., and Miryala, S. S. Ludwig: a type-based declarative deep learning toolbox. *arXiv preprint arXiv:1909.07930*, 2019.

Neelakantan, A., Xu, T., Puri, R., Radford, A., Han, J. M., Tworek, J., Yuan, Q., Tezak, N., Kim, J. W., Hallacy, C., et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. WaveNet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

Popov, S., Morozov, S., and Babenko, A. Neural oblivious decision ensembles for deep learning on tabular data. In *International Conference on Learning Representations (ICLR)*, 2020.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., and Gulin, A. Catboost: unbiased boosting with categorical features. volume 31, 2018.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. 11 2019. URL `https://arxiv.org/abs/1908.10084`.

Shi, X., Mueller, J., Erickson, N., Li, M., and Smola, A. J. Benchmarking multimodal automl for tabular data with text fields. *arXiv preprint arXiv:2111.02705*, 2021.

Shi, Y., Huang, Z., Wang, W., Zhong, H., Feng, S., and Sun, Y. Masked label prediction: Unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509*, 2020.

Shwartz-Ziv, R. and Armon, A. Tabular data: Deep learning is not all you need. *Information Fusion*, 81:84–90, 2022.

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Zhu, B., Shi, X., Erickson, N., Li, M., Karypis, G., and Shoaran, M. Xtab: Cross-table pretraining for tabular transformers. *arXiv preprint arXiv:2305.06090*, 2023.