Multi-Modal Tabular Deep Learning with PyTorch Frame

Akihiro Nitta
akihiro@kumo.ai
/pyg-team/pytorch-frame

```
pip install pytorch-frame
```

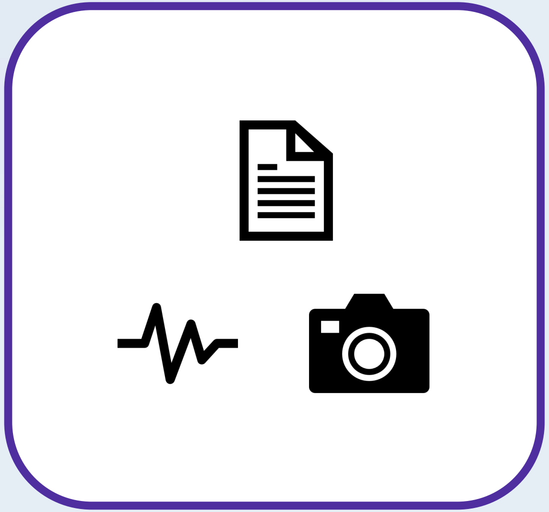
kumo

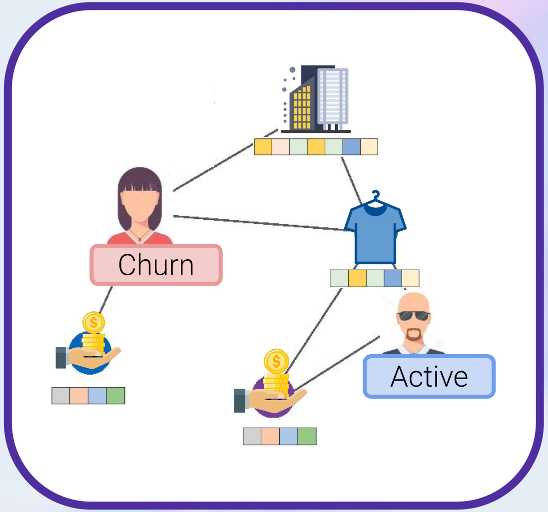
Table of Contents

kumo

Tree-based models are dominant – with limitations

- GBDTs are focused on numerical and categorical features
  - Modern tabular data have text and images
- Integrating GBDT with deep learning models is non-trivial
  - training tabular models with image and text encoders
  - training recommendation models with GNNs end-to-end





kumo

PyTorch Frame

PyTorch Frame

A modular framework for tabular learning

✅ Multi-modal features
numerical, categorical, multi-categorical,
image, text, embedding, timestamp

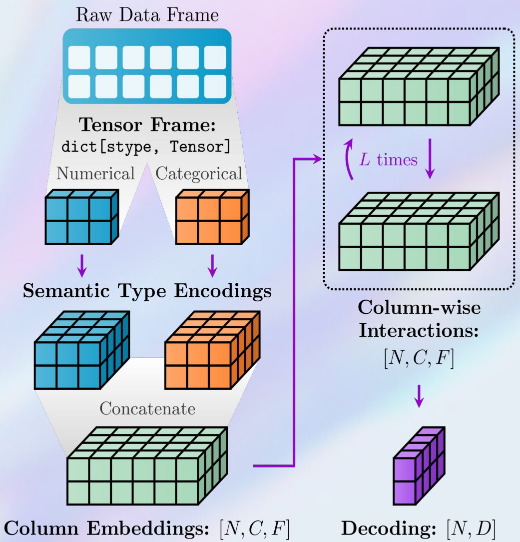
✅ Modular design covering various existing models
e.g., Trompt, ExcelFormer, FT-Transformer

✅ Integration with foundation models & GNNs

🎉 Best Paper Award at TRL@NeurIPS 2024 🇨🇦

```
pip install pytorch-frame
```

pyg-team/pytorch-frame



kumo

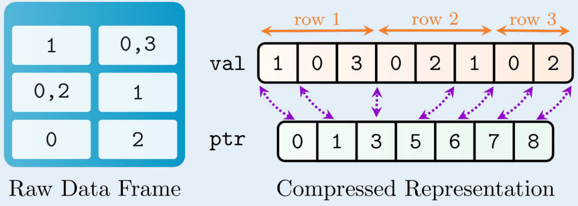
Data Materialization

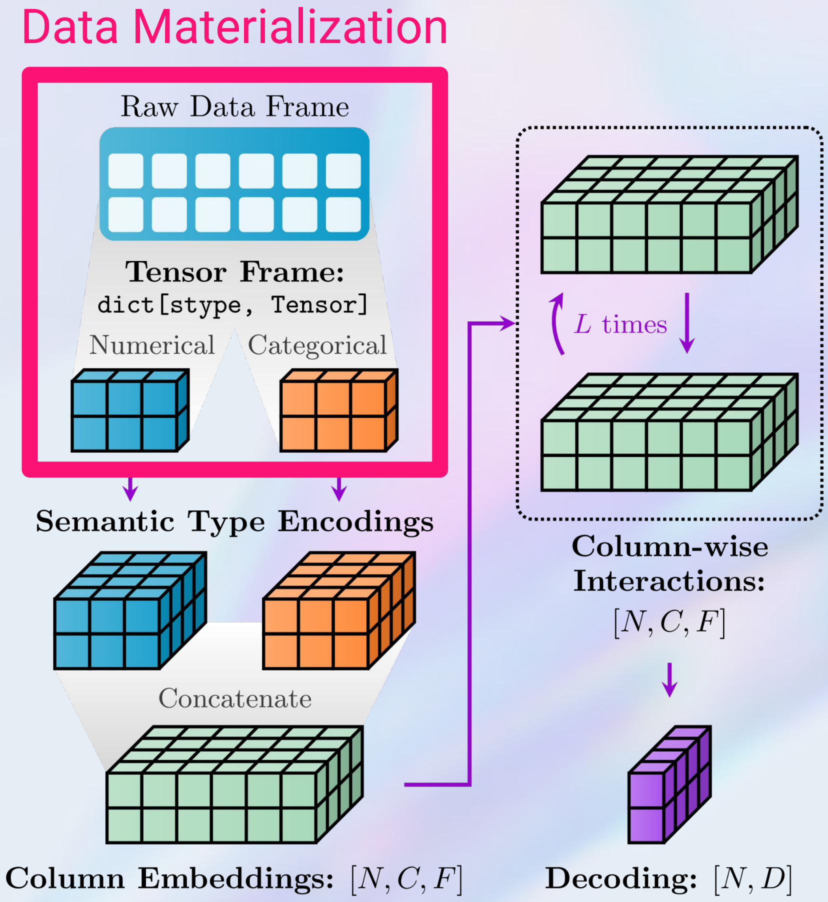
During data materialization, PyTorch Frame:

- **encodes a raw data frame** into a `TensorFrame`
- **computes statistics**, *e.g.*, mean, standard deviation, count of category elements

`TensorFrame` is a tensor-based data structure

- stores columns of the same type in a tensor
- stores sparse features efficiently, *e.g.*, via `MultiNestedTensor`





kumo

Data Materialization and Semantic Types

PyTorch Frame materializes columns of different *semantic types* as follows:

- `numerical` – Pass through of floating-point columns
- `categorical` – Map categories to indices
- `multicategorical` – Map multiple categories into indices of varying length
- `timestamp` – Map timestamps to integers of year, month, day, hour, minute, ...
- `embedding` – Pass through of pre-computed embeddings
- `text_tokenized` – Tokenize text into a list of integers of varying length
- `text_embedded` – Pre-compute text vectors via external text models
- `image_embedded` – Pre-compute image vectors via external image models

Native Integration with Foundation Models

OpenAI cohere

Hugging Face VOYAGE AI

kumo

Data Materialization and Semantic Types

PyTorch Frame materializes columns of different *semantic types* as follows:

- `numerical` – Pass through of floating-point columns
- `categorical` – Map categories to indices
- `multicategorical` – Map multiple categories into indices of varying length
- `timestamp` – Map timestamps to integers of year, month, day, hour, minute, …
- `embedding` – Pass through of pre-computed embeddings
- `text_tokenized` – Tokenize text into a list of integers of varying length
- `text_embedded` – Pre-compute text vectors via external text models
- `image_embedded` – Pre-compute image vectors via external image models

Native Integration with Foundation Models

OpenAI cohere Hugging Face VOYAGE AI

kumo

Data Materialization and Text Columns

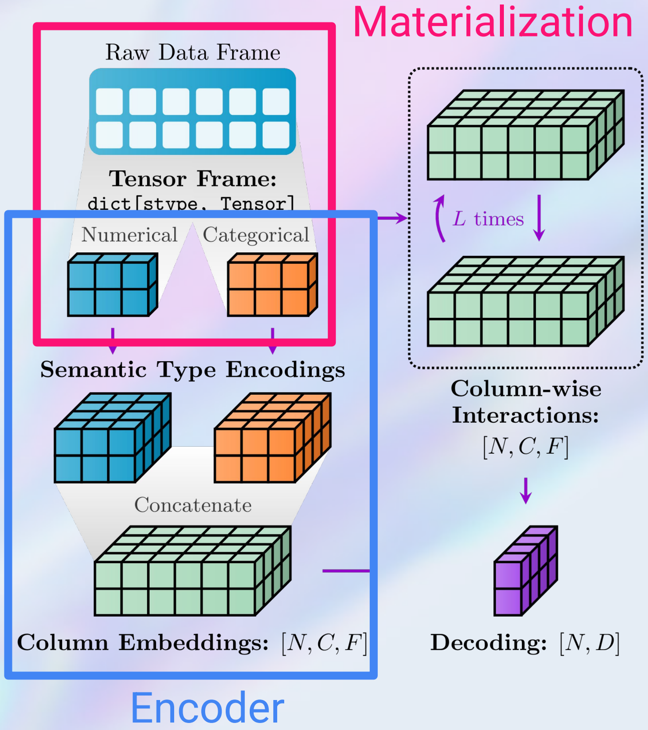
PyTorch Frame provides two ways to handle text columns:

- Pre-encode text during **materialization** once to avoid computing the same embeddings

"text_embedded"

- Tokenize text during **materialization** once
- Finetune embeddings in **encoder**

"text_tokenized"



Data Materialization and Semantic Types

PyTorch Frame materializes columns of different *semantic types* as follows:

- `numerical` – Pass through of floating-point columns
- `categorical` – Map categories to indices
- `multicategorical` – Map multiple categories into indices of **varying length**
- `timestamp` – Map timestamps to integers of year, month, day, hour, minute, …
- `embedding` – Pass through of pre-computed embeddings
- `text_tokenized` – Tokenize text into a list of integers of **varying length**
- `text_embedded` – Pre-compute text vectors via external text models
- `image_embedded` – Pre-compute image vectors via external image models

Native Integration with Foundation Models

OpenAI cohere Hugging Face VOYAGE AI

kumo

Sparse Features in PyTorch Frame

PyTorch Frame stores variable-length features efficiently via `torch_frame.data.MultiNestedTensor`

Size of dense representation:

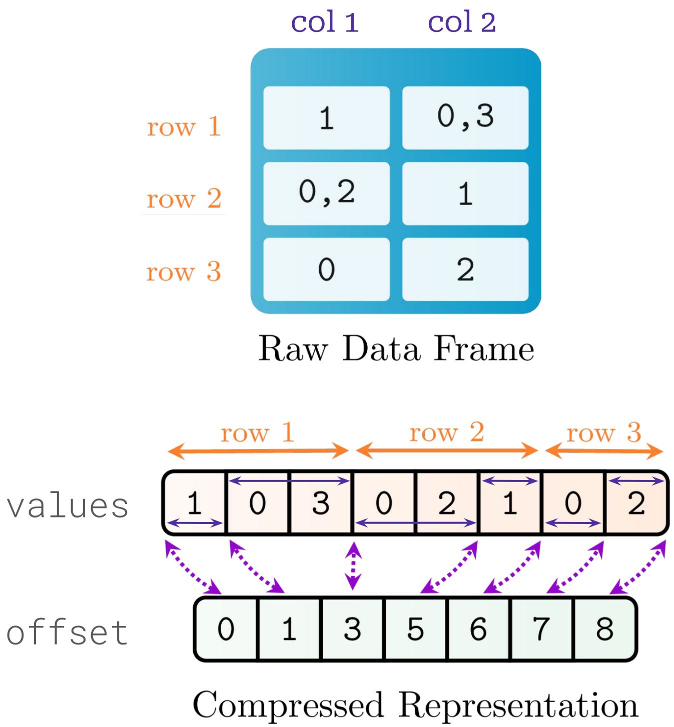
`N*max_entries*sizeof(dtype)`

Size of sparse representation:

`N*max_entries*sizeof(dtype)`**`*density`**
`+(N+1)*sizeof(int64)`

Example:

- dtype is int64, N is 1,000, max_entries is 1,000, density is 10%
- dense: 8,000,000 bytes
- sparse: 8,08,008 bytes (only 10% of dense version!)



https://pytorch-frame.readthedocs.io/en/latest/generated/torch_frame.data.MultiNestedTensor.html

kumo

Three-stage Model Architecture

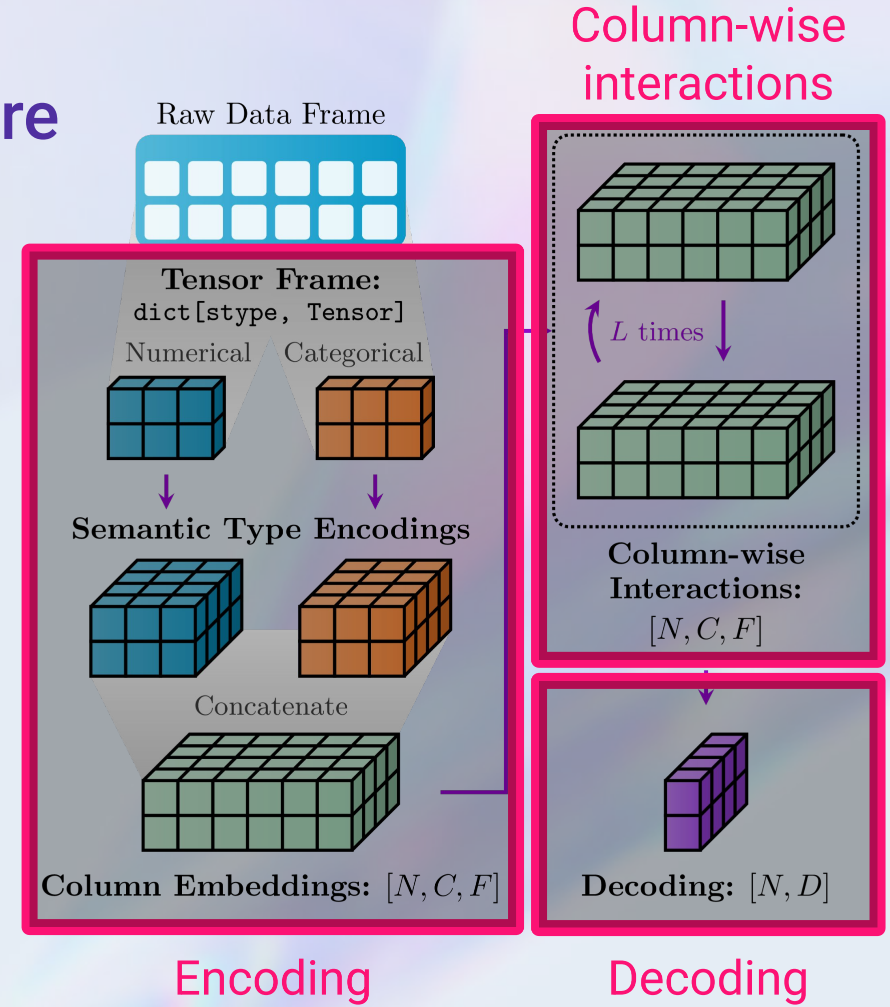
Encoding embeds each column independently

- missing data imputation
- normalization
- embedding lookup
- cyclic encoding
- positional encoding
- ...

Column-wise interaction performs message passing across columns.

Decoding summarizes column embeddings to obtain row embeddings.

- weighted sum of column embeddings
- MLP over the flattened column embeddings
- ...

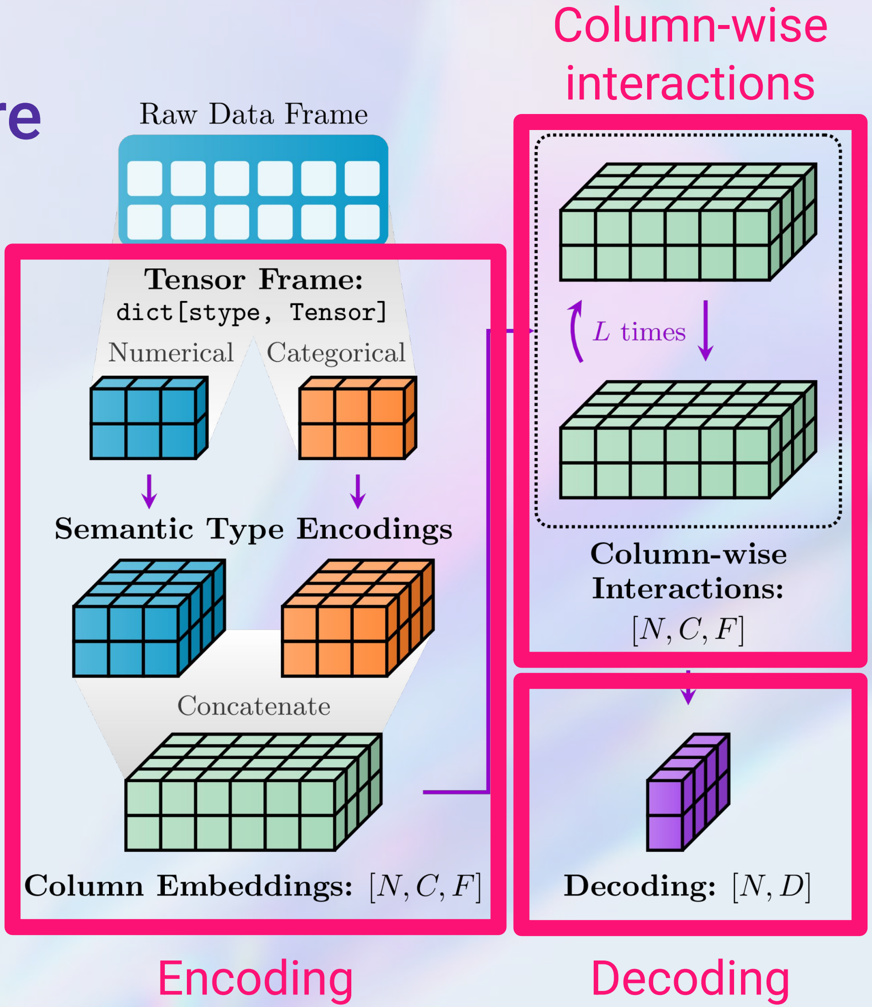


kumo

Three-stage Model Architecture

Many models fit within three-stage framework

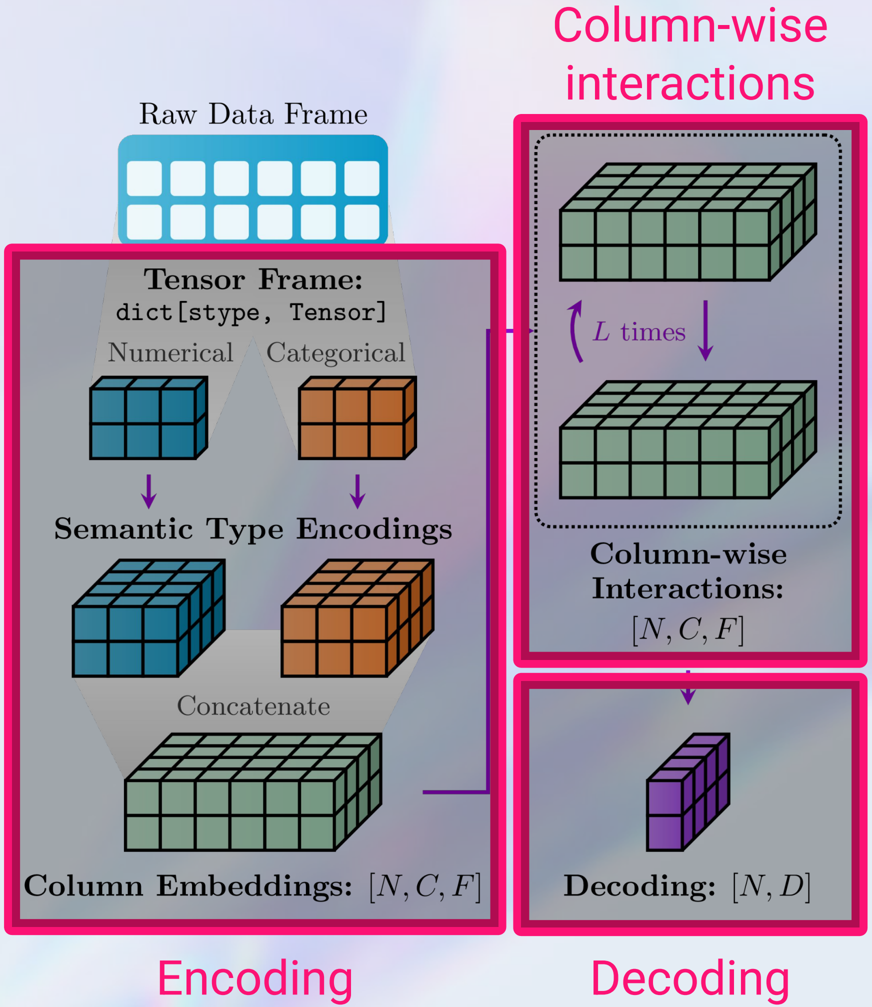
Model	Trompt	ExcelFormer	FT-Transformer
Encoder	Any	CatBoostEncoder	Any
Column-wise interaction	Trompt Cell	Transformer	CLS token + Transformer
Decoder	Trompt Downstream	MLP	MLP



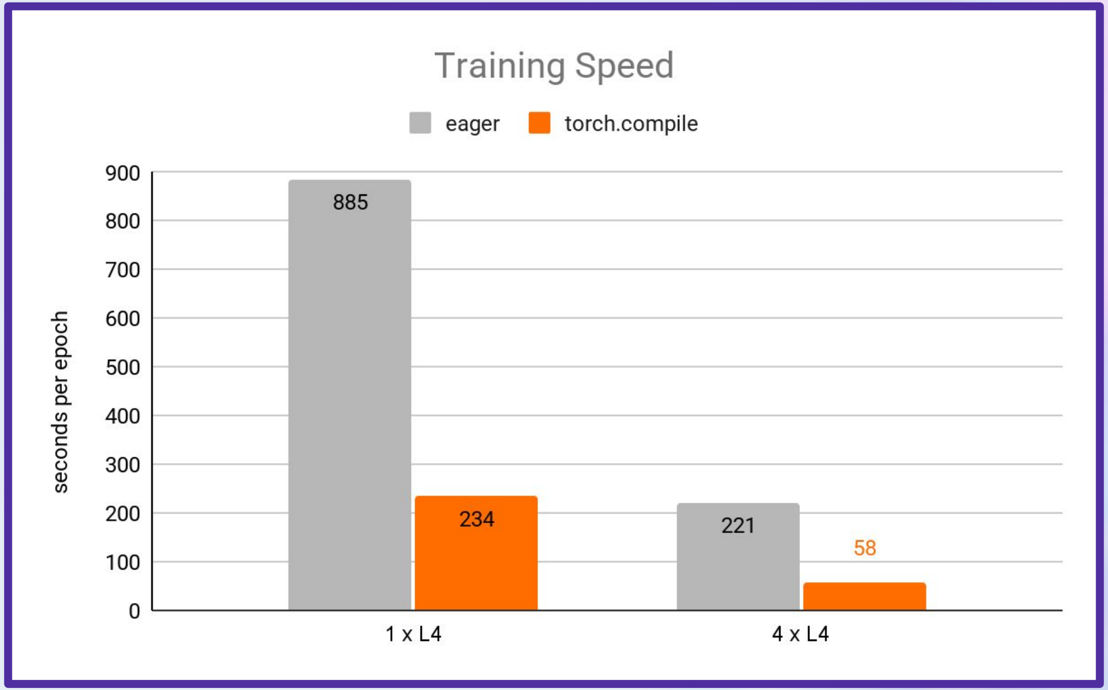
kumo

```python
class MyModel(torch.nn.Module):
  def __init__(self):
    self.encoder = StypeWiseFeatureEncoder(
      out_channels=channels,
      stype_encoder_dict={
        stype.categorical: EmbeddingEncoder(),
        stype.numerical: LinearEncoder(),
      },
    )
    self.convs = torch.nn.ModuleList([
      TabTransformerConv(
        channels=channels,
        num_heads=num_heads,
      ) for _ in range(num_layers)
    ])
    self.decoder = torch.nn.Linear(
      channels,
      out_channels,
    )

  def forward(self, tf: TensorFrame) -> Tensor:
    x, _ = self.encoder(tf)
    for conv in self.convs:
      x = conv(x)
    return self.decoder(x.mean(dim=1))
```



Compatible with `torch.compile` & DDP!



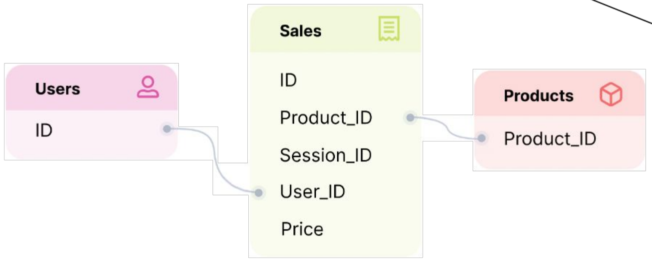

```
$ python examples/trompt.py --dataset Higgs --batch_size 512
$ python examples/trompt.py --dataset Higgs --batch_size 512 --compile
$ python examples/trompt_multi_gpu.py --dataset Higgs --batch_size 512
$ python examples/trompt_multi_gpu.py --dataset Higgs --batch_size 512 --compile
```

kumo

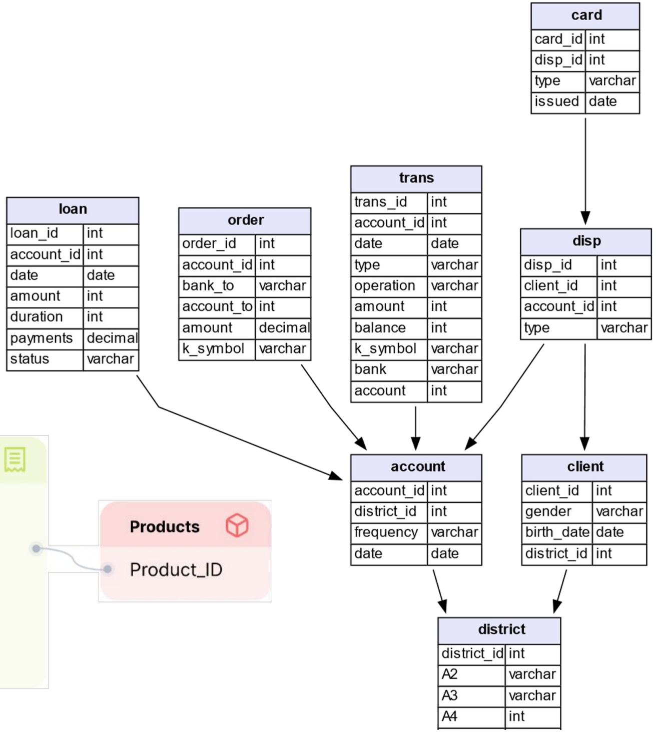
Single Table to Multiple Tables – Relational Deep Learning

kumo

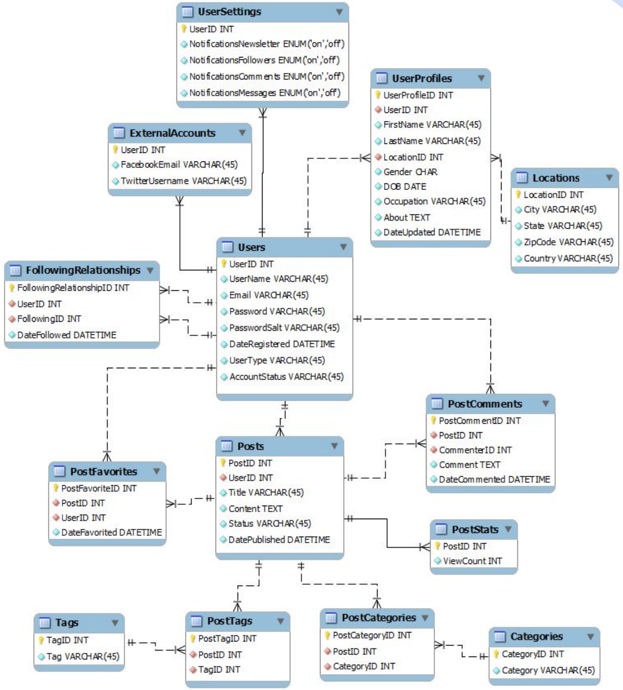
Relational Data in Real World



E-Commerce



Finance



Social Media

Relational Data Is a Graph

A **node** denotes a row in a table

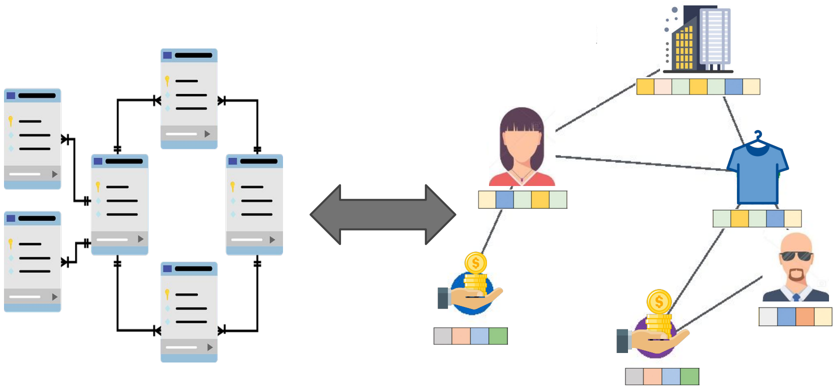
- A user in `USERS` table
- A product in `Sales` table

An **edge** denotes a pkey-fkey relationship between nodes

- `Users.ID` ↔️ `Sales.User_ID`
- `Products.Product_ID` ↔️ `Sales.Product_ID`

Nodes features are column features:

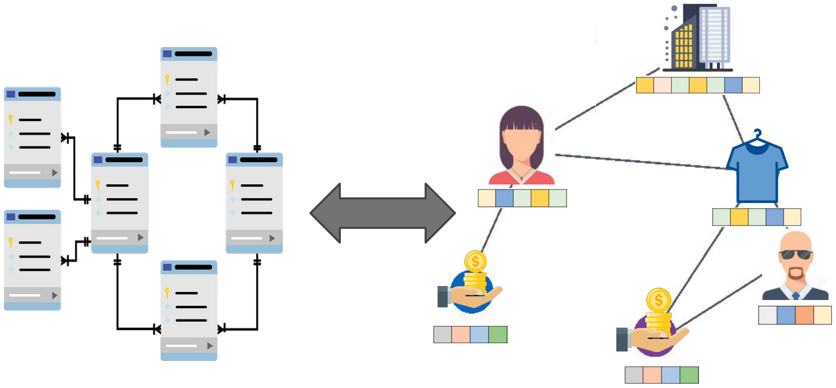
- User features: age, post code, signup date
- Product features: price, category tags, image, rating



Graph ML Tasks on Relational Data

Problem	Type
Age Prediction	Regression
Prediction of Movie Ratings	Regression
Lifetime Value	Temporal Regression
Active Purchasing Customer LTV	Temporal Regression
Active Purchasing Customer Churn	Temporal Binary Classification
Fraud Detection	Temporal Binary Classification
Next Item Category Prediction	Temporal Multiclass Classification
Probability of Liking an Item	Non-Temporal Classification
Item to Item Similarity	Static Link Prediction
Top 25 Most Likely Purchases	Temporal Link Prediction
Item Recommendation	Temporal Link Prediction
Top 25 “High Value” Purchases	Temporal Classification/Ranking

https://kumo.ai/docs/examples/predictive-query

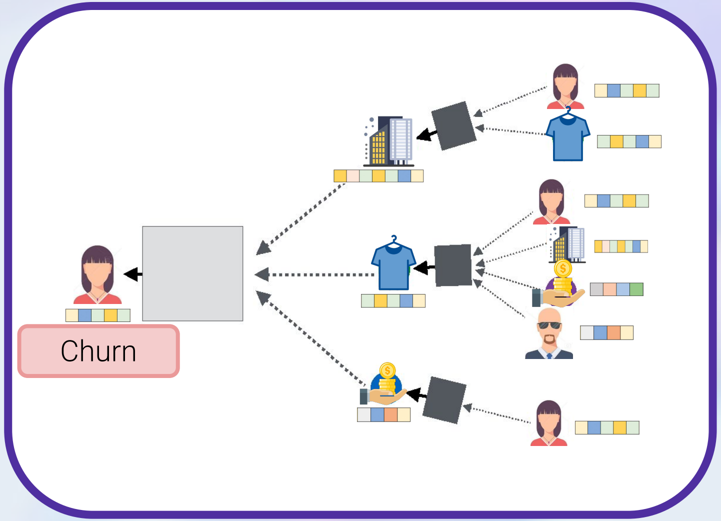


kumo

Graph Neural Networks on Relational Data

GNNs aggregate information from tables to make predictions

✅ No feature engineering
✅ No temporal information leakage
via temporal sampling
✅ More accurate model
via lossless graph representation

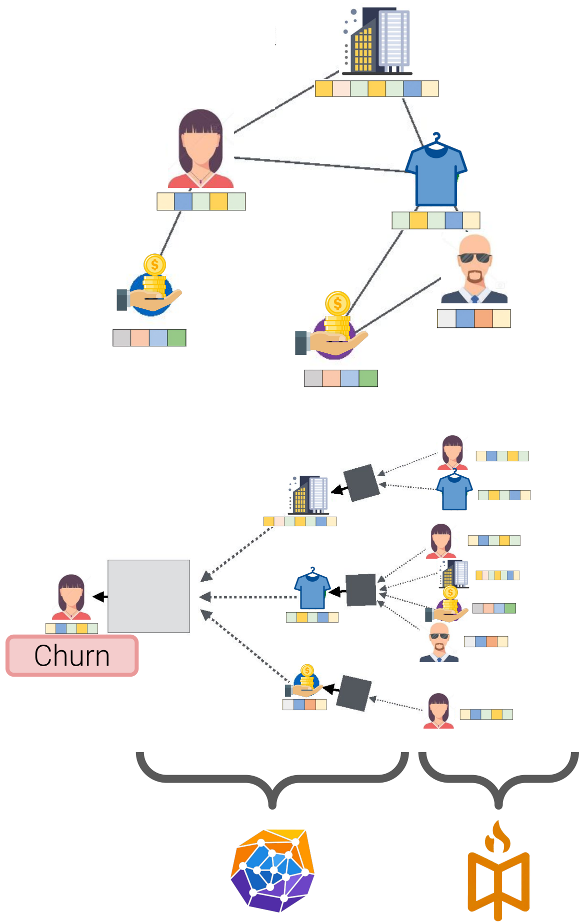


kumo

Tabular Models and GNNs

- Step 1: Sample a subgraph
- Step 2: Compute node embeddings for nodes in the subgraph
- Step 3: Perform message passing across table with GNNs

Optimize tabular model and GNNs end-to-end for your task



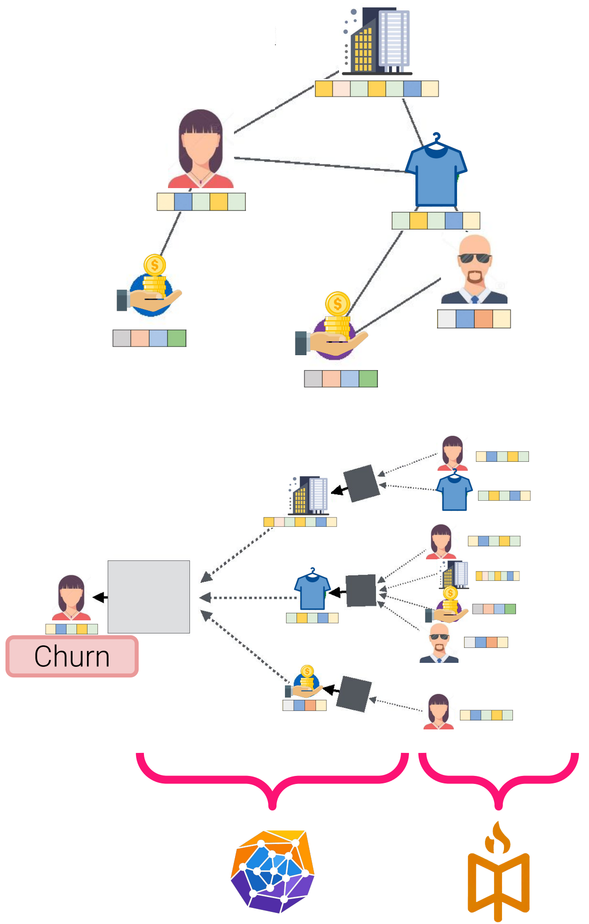
kumo

```python
class MyGNN(torch.nn.Module):
  def __init__(self):
    self.encoder = torch.nn.ModuleDict({
      "Users": MyTabularModel(...),
      "Products": Trompt(...),
    })
    self.gnn = to_hetero(GraphSAGE(...))

  def forward(
    self,
    tf_dict: dict[str, TensorFrame],
    edge_index_dict: dict[str, Tensor],
  ) -> Tensor:
    x_dict = {
      self.encoder[table_name][tf]
      for table_name, tf in tf_dict.items()
    }
    return self.gnn(x_dict, edge_index_dict)
```

RelBench: A Benchmark for Deep Learning on Relational Databases
https://arxiv.org/abs/2407.20060
https://github.com/snap-stanford/relbench

ContextGNN: Beyond Two-Tower Recommendation Systems
https://arxiv.org/abs/2411.19513
https://github.com/kumo-ai/ContextGNN



PyTorch Frame Community & Summary

✓ Support for multi-modal features
✓ Support for LLMs and foundation models
✓ Support for Relational Deep Learning

/pyg-team/pytorch-frame
/pyg-team/pytorch_geometric

```
pip install torch-geometric
pip install pytorch-frame
```

What's coming?

- Add more layers
- Add more models
- Integrate NestedTensor?
- Integrate fbgemm kernels?

Next steps:

- Run examples
- Use your own data
- Build your own models
- Check out RelBench
- Check out ContextGNN
- Contribute!
- Join PyG Slack!

https://join.slack.com/t/torchgeometricco/shared_invite/zt-2ppirgs10-_krkJVMgCeXiP92jDCRbig

💜 Community Contributors

anas-rz
berkekisin
Borda
crunai
DamianSzwichtenberg
drivanov
eliazonta
February24-Lee
HoustonJ2013
itsjayway
jyansir
kaidic
NeelKondapalli
puririshi98
rishabh-ranjan
SimonPop
toenshoff
xnuohz

🛠️ Core Contributors

akihironitta
rusty1s
vid-koci
weihua916
wsad1
XinweiHe
yiweny
zechengz

kumo akihiro@kumo.ai