

Contents

Problem Statement	1
Example Use Cases	4
Terminology	4
Tensor Notation	6
Background	6
TabBERT	6
UniTTab	7
FATA-Trans	8
Consolidating Sequence Modeling with Modular Fields	8
Modular Field Embedding System (MFES)	8
Discrete Field Embeddings	9
Continuous Field Embeddings	10
Temporal Field Embeddings	14
Geospatial Field Embeddings	14
Entity Field Embeddings	15
Media Fields	17
Textual Field Embeddings	17
Images, Video, and Audio Field Embeddings	17
Putting it all together: MFES and examples	18
Model Architecture Summary	18
Self-Supervised Learning Strategy	19
Masked Event and Masked Field Modeling Tasks	19
Semi-Ordinal Classification of Quantiles	20
Fine-tuning Sequence Models	20
Versatility of Supervised Tasks	21
Observation Sampling Strategies	21
Additional Considerations	21
Data Preprocessing for Efficient Streaming Operations	21
Sequence Entropy	22
Arbitrary Dimensionality	22
The Clipping Problem	22
The Flushing Problem	23
Including Tabular Data and Multiple Sequences of Varying Event Rates	23
Sub-Quadratic Alternatives to Self-Attention	23
Conclusion	23
Bibliography	24

Problem Statement

Over the last five to ten years, we have observed tremendous leaps in machine learning for universal modalities such as vision, language, audio, and video. With approximately twenty lines of code, anyone can download and fine-tune a foundation model to solve many common problems because of the extraordinarily accessible high-level abstractions built for these modalities. However, these high-level abstractions are limited exclusively to the problems within and in-between these select few universal modalities (text-to-speech, image-to-video, text-to-audio, image-to-text, text summarization, sentence similarity, etc). These high level, generalized abstractions are possible because the modalities are universal.

Tabular machine learning, in practice, has not really changed all that much over the last decade. If you ask anyone how they would approach a tabular machine learning problem they will likely recommend a gradient boosted machine (GBM). GBMs provides excellent results for most tabular machine learning problems. Beyond

their spectacular performance, they are intuitive, interpretable, and easy to train. They are able to scale to larger-than-memory datasets with distributed compute and accelerated hardware. GBMs are the long-reigning, undisputed champion for many tabular machine learning problems.

GBMs may be the best solution for actual tabular machine learning. However, I do not think that they are not the correct solution for many of the problems in which they are currently utilized. In other words, I strongly believe that the vast majority of machine learning problems that are currently being modeled with GBMs are not actually tabular.

That is a really bold statement that I would like to contextualize with an example:

Suppose you are to train a model to of customers. You are currently passing in the following tabular features into a model in order to predict whether a given customer is lying about their identity: `income`, `phone_local`, `phone_confirmed`, `email_confirmed` and `verified_income`.

Consider these two sample training observations:

customer	income	phone_confirmed	phone_local	email_confirmed	verified_income
John	80,923	False	True	False	False
Jane	90,413	True	True	True	True

Table 1: A simple example of tabular data

At first glance, the use of tabular machine learning is completely reasonable. A GBM will make quick work of such tabular data, likely providing the best results if you have enough labeled training observations and you rigorously optimize your hyperparameters. However, real production data does not actually look like this. This tabular feature representation is one of convenience. It is an abstract “view” that has been created in order to train a tabular model. The data, in its original form, might instead look like so:

customer	timestamp	event
John	01-01-2024 03:04:50	CreateAccountEvent()
John	01-01-2024 03:05:43	AddPhoneEvent(phone='1236540987', is_local=True)
John	01-01-2024 03:07:12	AddEmailEvent(email='john@gmail.com')
Jane	01-01-2024 15:42:51	CreateAccountEvent()
John	01-02-2024 07:43:49	AddIncomeEvent(income=80923)
Jane	01-02-2024 07:44:25	AddEmailEvent(email='jane@hotmail.co')
Jane	01-02-2024 07:45:41	AddPhoneEvent(phone='4217838914', is_local=True)
Jane	01-02-2024 07:48:53	ConfirmPhoneEvent(phone='4217838914', is_local=True)
Jane	01-02-2024 07:51:55	AddIncomeEvent(income=90413)
Jane	01-02-2024 08:05:03	ConfirmIncomeEvent(income=90413)
Jane	01-03-2024 11:18:29	FailedEmailAttemptEvent(email='jane@hotmail.co')
Jane	01-03-2024 16:33:49	AddEmailEvent(email='jane@hotmail.com')
Jane	01-03-2024 16:35:12	ConfirmEmailEvent(email='jane@hotmail.com')

Table 2: How sequential / transactional data might look like in a production table

This is the complete history of every event observed of each customer. This construct may fit in a table, but with respect to machine learning is not necessarily tabular. It is actually something more universal: a “ledger” of posted events. Each event streams in asynchronously, perhaps over a data stream such as Kafka, and is immutable such that once a customer does something, it stays on their ledger. Each event can contain any number of complex fields, and each field can have any data type (categorical, textual, numeric, boolean, timestamp, geospatial coordinates, and even file paths to images or audio clips). Additionally, each event may have an associated timestamp for when it was received. This data architecture pattern is called “Event Sourcing”. It is

extremely powerful for applying business logic and maintaining a record of every customer's history for the sake of reproducibility. It defines the complete history of everything ever done by every customer.

Here are the distinct fields among all of the events in the example above:

- `event_type`: what the customer did (`CreateAccountEvent`, `AddPhoneEvent`, `AddEmailEvent`, etc)
- `phone`: The customers reported phone number
- `is_local`: Whether the phone number is local to the customer's reported address
- `email`: The customers reported email address
- `income`: The customers reported income
- `timestamp`: The time at which the event occurred

The ledger is the source of truth for everything that has ever happened, and will always contain a superset of the information available within any tabular view made from it.

You may be thinking that the ledger is going to be more complicated to model than the tabular view we had looked at before. It certainly is more complicated, yes, however it contains significantly more information. For example, we can see that Jane originally misspelled her email address as `jame@hotmail.co`, so she couldn't verify it until she corrected it to `jane@hotmail.com`. We can see the duration of time that transpired between each event observed of a customer. All of this information *could be* vital to modeling an identity verification problem. Tabular machine learning is incapable of modeling all of these potential signals without significant manual preprocessing and feature engineering, all of which is subject to human bias and human error.

Of course, you *can* create more tabular features that are able to collectively approach the signal available within the ledger. For example, you could create more tabular features to, say, count the number of times each customer has changed their email over varying time windows (over the last day, week, and month). You would also need to count the number of times the email was verified, however, also for varying time spans (over the last day, week, and month) as well as the number of times the email verification failed. This approach doesn't scale well in practice, however. With more complex examples, you will need to create hundreds, if not thousands of tabular features to account for many different possible intersecting events, over many varying time windows, with multiple possible aggregation functions.

When you model sequential data with tabular models, these aggregative window functions are required to capture the complex signal. They are extremely computationally expensive to calculate, and it takes a great deal of time and domain expertise to know which kinds of features you should engineer, often leading to a painful, iterative process. Even beyond the absurd computational and domain expertise requirements, however, is the limitations for real-time inference. Calculating hundreds of arbitrarily unique window functions in real time requires entire teams of dedicated on-call engineers. Because of all of these aforementioned costs of modeling sequential data with tabular models, many organizations are simply unable to address many problems that would otherwise provide great value-add.

All of these factors result in a painful, manual iterative process of engineering tabular features via SQL or Spark, training models with them, and then pruning the tabular features for the sake of model complexity and real-time inference requirements. Additionally, GBMs are especially prone to overfitting with many tabular features, adding additional constraints to the optimization process.

By modeling the data exactly how it already exists in its production source table, as opposed to using a manually constructed view of it, we may be able to skip all of the feature engineering! Feature engineering is the bane of tabular modeling. It is time-consuming, tedious, prone to error, and extremely hard to scale for real-time computation. It requires extensive testing, maintenance, and monitoring. However, by using the *raw* data to model our problem instead of an abstract view of it allows us to instantly unburden the modeling process from all of these significant costs.

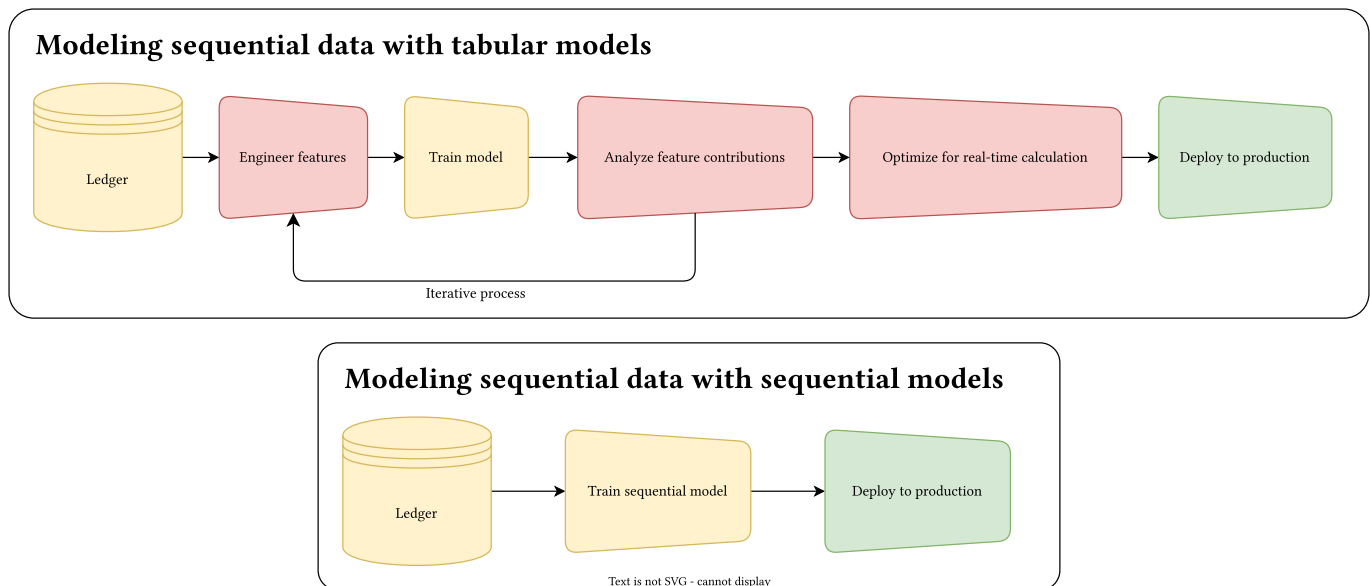


Figure 1: Cutting out tabular feature engineering dramatically increases speed-to-market

We should change our model to fit our data instead of changing our data to fit our model. This has many parallels to the history of machine learning for universal modalities. The modality of computer vision comes to mind. Before the birth of CNNs, data scientists were manually defining convolution kernels. This is, in essence, how we are currently modeling business problems: with manually hardcoded tabular features.

In summary, instead of manually aggregating the events through miscellaneous window functions in order to represent their current state, with the use of deep learning, we may create a representation of each customer at any point in time given all of the events that they have posted up until that point in time. We assume, from the perspective of our model, an individual may be defined by the sum of their actions. This customer representation may then be used to solve any arbitrary machine learning problem with a single generalized modeling approach that does not require any manual feature engineering. This would allow us to focus exclusively on the construction of an automated modeling pipeline. The remainder of this paper will discuss how this may be done.

Example Use Cases

Before diving into how this may be done, I want to reflect on the types of problems that this approach is able to solve: The above example may seem a little cherry-picked. I would like to provide a few more problem statements to highlight just how prevalent this data structure is:

1. Given every previous ride observed by an Uber customer, what is the probability that they will be late for the next ride?
2. Given every activity log observed by a customer's Apple Watch, what is the probability that they will meet their workout goal this week?
3. Given every single transaction posted by a credit card holder of Chase, what is their survival curve (either due to attrition or default) over the next five years?
4. Given previous energy usage events observed by a customer of Duke Energy, what is their expected energy usage for the next week?
5. Given the previous purchases observed by a vendor on Ebay, what is the probability that they will be reported as a "scammer" in the next week?
6. Given the previous transactions posted by a user of Venmo, what is the probability that the current transaction is an attempt to defraud another user?
7. Given the previous tweets posted by a user of Twitter, what is the probability that this user is a bot?

There are hundreds of more problem statements associated with dozens of varying industries. This framework that I am describing applies to all of them.

Terminology

It will be helpful to define some terminology before going forward. I will be discussing the architecture of an extensible *sequence model*. The sequence model is able to train on *observations*, which each are a slice of a

sequence. A sequence is made of multiple **events**, in which each event may have any number of **fields**, which may have multiple possible data types. I may refer to this data as being “sequential”, “transactional”. It really is, however, a sequence of multivariate events with nullable, heterogeneous fields.

The population of sequences exists in some data table, in which each event is a record. I refer to this source of raw data as a **ledger**. I want to go through the ledger, find all of the unique sequence IDs (IE, customer IDs), and find all of the events that exists within each sequence. Each event includes multiple fields. Every field exists as a column in this table. The values of each field are nullable; any field value can be empty, which the model must be able to distinctly represent and learn.

I want to stream over each sequence, looking at every event available within each sequence. I will then sample random events from the sequences. Each event has an equal probability of being sampled. After sampling each event, I then take that event, and then I collect the L preceding events to calculate the **context** for the training sample. If there are not L preceding events before the sampled event, then the context is equal to however many events are available. I then pad and collate the context, collecting every field value within the context, to create an **observation**.

For any supervised modeling task, I am able to take an associated **target** value from the sampled event. In other words, using this methodology, I am able to support event-level targets for both classification or regression.

I then create many more observations from other sampled events, some of which may be from the same sequence, in order to create a training batch, which I then pass into the sequence encoder. Observations are used to pre-train and fine-tune the sequence model.

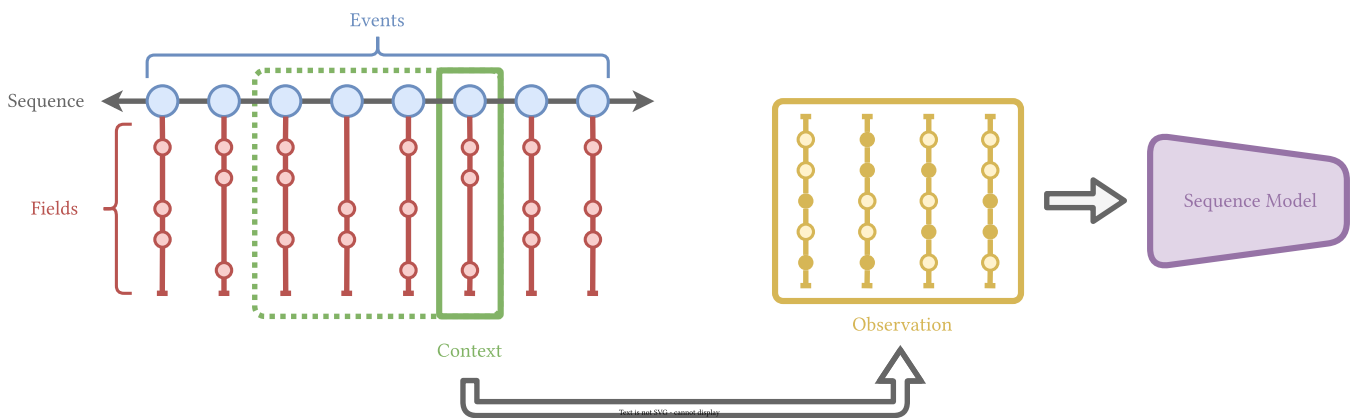


Figure 2: An observation being sampled and collated from a sequence

This section describes how this process may be optimized for streaming arbitrarily large data from disk.

The model architecture, which I will discuss in detail in the next section, utilizes transformer-encoder blocks in a manner similar to BERT. The sequential model is creating a representation of the events within the context to answer questions about the sequence at a specific point in time. This is in contrast to models like GPT, which utilize transformer-decoder blocks to predict subsequent tokens. The data structure that I have described above may seem similar to how a corpus is sampled to train a models like BERT. That is not entirely inaccurate.

However, there is two significant differences:

1. You may think a list of tokens as being events with a single discrete field, whereas multivariate sequential data may include more than one field. In other words, my model architecture requires one more dimension than BERT.
2. BERT samples observations at a sequence-level, but we need to sample observations from sequences at an event-level. For example, a customer with 10,000 events should result in approximately 10 times more observations than a customer with only 1,000 events. We are creating observations from the context, which are each defined by a randomly sampled event. This enables the model to support fine-tuning with targets uniquely defined for each event.

Peltarion published an excellent walkthrough that better describes how BERT works, which is a prerequisite to understanding the model architecture described below. [1]

Tensor Notation

Given that we are talking about a rather complicated model structure, here is a reference to the common dimensions used for the tensors:

Tensor Notation	Hyperparameter Value Name	Definition
N	<code>batch_size</code>	Batch Size
L	<code>n_context</code>	Maximum Context Size of each Observation
F	<code>n_fields</code>	Number of Fields
C	<code>d_field</code>	Hidden Dimension for each Field
FC	<code>d_field * n_fields</code>	Hidden Dimension for each Event

Table 3: Tensor notation and definitions for reference

Background

TabBERT

In early 2021, developers at IBM published *Tabular Transformers for Modeling Multivariate Time Series* in which they described a novel model architecture (TabBERT) that could outperform GBMs trained on such handcrafted tabular features for a set of supervised problems with sequential data. [2]

In essence, they are framing the sequence as a two dimensional array. One dimension of the array represents each event in the sequence, and the second dimension represents each field in each event. In their implementation, every single field is limited to being exclusively discrete, such that each value is an integer that needs to be embedded via a standard embeddings table with respect to each field.

BERT was embedding a 1D discrete input of tokens (a sentence of words) into 2D space (an array of embeddings):

```
def embed_1d(tokens: Int[torch.Tensor, "N L"]) -> Float[torch.Tensor, "N L C"]:  
    ...
```

Whereas TabBERT embeds a 2D discrete input (a sliced sequence of events) into 3D space:

```
def embed_2d(tokens: Int[torch.Tensor, "N L F"]) -> Float[torch.Tensor, "N L F C"]:  
    ...
```

TabBERT then utilizes a hierarchical transformer-encoder architecture with two encoders. The first encoder (in my terminology, the **field encoder**) lets every field embedding attend to every field embedding within an event. Those attended field representations are then concatenated to represent the event as a whole (`["N L F*C"]`). The second encoder (in my terminology, the **event encoder**) lets every event representation to attend to one another. The event encoder is constructed similarly to BERT, such that there is an appended class token that may be used to represent the sequence as a whole, but one can also use the attended event representations for a MLM-like self-supervised learning task for model pre-training. They defined a custom pre-training task in which a portion of the events and fields are masked, and the event representations outputted from the event encoder needs to be able to reconstruct the masked events and fields. For fine-tuning, the appended class token is passing through a MLP to solve the supervised task at hand.

For those unfamiliar with hierarchical transformer architecture, a great analogy comes from the domain of computer vision. If one is building a representation of an image, ideally each pixel should interact with every other pixel. However, if one is building a representation of a video, it would be virtually impossible to allow every pixel of every frame to interact with every pixel of every frame. Instead, it is common to first build a representation of each frame by letting each of its pixels interact with one another, and then build a representation of the video by allowing each representation of each frame interact with every other representation of each frame. Similarly, TabBERT breaks down a very large and complex problem into two smaller problems: representing each unique event and then using those event representations to create a representation of the sequence.

The hierarchical transformer architecture dramatically reduces the memory complexity of the model. Transformers are notorious for their quadratic memory complexity (both during training and inference) with respect to the context size. By utilizing two transformers to encode the fields and then events separately, we do not require the extensive memory requirements that troubles language models. In effect, we have reduced the memory complexity from a worst case scenerio of $O((LF)^2)$ into a slightly less scary $O(L^2F + LF^2)$. Within the typical ranges of L (128 to 512) and F (6 to 15), this reduces memory requirements by approximately a factor of 10 to 20.

TabBERT, while elegant, did come with some major drawbacks. The largest of which is that their implementation required that every field be coerced into a categorical data type. For example, dollar amounts could not be fully represented, but you can bin them into categorical levels:

Continuous Input	Discrete Output
\$4.95	"\$x<\$5"
\$8.50	"\$5<x<\$10"
\$19.95	"\$10<x<\$20"
\$103.32	"\$100<x<\$200"

Table 4: How some example continuous dollar amounts might be discretized to categorical inputs

Coercing every field to become of a categorical data type comes with model performance costs. Much of GBM's success comes from their ability to define arbitrary cutoff points for continuous features. For example, in some fraud patterns, fraudsters might prefer gift cards, which commonly come in denominations of exactly \$25, \$50, or \$100. In such a fraud pattern, transactions with dollar amounts equal to exactly those values may be far more likely to associated with known fraud patterns than transactions with dollar amounts with, say, values around \$25.08, \$49.12, or \$99.53. A GBM is free to chase the residuals around these exact cutoff in the continuous distributions. My point being, for some tasks arbitrary precision of continuous fields could be extremely important such that discretizing the continuous fields destroys valuable information.

Additionally, the pre-training task suffers from the masked categorical targets if you are working with continuous values that were binned. Neither the model nor the loss function are aware of the ordinal nature of the binned categories. For example, if a masked dollar amount is actually \$9.95 but the model predicts that the value is in the bucket "\$10<x<\$20", the loss function harshly penalizes the model even though the model was almost correct. By "harshly", I mean that the loss function believes that the models prediction of "\$10<x<\$20" is as "wrong" as a prediction of "\$100<x<\$200".

UniTTab

In 2023, a small Italian consulting firm Prometeia published *One Transformer for All Time Series: Representing and Training with Time-Dependent Heterogeneous Tabular Data* which introduced the UniTTab architecture. UniTTab made use of some very interesting ideas to represent continuous field values. Their novelties included the use of Fourier feature encodings to "embed" floating point field values without having the bin the values. They referenced the NeRF (Neural Radiance Fields) paper which highlighted how Fourier feature encodings provide neural networks with a better representation of floating point values than passing in raw scalar values. [3]–[5]

Fourier feature encodings effectively embed the true value of continuous inputs. They are quite similar to the sinusoidal positional embeddings used by early original transformer-based models. The authors of *Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains* discussed this approach at a [NeurIPS 2020 spotlight](#) which they describe how this technique works. [5]

UniTTab also adapted TabBERT's custom self-supervised learning task to work with continuous fields. Instead of trying to reconstruct the raw scalar values of each field via regression, UniTTab simplified to problem to become one of ordinal classification, in which the model attempts to determine the discretized bin of masked continuous field values. They modified the loss function to "smoothen" the loss if the predicted binned logits are closer to the actual binned target value, a technique that they described as "neighborhood loss smoothing".

UniTTab's improvements over TabBERT resulted in significantly better performance. UniTTab did not share its source code, however. Additionally, they were quite vague in the implementation. They require that each event be associated with a registered "row type", and that each "row type" has a required, non-nullable schema, which, at least in how I consider their approach, requires an extraordinarily complex implementation in order to be able to trained with accelerated hardware. I designed a modified approximation of UniTTab, and I found that it outperformed my implementation of TabBERT.

FATA-Trans

In November of 2023, Visa published *FATA-Trans: Field And Time-Aware Transformer for Sequential Tabular Data*. FATA-Trans was built on top of TabBERT, and did not provide any support for continuous fields. They included two major contributions: the ability to encode timestamps, and the ability to use tabular features alongside sequential data. Theoretically, one could already pass in tabular features by making the values constant among every event in the sequence, but as Visa stated, this is pretty computationally inefficient. They concluded that their architecture had marginal improvements over TabBERT. [6]

Consolidating Sequence Modeling with Modular Fields

TabBERT, UniTTab, and FATA-Trans have all contributed a great deal to sequential modeling. TabBERT introduced the idea of hierarchical attention to capture the two-dimensional nature of multivariate sequences. UniTTab improved upon TabBERT to utilize continuous fields. FATA-Trans introduced encoding timestamps and tabular features. [3], [2], [6]

In this next section, I will be discussing a novel approach that I have developed to consolidate the embedding strategies of the previous papers with a focus on automation and extensibility, providing the following benefits:

1. Combines the abilities of the three aforementioned papers (supporting discrete, continuous, and temporal field types), while also removing their limitations and rigidity by support arbitrarily nullable field values.
2. Introduce a unique representation of "entities" and geospatial coordinates. I am defining an "entity" as a categorical input with infinitely many, or effectively infinitely many levels. For example, a device ID, a login session, a phone number, or a merchant name.
3. Introduce the idea of including "embedded media", such as text, images, video, and audio within the context leveraging foundation models for each universal modality as required.

In order to support the wide variety of these fields, I have designed a custom implementation architecture that is able to adapt its submodules to support any combination of field requirements. The implementation manages a set of unique field embedders for each supported field type with extensibility in mind, such that embedders for new field types may be independently managed as a "plugin" system. I have called the system that manages the instances of the field embedders the "Modular Field Embedding System" (MFES).

Modular Field Embedding System (MFES)

As stated above, UniTTab supports discrete and continuous fields through a system the authors call "row types", in which each event must have a specific type with a rigid, non-nullable structure. I have found this system to be inflexible, and not necessarily intuitive to implement either. I instead offer an alternative approach that does not require any concept of "row types", but instead relies upon "field types". Instead of trying to define the unique "type" of each event, I view the data structure from the perspective of its unique fields.

A simple analogy comes from the concept of "dataframes". You might view a dataframe as a list of tuples, in each each tuple is a row. Alternatively, you could instead view a dataframe as a dictionary of lists, in which each list is a column. In this analogy, UniTTab elected to focus on the row-major structure of lists of tuples in order to avoid having to represent null values. I am taking the second approach, where each column is represented by a list within a dictionary.

Instead of creating "row types" in the style of UniTTab, I am creating "field types". Each field type is completely modular in that its contents are defined within a `tensorclass` (basically a `dataclass` of tensors). Each field type requires several components:

1. A type-specific field embedding module (to create the field embeddings for a specific field type)
2. A type-specific field masking function (to mask the field values during pre-training)

3. A (optional) type-specific field decoder (to turn an contextualized event representation back into field-level predictions for the self-supervised learning task). If one isn't specified, the field types will not be used in the self-supervised learning task.
4. A (optional) custom attention mask generation function to facilitate sparse attention for the field encoder and event encoder.

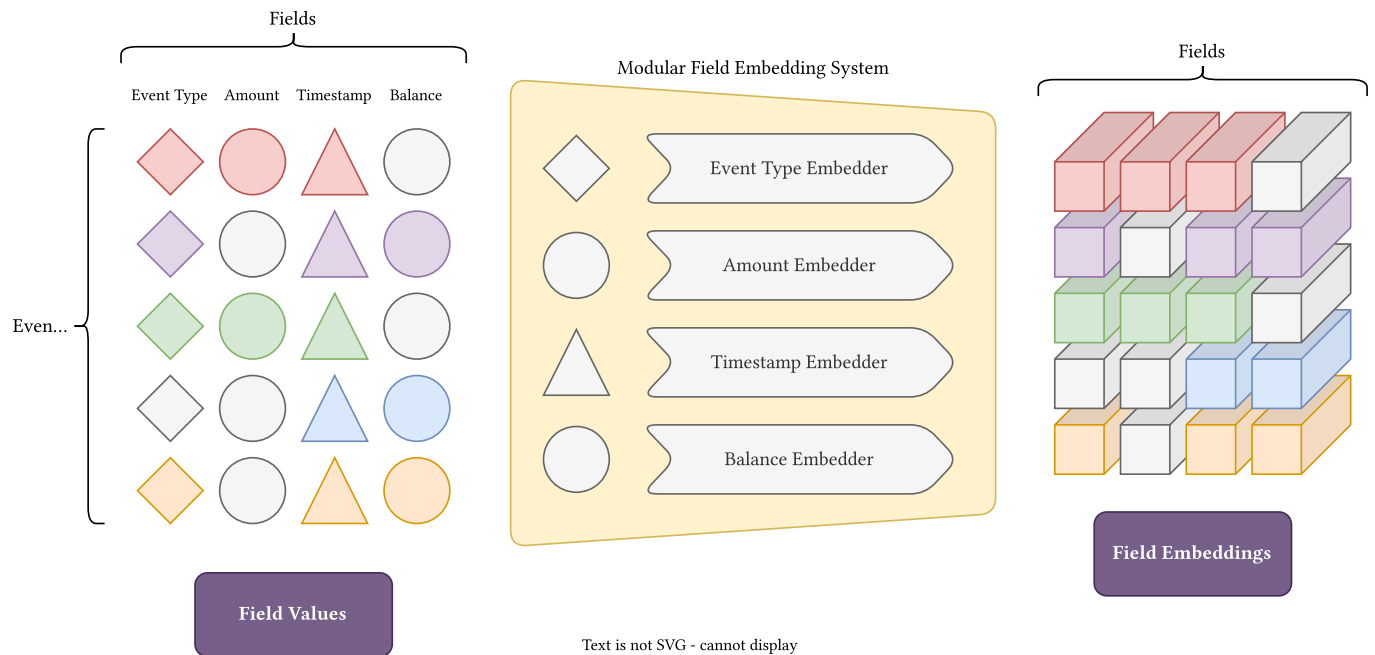


Figure 3: The MFES is able to embed fields of arbitrary types

It is extremely important to note that this approach is only valid with the assumption that each field's non-valued states (being masked, padded, or otherwise null) are able to be uniquely represented without overlapping with the valued states. This is the cost of viewing the data from a column-major perspective.

In order to embed each field, we need to ensure that each field is able to be fully represented regardless of its state. There are four unique possibilities for any given field:

1. There is a field value present (the field is "valued").
2. There is not a field value present because the field's event was padded (the field is "padded")
3. There is not a field value present because there wasn't a field value available in the data source (the field is "null")
4. There may or may not be a field value present, but the model is not allowed to see it (the field is "masked")

Discrete Field Embeddings

For an extremely simple example, let's start with a simple model architecture that needs to support four discrete fields per event: `is_foreign`, `is_online`, `merchant_category`, and `transaction_type`.

I use torch's `TensorDict` and `tensorclass` implementations to bundle the heterogeneous fields. For those unfamiliar with these constructs, they are almost exactly how they sound. A `TensorDict` is a torch dictionary that can contain instances of `torch.Tensor`. A `tensorclass` may be used similarly to `dataclass`. I use it to create a unique, type-checked container for each field type.

Below is an example of the input to a sequence model with four discrete fields. The input for each discrete field is contained within a `tensorclass` called `DiscreteField`.

```
@tensorclass
class DiscreteField:
    lookup: Int[torch.Tensor, "N L"]

    def mask(self):
        ...
```

```
# example input with a batch size of 2, sequence length of 8, and 4 discrete fields
```

```
TensorDict({
    is_foreign: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    is_online: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    merchant_category: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

This may seem unnecessarily complex at first, but it is quite necessary to support multiple the contents of varying field types. Additionally, methods may be attached to each tensorclass to support field-type specific operations, such as masking, preprocessing, and how to define the targets for the self-supervised learning task.

The modular field embedding system for this field schema looks like so:

```
ModularFieldEmbeddingSystem(
    (embedders): ModuleDict(
        (transaction_type): DiscreteFieldEmbedder(
            # six unique transaction types plus padded, null or masked
            (embeddings): Embedding(9, 12)
        ),
        (merchant_category): DiscreteFieldEmbedder(
            # twenty unique transaction types plus padded, null or masked
            (embeddings): Embedding(23, 12)
        ),
        (is_foreign): DiscreteFieldEmbedder(
            # yes, no, padded, null, or masked
            (embeddings): Embedding(5, 12)
        ),
        (is_online): DiscreteFieldEmbedder(
            # yes, no, padded, null, or masked
            (embeddings): Embedding(5, 12)
        ),
    )
)
```

Upon passing in the above TensorDict into the ModularFieldEmbeddingSystem, the MFES will iterate over each of the four discrete fields, embedding each of them with their respective DiscreteFieldEmbedder to create four tensors of dimensionality ["N L C"]. After embedding each discrete field, the ModularFieldEmbeddingSystem will then concatenate the embeddings into a single tensor of dimensionality ["N L F C"], which is then passed into the field encoder.

Representing the non-valued states (padded, null, or masked) is trivial in this case. They are additional tokens available within the DiscreteFieldEmbedder's embedding table. This is almost exactly like language models with three exceptions:

1. In this case each unique field embedding module has its own embeddings for these non-valued states. The special token embeddings are not shared.
2. Positional field embeddings are technically not required to embed each field. Because each field is guaranteed to exist one, and only once, the learned embeddings will learn to collectively learn to become their own positional embeddings for the field encoder. For the event encoder, positional embeddings are still required to help the event encoder understand the ordering of the events.
3. There is no [CLS] token present in the field embedding module. The [CLS] token will be appended to the event representations right before the event encoder module. The field encoder will not add any value to a [CLS] token.

Continuous Field Embeddings

UniTTab's approach avoids the complications of having to represent null, padded, or masked continuous field values by enforcing a rigid structure for each "row type". I found this to be extremely challenging to implement. Instead, my approach fully represents continuous fields in their true form as, effectively, enum data types:

```
enum ContinuousFieldValue {
    Value(float),
```

```

    Null,
    Padded,
    Masked,
}

```

In other words, a continuous field value can either contain a real number, or it could be non-valued, in which case it must be padded, masked, or otherwise null. I represent this enum-like structure with continuous fields with two tensors: values and lookup, both of dimensionality $[N \ L]$.

The values tensor contains the cumulative distribution function (CDF) of the original floating point values multiplied by a scalar value slightly less than 1 ($\bar{x} = \text{CDF}(x) \cdot 0.9$). This guarantees that $\bar{x} \in [0, 1)$ for purposes of optimization. Wherever a field is non-valued (either null, padded, or masked), I impute values to be equal to 0.0. By using the CDF of continuous field values, as opposed to the raw scalar values, I do not require any data standardization / normalization operations in order to represent values of arbitrary distributions. The CDF of the values also provide additional benefits that I will describe later in this section.

The lookup tensor contains one of four integers to explain the state of each field (valued, padded, masked, or otherwise null). In practice, I define the definitions of these four unique integers within a python IntEnum:

```

class SpecialTokens(enum.IntEnum):
    VAL = 0 # the field is "valued"
    UNK = 1 # the field is otherwise "null"
    PAD = 2 # the field is "padded"
    MASK = 3 # the field is "masked" for pre-training

```

This token definition is used as the source of truth for every field. It is important to note that the VAL token is mapped to 0, which allows for a beautiful mathematical optimization trick described later in this section.

Now that we have these two tensors, values and lookup, we may combine them in a new tensorclass to represent any continuous field:

```

@tensorclass
class ContinuousField:
    lookup: Int[torch.Tensor, "N L"]
    values: Float[torch.Tensor, "N L"]

```

In order to solidify this concept, let's walkthrough some examples to show how every possible state of our ContinuousFieldValue enum can be mapped to an instance of ContinuousField:

Input Field Value	Output Tensorclass Representation (assuming $N = 1$ and $L = 1$)
PAD	ContinuousField(lookup=[[2]], values=[[0.00]])
VAL(0.93)	ContinuousField(lookup=[[0]], values=[[0.93]])
VAL(0.65)	ContinuousField(lookup=[[0]], values=[[0.65]])
UNK	ContinuousField(lookup=[[1]], values=[[0.00]])
VAL(0.31)	ContinuousField(lookup=[[0]], values=[[0.31]])
UNK	ContinuousField(lookup=[[1]], values=[[0.00]])
VAL(0.00)	ContinuousField(lookup=[[0]], values=[[0.00]])
MASK	ContinuousField(lookup=[[3]], values=[[0.00]])

Table 5: How enum continuous inputs can be uniquely mapped to a pair of tensors

This mapping provides a guarantee that every possible input has a unique tensor representation. Similar to UniTTab and NeRF, I can embed the values tensor with Fourier feature encodings. This process looks like the following:

$$B = 2^{[-8..1]}$$

$$\gamma(\bar{x}) = [\cos(\pi B \bar{x}), \sin(\pi B \bar{x})]$$

$$\text{FourierEncoding}(v) = \text{MLP}(\gamma(\bar{x}))$$

In other words, I am embedding the continuous fields by multiplying the tensor values with a sequence of bands, B , defined as $2^{\text{range}(-8, 2)}$ and then multiplying that again with π . I then compute both $\sin(\text{values} * B * \pi)$ and $\cos(\text{values} * B * \pi)$, concatenating these two sequences together to create γ . This tensor is then passed through a simple MLP to represent the tensor values to the neural network.

However, there is a small problem. Because I imputed the non-valued fields to 0.0 , how is the model able to differentiate a value that is masked, padded, null, or is actually a value but just happens to be equal to 0.0 ? The solution is surprisingly simple. I can simply embed the lookup tensor with standard learned embeddings. Because of some mathematical wizardry, I am able to add these two embeddings to fully represent each and every continuous field value, whether they are valued, null, padded, or masked!

This is due to several wonderful happenstances:

1. When values is equal to zero, $\gamma(\bar{x})$ will result in alternating ones and zeroes, so that $\text{MLP}(\gamma(\bar{x}))$ will result in a very simple FourierEncoding for non-valued indicators
2. The learned embeddings for the three learned non-valued indicator (UNK, PAD, and MASK) will learn around the simple FourierEncoding in the same way that the learned embeddings of vocabulary tokens used in BERT are able to learn around the sinusoidal positional embeddings.
3. The field's non-valued states are all effectively mutually exclusive (a field value can not be both padded and null at the same time). While a field can be both masked and null or masked and padded, the model should only ever see that it is masked.

This method also comes with a very beautiful runtime assertion to rigorously validate the two tensors values and lookup for each continuous field: Given that every non-valued input should be imputed to 0.0 , and that the VAL token itself is equal to 0 , we can *guarantee* that the element-wise product of lookup and values will always equal zero! This assertion prevents invalid field values from being processed during inference.

```
assert torch.all(values.mul(lookup).eq(0.0)) ,\
    "all values should be imputed to 0.0 if not null, padded, or masked"

assert torch.all(values.lt(1.0)) ,\
    "values should be less than 1.0"

assert torch.all(values.ge(0.0)) ,\
    "values should be greater than or equal to 0.0"
```

One last caveat to this approach is related to the precision of the floating point tensors. Fourier feature encodings have a limited range, which yet another reason I am using the CDFs of the original floating point values.

As I had mentioned above, UniTTab requires an ordinal classification pre-training task. In other words, we will need to classify which quantile bin a masked value should fall into. If we chose a total of 10 quantile bins than we would need to determine to which decile each of the masked values belong. The choice of quantile bins is a pre-training hyperparameter itself ($n_{\text{quantiles}}$), so it is much better to calculate the CDF of the original values for the pre-training task, and then when we choose a value for $n_{\text{quantiles}}$ we can simply calculate the appropriate bucket index by calculating $\lfloor n_{\text{quantiles}} * \text{CDF}(x) * 0.9 \rfloor$. That means that we can use the CDFs for the inputs of the model, and then modify the CDFs for the targets too!

FIXME add TDigest paper citation

In practice, Ted Dunning's TDigest algorithm is able to calculate the CDFs of the original values as a *streaming* operation during both model training and inference, which means we never have to calculate the CDFs as a batch operation. This is extremely valuable, especially at the scale of billions to trillions of data points, because otherwise the calculation of the CDFs would require a great deal of compute. This also saves a great deal of storage as well because we do not have to store the normalized values or the binned value targets. We only have to store the original values.

In practice, I found that the library `pytdigest`, developed by Tomas Protivinsky, provides a blazingly fast implementation of the TDigest algorithm. [7] All of the other implementations with which I have experimented are far too slow such that the streaming CDF calculations would bottleneck model training and inference.

There are potential alternatives to the aforementioned embedding strategy with which I have not yet experimented. Instead of embedding the values tensor with Fourier encodings and then embedding the lookup tensor with `torch.nn.Embedding` before adding those two embeddings together, we could instead perform an element-wise addition operation and embed the aggregated tensor once with `FourierEncoding`. The model will learn that values exactly equal to 1, 2, and 3 are each of a special token representation, and that the floating point inputs in the range of $[0, 1)$ are of actual values. You will need to expand the bands to $B = 2^{\lceil -8..1 \rceil}$ to account for the larger domain.

Regardless, we now have a strategy to embed a nullable continuous field of an arbitrarily complex numeric distribution. This is, to my understanding, a novel approach to this very specific problem. Now, let's look at how we may represent multiple, heterogeneous fields within a `TensorDict` input for the MFES.

```
@tensorclass
class ContinuousField:
    lookup: Int[torch.Tensor, "N L"]
    values: Float[torch.Tensor, "N L"]

# example input with a batch size of 2, context size of 8, and 2 continuous fields, 1 discrete field
TensorDict({
    amount: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    balance: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

The `ModularFieldEmbeddingSystem` is able to utilize instances of `ContinuousFieldEmbedder` for each of the continuous fields, each of which is dedicated to embedding a specific `ContinuousField`.

```
ModularFieldEmbeddingSystem(
    (embedders): ModuleDict(
        (amount): ContinuousFieldEmbedder(
            (linear): Linear(in_features=16, out_features=12, bias=True)
            # masked, null, padded, or valued
            (positional): Embedding(4, 12)
        ),
        (balance): ContinuousFieldEmbedder(
            (linear): Linear(in_features=16, out_features=12, bias=True)
            # masked, null, padded, or valued
            (positional): Embedding(4, 12)
        ),
        (transaction_type): DiscreteFieldEmbedder(
            # six unique transaction types plus padded, null or masked
            (embeddings): Embedding(9, 12)
        ),
    )
)
```

Similarly to the instance of `ModularFieldEmbeddingSystem` with only discrete fields, this `ModularFieldEmbeddingSystem` with heterogeneous field types will iterate over each of the six fields, embedding them individually and then concatenating their embeddings to create a single embedding for the entire batch of dimensionality $[N \ L \ F \ C]$ where $F = 3$.

Temporal Field Embeddings

Working with dates and timestamps might seem tricky, but we can actually tackle it similarly to how we handled continuous fields! Temporal fields (dates / timestamps) may be processed as raw numerical data via UNIX Epoch time, the number of seconds since 1970. That numerical data is then passed through a TDigest to retrieve the CDF of the timestamp. The CDF is then passed through an instance of TimestampFieldEmbedder to create a timestamp embedding. The usage of CDFs ensures that outlier date / timestamp values are still able to be understood by the model. Even though we are using epoch time, the extraordinarily large numbers (1 to 2 billion seconds since 1970) have no affect on the stability of the model. The TDigest model will learn the distribution of the values available in the training data.

Regardless, a TemporalField operates exactly like a ContinuousField, although it may be configured to have more or less quantiles during pre-training, and/or more or less bands (B) for varying precision.

```
@tensorclass
class TemporalField:
    lookup: Int[torch.Tensor, "N L"]
    values: Float[torch.Tensor, "N L"]

# example input with a batch size of 2, context size of 8
# 1 temporal field, 1 continuous field, 1 discrete field
TensorDict({
    timestamp: TemporalField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    amount: ContinuousField(
        lookup=Tensor(torch.Size([2, 8]), torch.int),
        values=Tensor(torch.Size([2, 8]), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([2, 8]), torch.int)),
})
```

In addition to passing in the raw timestamp data, you might instead include data related to the time difference since the previous event (duration), the number of seconds passed since the previous event. This could prevent the model from overfitting on irregular activity that is limited a small window of time that is not expected to generalize to the population of training data.

However, timestamps may still be important depending on your data. For example, the “Event Sourcing” pattern is not backwards compatible should the schema for your data change, such that some event types may not have been available in the past. It may be important for your use case to be able to explicitly tell the model about the absolute time an event took place may understand the API schema version at the time of any given event in order to prevent ambiguity. Alternatively, in this specific example you could also elect to pass in the API schema version number as a discrete field itself.

Geospatial Field Embeddings

Geospatial data also seems rather tricky. However, we can one again use a technique similar to the continuous field embedding mechanism to accomplish this. Unlike the continuous or temporal fields, geospatial coordinates are defined by two numbers: longitude and latitude. Both longitude and latitude requires their own TDigest model.

```
@tensorclass
class GeospatialField:
    lookup: Int[torch.Tensor, "N L"]
    longitude: Float[torch.Tensor, "N L"]
    latitude: Float[torch.Tensor, "N L"]

# example input with a batch size of 3, context size of 64
# 1 geospatial field, 1 temporal field, 1 continuous fields, 1 discrete fields
TensorDict({
```

```

location: GeospatialField(
    lookup=Tensor(torch.Size([3, 64]), torch.int),
    longitude=Tensor(torch.Size([3, 64]), torch.float),
    latitude=Tensor(torch.Size([3, 64]), torch.float),
),
timestamp: TemporalField(
    lookup=Tensor(torch.Size([3, 64]), torch.int),
    values=Tensor(torch.Size([3, 64]), torch.float),
),
amount: ContinuousField(
    lookup=Tensor(torch.Size([3, 64]), torch.int),
    values=Tensor(torch.Size([3, 64]), torch.float),
),
transaction_type: DiscreteField(lookup=Tensor(torch.Size([3, 64]), torch.int)),
})

```

However, using raw geospatial coordinates is not always the best course of action. If your data is too small it may lead to overfitting. If you are trying to create representations of ride-share drivers based on their previous trips, you might include both the pickup and dropout geospatial coordinate for each trip. Alternative, you could simply include the distance and zip code of the trip.

Additionally, the lines of longitude overflow at the prime meridian. This means that a coordinate with a longitude value of 179 is as close to the prime meridian as a coordinate value of -179. This is fundamentally different from how our numeral system works, and so it requires a slightly different strategy to decode during the self-supervised learning task.

Entity Field Embeddings

Entities are extremely common in production datasets. By an entity, I am referring to one discrete item in an infinitely large set of items. The set of items, while similar to categorical data, is so large or otherwise ephemeral that each unique level in the category cannot possibly be explained by a unique embedding.

For example, whenever a customer logs in to their account, I could create an ephemeral login session token that is unique. This login session token is stored as a string alongside all of the events that happen during the same login session. The sequence model could then use this login session ID to see which events happened during each login session.

We might also consider each unique “merchant” as an entity. We might want to be able to allow the model to see which purchases happened at each merchant. This is different than “merchant name”, which may include a rigorously cleaned string like “mcdonalds”. A “merchant” may instead include the address or store number, allowing even for a greater level of detail.

Another common example of an entity is the “account” of a customer, or the “customer” of an account. For some businesses, an “account” and a “customer” are not synonymous. If are defining a customer-level model (such that each sequence is defined by all of the events posted by a customer), but customers may have more than one account, you could represent each of the accounts available to a customer with an entity. The opposite is also true; if you are designing an account-level model (such that each sequence is defined by all of the events posted to an account), but each account may be accessed by more than one customer, you could represent each customer ID as an entity.

Lastly, we might consider the devices of each customer to be entities. Device identifiers may be tracked and stored across multiple events observed of each customer (IP address, unique smartphone application tracker, etc). These device identifiers may allow the model to understand which event happened from which of the customer’s devices, providing significant value for protecting customers theft or fraud.

Needless to say, effectively tracking the locations, devices and login sessions of customers may be considered a breach of privacy. However, in some cases, such as fraud detection, these data points are absolutely critical for optimal coverage in order to deliver the best customer experience.

The use of entities is very common in recommendation systems, which typically utilizes large lookup tables which are indexed with multiple lookup values created by many different hash functions in order to prevent hash collisions. However, there are other ways of accomplishing the same goal.

FIXME add Google unified embeddings paper citation

Instead of using these RecSys-style hash embeddings in order to uniquely represent every possible entity, we could instead elect to simply try to differentiate between the entities available within an observation's context. In other words, we don't necessarily need to know which device you are currently using, but we just need to know whether that device is the same that you used for other events within the context. This can actually be done by assigning each unique device in the context a single random number. We can guarantee that there will be, at most, the same number of unique devices equal to the size of the context window. The random lookup value assignment can change between each observation in order to prevent overfitting, but the assignment operation can be fixed to a seed value during inference for reproducibility.

```
def hash(values: list[str]) -> list[int]:
    '''Embeds list of strings into list of random, idempotent embedding lookup values
    ...

    offset = len(SpecialTokens)
    unique = set(values)
    integers = random.sample(range(offset, params.n_context + offset), len(unique))
    mapping = dict(zip(unique, integers))
    mapping.update({"[UNK]": Tokens.UNK})

    return list(map(lambda value: mapping[value], values))
```

Additionally, the usage of these “anonymous” entity embeddings has several benefits over traditional RecSys-style embeddings:

1. Able to be used when you have very little labeled data.
2. More appropriate when the entities are “ephemeral” (login sessions shouldn't persist indefinitely)
3. Prioritize customer privacy

```
@tensorclass
```

```
class EntityField:
```

```
    lookup: Int[torch.Tensor, "N L"]
```

```
# example input with a batch size of 3, context size of 64
```

```
# 1 entity field, 1 geospatial field, 1 temporal field, 1 continuous fields, 1 discrete fields
```

```
TensorDict({
    device_id: EntityField(lookup=Tensor(torch.Size([3, 64])), torch.int),
    location: GeospatialField(
        lookup=Tensor(torch.Size([3, 64])), torch.int),
        longitude=Tensor(torch.Size([3, 64])), torch.float),
        latitude=Tensor(torch.Size([3, 64])), torch.float),
    ),
    timestamp: TemporalField(
        lookup=Tensor(torch.Size([3, 64])), torch.int),
        values=Tensor(torch.Size([3, 64])), torch.float),
    ),
    amount: ContinuousField(
        lookup=Tensor(torch.Size([3, 64])), torch.int),
        values=Tensor(torch.Size([3, 64])), torch.float),
    ),
    transaction_type: DiscreteField(lookup=Tensor(torch.Size([3, 64])), torch.int),
})
```

This technique is only possible because we are defining a context window with an upper-bound size limit. If we have a context window with a maximum size of 64 then we know there are, at most, 67 maximum possible unique values that we need to represent (64 plus the three non-valued states). We are simply trying to differentiate these

unique values from one another. Devices, login sessions, and merchants are ephemeral and infinite, and this novel mechanism allows us to represent them as such without storing many different large look up tables.

Media Fields

Using standard field types (discrete, continuous, temporal, geospatial, entity field types) solves a great deal of common business problems. However, there are more advanced data types that may also be utilized, such as raw text, images, audio, and video. These are more complicated, and rely upon pre-trained models specific to each modality.

Textual Field Embeddings

We can also *potentially* use raw string text. This one is a little bit tricky, however. In effect, this is akin to running BERT inside of the modular field embedding system before passing the textual representation into two more transformer encoders. As you can imagine, this requires enormous amounts of memory for model training. The computational requirements are especially hard because the effective batch size for this BERT model becomes equal to $N * L * F$. In other words, you need to individually encode every textual single field of every event of every observation simultaneously.

For each event in each observation, we want to take the textual fields, tokenize their content, and then run them through BERT, collecting the text representation from the class token. There are two ways of doing this, neither of which are especially pleasant:

1. Use a single instance of a pre-trained BERT model and fine-tune it for all of the textual fields.
2. Use multiple instances of a smaller pre-trained BERT model and fine-tune them for each of the textual fields.

The BERT model would likely need to be limited to work with an extremely small token context in order to prevent running out of memory, hindering the quality of its field embeddings.

Additionally, pre-training textual field embedding modules is difficult! Running masked-language modeling *inside* our custom self-supervised learning task is not possible. That being said, if the BERT model is already pre-trained, it might be best to leave it frozen (thus slightly alleviating the memory requirements). At this current point in time, I do not plan on including a custom textual field event decoder to facilitate the self-supervised learning task. Instead, they will be frozen during pre-training, and may be unfrozen during fine-tuning.

```
@tensorclass
class TextualField:

    # the T dimension refers to the tokens within the field values
    lookup: Int[torch.Tensor, "N L T"]
```

Images, Video, and Audio Field Embeddings

In addition to working with just embedded textual fields, one could *theoretically* include images, audio, and video data as well. In other words, any event of any sequence could include a reference to a file. As a streaming operation, that file is fetched, opened, preprocessed, and embedded.

For example, suppose you are trying to create representations of all Twitter users. Each Tweet may include a photo or video, so that each Tweet event may optionally include a photo or video. In other words, each TweetEvent might include the following fields: timestamp, content, device, location, photo, video, is_deleted. The photo (or video) is then embedded, as are all of the other fields. Additionally, each Twitter user may include post a ProfilePictureUpdateEvent, which may include the following fields: timestamp, device, location, video.

Streaming the operations to read, decode, and collate images will certainly add complications to the data streaming pipeline, there are some ways to optimize it. For example, you could store each individual sequence as its own small file, and then store all of its associated media in separate files. All of these files for each sequence is compressed into a single TAR file, which could then be streamed through and sampled. If any of the media fields include a file path name, that file will be read from, decoded, and collated.

```
@tensorclass
class ImageField:
```

```
# the X and Y dimensions refers to pixel coordinates of each image
lookup: Int[torch.Tensor, "N L X Y"]
```

```
@tensorclass
class VideoField:
```

```
# the S dimension refers to each frame within the video
# the X and Y dimensions refers to pixel coordinates of each image
lookup: Int[torch.Tensor, "N L S X Y"]
```

The usage of media is extremely complicated, but really emphasizes the versatility of approaching your data with sequential modeling. In practice, using asynchronous streaming operations (RayData) will be key to scaling these more complex media fields. I do not currently have the expertise to experiment with these ideas.

Putting it all together: MFES and examples

Discrete, continuous, temporal, geospatial, entities, media, and external fields. MFES is able to work with all of these field types over large sequences of data. That means I am able to input sequences that describe all of the transactions ever posted by a customer of a financial institution:

Device ID	Action	Amount	Location	Timestamp	Comments	Merchant Name
e8e905fbb00f	Annual Fee	395.00		2022-03-19 08:34:02	Annual Card Fee	BigBank
	Purchase	171.54	(-74.204573, -158.007852)	2022-03-21 11:19:53	Home improvement supplies	HomeCraft
	Purchase	464.69	(-76.638826, -36.341502)	2022-04-30 20:43:02	Charity donation	GiveBack
e8e905fbb00f	Online Payment	463.58		2022-09-13 16:21:23	Hotel booking	StayInn
f2e31f29343b	Dispute Transaction	489.35	(-34.316692, 170.608224)	2023-05-01 20:40:15	Pet supplies	PetWorld
e8e905fbb00f	Refund	53.73		2023-06-09 06:55:00	Refund for returned item	ReturnDesk
e8e905fbb00f	Balance Inquiry			2023-08-23 01:48:57		BigBank
	Purchase	25.00	(81.202198, 175.041279)	2023-11-18 19:42:07	Coffee shop	BrewBeans
	Purchase	251.61	(-41.539246, 154.148774)	2023-12-01 22:12:23	Transfer to friend	PeerPay
e8e905fbb00f	Balance Inquiry			2024-02-11 09:38:10		BigBank

Table 6: An example sequence of mixed field types

This approach is extremely versatile. It is able to work with any data type. Additionally, it is able to work with your data exactly how it already is. Similar to language and vision, there is effectively zero batch preprocessing. There are no concerns about nullable fields values.

Model Architecture Summary

Upon the creation of the field embeddings with the Modular Field Embedding System, TabBERT's hierarchical transformer-encoder architecture is able to build sequence and event representations as before. I include two transformer-encoder modules, the "field encoder" and the "event encoder". The field encoder allows each field to

attend to every other field within each event. The contextualized field representations are then concatenated to form event representations for every event. Each event representation then attends to every other event representation in the event encoder, which produces contextualized event representations, as well as a sequence representation via a special [CLS] token, which is appended to the inputted event representations. The event representations are used to pre-train the model, and the sequence representation can be used for arbitrary supervised learning problems.

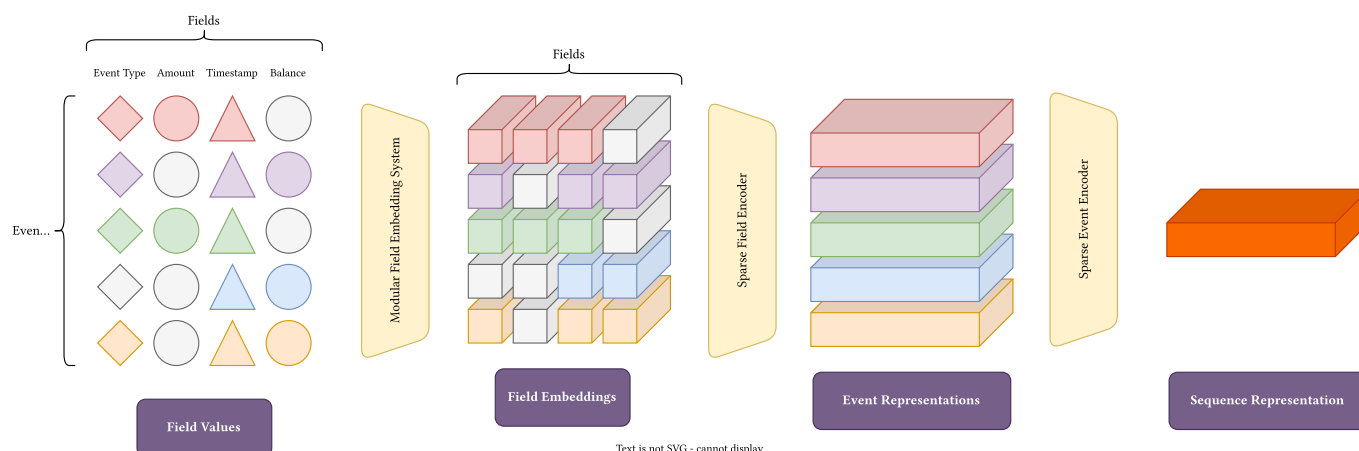


Figure 4: The consolidated model architecture outputting a sequence representation

Self-Supervised Learning Strategy

One of the many advantages of this network architecture is the opportunity to utilize self-supervised learning (SSL). In many business problems, there is a large amount of unlabeled data that traditional machine learning models are not able to utilize. The modality of language has benefitted enormously by applying masked language modeling (MLM) to pre-train models like BERT. TabBERT, as the name implies, also utilized an adaption of MLM, in which they mask random fields or events.

Masked Event and Masked Field Modeling Tasks

Applying SSL to our data is tricky, however, because every field can be of a different type. We needed to ensure that each field type is able to support non-valued states so that we are able to mask any field. In addition to being able to mask any field, we also need to be able to decode the contextualized event representations back into their original values. This is very easy for a discrete field, however it is more difficult for the other fields.

Device ID	Action	Amount	Location	Timestamp	Comments	Merchant Name
e8e905fbb00f	[MASK]	395.00		2022-03-19 08:34:02	Annual Card Fee	BigBank
	Purchase	171.54	(-74.204573, -158.007852)	2022-03-21 11:19:53	[MASK]	HomeCraft
[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]
e8e905fbb00f	Online Payment	463.58		2022-09-13 16:21:23	Hotel booking	StayInn
[MASK]	Dispute Transaction	489.35	(-34.316692, 170.608224)	2023-05-01 20:40:15	Pet supplies	PetWorld
[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]
e8e905fbb00f	Balance Inquiry			2023-08-23 01:48:57		BigBank
[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]	[MASK]
	Purchase	251.61	(-41.539246, 154.148774)	[MASK]	Transfer to friend	PeerPay
[MASK]	Balance Inquiry		[MASK]	2024-02-11 09:38:10		BigBank

Table 7: An example masked sequence

Semi-Ordinal Classification of Quantiles

For the continuous, temporal, and geospatial fields, instead of attempting to reconstruct the original CDFs, we can instead attempt to reconstruct the binned CDFs. This requires that we set a hyperparameter, `n_quantiles`, to define the number of bins for the customer SSL task. Going forward, let’s assume we want to calculate the number of deciles (`n_quantiles = 10`).

If one of the masked continuous fields has a CDF of 0.05 , the model should learn to predict that this value belongs in the 1st decile. If another masked continuous field has a CDF of 0.51 , the model should learn that this value belongs in the 6th decile. Every masked CDF should be mapped to the bin $\lfloor \text{CDF}(x) \cdot n_quantiles \cdot 0.9 \rfloor$.

However, this alone is imperfect. We are penalizing the model if it guesses the wrong bin, but we also want encourage the model if it guesses the incorrect bins that are *closer* to the correct bin. In other words, if the model says that the masked value belongs in the 8th decile, but in reality it belongs in the 10th decile, then it should be punished less than if it had said the masked value belongs in the 1st decile. In other words, we want to run a semi-ordinal classification task. We need to apply the “neighborhood loss smoothing” function from the UniTTab paper, however we need to modify it in order to support the non-valued states of null and padded as well.

If the a masked field is really “padded”, but the model guessed that the value is in any decile (or is null), we want to fully penalize the model. Additionally, if a masked field is of any decile, but the model predicted that masked field is either null or padded, then we should fully penalize the model. The “neighborhood loss smoothing” only applies if the masked field is valued, and even then it only applies among valued quantile bins.

Geospatial fields are a little more complicated, however, because lines of longitude wrap around at the prime meridian. This requires an adaption such that the 1st decile is as close to the 10th decile as the 9th decile. This is only relevant to the longitude tensor of geospatial fields.

Fine-tuning Sequence Models

The fine-tuning of this sequence model is pretty straightforward. The sequence representation is a tensor of fixed size. The sequence representation is passed through a small MLP that can be trained to resolve arbitrarily supervised learning tasks, including classification and regression. By fine-tuning the sequence representation,

you may also use the sequence representation for similarity search or clustering problems as well. In my experience, a properly pre-trained sequence model can be fine-tuned in a matter of hours on modest hardware.

However, because it is not of a universal modality like language, vision, video, or audio, the pre-trained models are only applicable among problems that share their fields.

Versatility of Supervised Tasks

Sequence models are able to support a variety of tasks. In my implementation, I have designed my sequence model to support event-level supervised tasks: classification and regression. In other words, I can have different target labels depending on which event was sampled. This is in contrast to sequence-level tagging, which would restrict the target to be homogeneous for every event in a sequence. Additionally, I have enforced a mechanism that I call “nullable tagging”, such that I can include null values in my targets. Events with null targets are able to be used in the historical context of future events, but they can not be sampled as events. I found this mechanism useful for solving the “clipping problem” which I describe in a later section

Observation Sampling Strategies

Sampling events from a sequence to create observations is difficult. Sampling too many overlapping observations from a sequence can quickly result in overfitting. By “overlapping observations”, I am referring to observations that share events with one another. I have developed an intuition that works well for me. I found success in iterating over each sequence in the population of sequences. Within each sequence, I will sample $(\text{len}(\text{sequence}) / n_{\text{context}})$ many events on average. In other words, I will sample random events from a sequence at a rate of $(1 / n_{\text{context}})$.

This event-sampling technique is unique to this problem. In computer vision, you can uniformly sample from the population of images. In language modeling, you can uniformly sample from the population of documents from a corpus. However, for sequence modeling, sampling observations at a sequence-level does not necessarily work because some sequences may have very few events. For example, you may be running a financial institution with 10 million customers, each with a credit card. However, a large portion of your credit card users may be “gamers” that only signed up for the card to collect a sign on bonus. These inactive customers may only have tens of events on their records, whereas other customers may have thousands of events in their records.

Sampling uniformly from each customer’s sequence will result in sampling the same few transactions from your inactive customers a hundred times before you have sampled through the events of your active customers. This results in extraordinarily long time to train and poor performance because the model has overfit on the less active customers and is left underfit on the more active customers.

Additional Considerations

Data Preprocessing for Efficient Streaming Operations

I found success in storing the data in a row-major format for custom streaming operations. ndjson and avro are both good options. I found that parquet had issues supporting complex, nested fields. I store my data in many small avro files, in which each sequence was stored as an object.

Within each sequence object, I stored the `sequence_id(str)`, `event_id(list[str])`, event fields (`dict[str, list[int] | list[float] | list[str]]`), and event targets (`list[int] | list[float]`). With this format, I may simply sample any event index number, and then slice the context directly from the event fields (`slice(max(0, idx - L), index)`), which can be trivially padded and collated to a tensor, which may then be stored in a `TensorDict` of tensorclasses.

This data structure ensures that the grouping and filtering operations required to collect all of the events for each sequence will not bottleneck the model training process. It optimizes streaming larger-than-memory sequences from disk to accelerate model training and inference. As a reference to the `pytorch-lifestreams` python library, developed primarily by Dmitri Babaev, I refer to this data structure as a *lifestream*.

As mentioned above, this preprocessing pipeline requires the calculation of CDFs as a streaming operation. This operation can be extremely expensive. The `TDigest` implementation specifically from the `pytdigest` package is

extremely optimized. According to their benchmarks, they are 10x to 1000x faster than other such open source implementations.

As a sidenote, performance may be further improved by utilizing another data format, such as protobuf. I expect that protobuf, as an alternative to avro, would be approximately 30% faster to decode at a negligible cost (~10%) to the size of compressed files.

Sequence Entropy

A unique characteristic of such multivariate sequences is that they are not nearly as structured as human language. Multivariate sequences can be quite noisy, whereas human language is bound by the rules of grammar. Human language has evolved to be structured, redundant, and fault-tolerant. For example, An adjective must always be followed by another adjective or the noun they describe. If you were to shuffle all of the words in a sentence, the fact that it was changed will be extraordinary obvious. However, if one were to just randomly mask out a random word, you may still have a very good guess as to the general message of the sentence regardless.

On the contrary, our complex sequences of data tend to have relatively less structure. If you shuffle all of the events in a sequence, it will likely remain just as “plausible” as before. However, if you were to mask out a random event, it will be much more difficult to reconstruct. For example, if you had to go both grocery shopping and also top your tank with gas, what portion of the time do you go to the grocery store *before* you go to the gas station? There is no right answer. The order is almost completely arbitrary.

Additionally, pre-training a large language model on every human language is extremely complicated due to large vocabulary sizes, disproportionate word usage, quotes, mannerisms, plagiarism, and writing styles. In my experience pre-training sequence models with transactional data is extremely simple in comparison because there is less structure to be learned. The hardware requirements are surprisingly minimal.

Arbitrary Dimensionality

In all of my examples above, I use a specific dimensionality: Each sequence contains events, and each event contains fields. In other words: Customer > Transaction > Field. This is extremely simple, and fits the vast majority of use cases that come to mind. However, I realize there are counter examples to this design.

Walmart may want to model their transactions differently. For example, they might want to include the concept of “shopping trips”, in which each customer might make multiple shopping trips, and during each shopping trip they might purchase multiple items, and each item may have multiple fields.. So, instead of Customer > Transaction > Field, we might instead model the sequence as Customer > Trip > Item > Field.

While the implementation of the model will change completely, the concept is not all that different. The usage of sequential modeling is still applicable. I do not currently intend on developing this 3D implementation, however.

The Clipping Problem

I came across an interesting complication while training sequential models on slices of time series. It is intuitive in hindsight, but it is challenging to identify.

It is common to subset slices of time series based on some time window (IE: all events between January 2020 and January 2023). This can lead to very unexpected behavior, however, due to what I have called the “clipping problem”.

Simply put, the model may sometimes not have enough information to understand if a sequence with an event in the onset of the window is truly the first event in the sequence, or if it just looks like the first event in the sequence because the prior events were filtered out. There is ambiguity in what had happened before the context.

This problem has multiple solutions:

1. Include a “buffer” window in the data, during when the model may sample the events as historical context, but not as the event at the end of an observation
2. Include information regarding the start of the sequence, such as the creation of the account (AccountOpenEvent) or a field that provides the time since the sequence started (account_tenure)
3. Instead of creating a fixed time window (Jan 2020 to Jan 2023) at an event level, perhaps filter your data to only accounts that were opened between some two dates (depending on your use case)

All three of these solutions prevents the ambiguity between an observation that includes an artificial cutoff versus an observation that includes the start of a sequence.

The Flushing Problem

A unique adversarial vulnerability arises when modeling sequential data with a fixed context size depending on the types of events allowed to fill the context window. Theoretically, an ill-intended individual could “flush” the context by spamming innocuous events. In other words, an attacker may simply “login” and “logout” of their account several thousand times in order to attempt to manipulate the model in some way.

A more concrete example: A savvy hacker, having recently hacked into their victim’s account at Big Online Store, could choose to reset the user’s password, email, phone number, and mailing address before attempting to purchase gift card codes. Such risky actions, which may otherwise result in a security alert, may go unnoticed if the fraudster intentionally spams many login events between each nefarious action, such that the model doesn’t see more than one such nefarious action at any point in time. In other words, if you perform a thousand low-risk events between every high-risk event, the model will not see just how risky your recent actions truly are.

The risk of such an attack may be mitigated by multiple means:

1. Increase the context size.
2. Set up policies to flag high event rates, where needed.
3. Do not include trivial events (login / logout events) in the context, but rather represent them whenever possible, as entity fields (such as a “login session”)

Including Tabular Data and Multiple Sequences of Varying Event Rates

I have not yet explored integrating static tabular data with my sequential data, although this approach has been explored by Visa Research in their FATA-Trans paper. I am similarly curious about working with multiple sequences of data per account, in which each sequence has different event rates. For example, one sequence of 48 monthly billing statements over the last four years, another sequence for the last 256 customer financial transactions, and a final sequence of the last 2048 online click stream events. This would allow the model to learn from multiple sources without low frequency events being excavated from the context by high frequency events.

The most significant obstacle to implementing multiple sequences of varying event rates is the systematic synchronization of these events as a streaming operation. For example, which monthly statements should be available to the context given which online click stream events? This would require each click stream event to contain an index to its corresponding monthly statement.

Regardless, the utilization of multiple sequences and tabular data is yet another solution to the aforementioned “flushing problem” because the signal may be captured from multiple potential sequences of varying event rates.

Sub-Quadratic Alternatives to Self-Attention

An interesting property of sequential data is that events asynchronously, arriving in separate streaming messages. Consequently, there is a potential optimization trick for real-time inference. Using emerging alternatives to transformers, such as RWKV or Mamba, one can actually save an enormous amounts of the compute requirements for inference. The memory complexity with respect to the context size decreases from quadratic to constant!

- Store sequence state representations in a in-memory cache like Redis
- Whenever a new event arrives:
 1. Let each field attend to one another within the new event
 2. Concatenate the attended fields to create a contextualized event representation
 3. Retrieve the current contextualized event representation hidden state
 4. Update the hidden state with the new event representation
 5. Pass the new hidden state to a decision head to create and return a new model score

Conclusion

Conclusion: use sequential modeling! Its better. Tabular machine learning sucks and you are wasting hundreds of millions of dollars for no reason.

Bibliography

- [1] Peltarion, “How to get meaning from text with language model BERT | AI Explained.” [Online]. Available: <https://www.youtube.com/watch?v=-9vVhYEXeyQ>
- [2] I. Padhi *et al.*, “Tabular Transformers for Modeling Multivariate Time Series.” 2021.
- [3] S. Luetto, F. Garuti, E. Sangineto, L. Forni, and R. Cucchiara, “One Transformer for All Time Series: Representing and Training with Time-Dependent Heterogeneous Tabular Data.” 2023.
- [4] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis.” 2020.
- [5] M. Tancik *et al.*, “Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains.” 2020.
- [6] D. Zhang *et al.*, “FATA-Trans: Field And Time-Aware Transformer for Sequential Tabular Data,” in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, in CIKM ’23. ACM, Oct. 2023. doi: [10.1145/3583780.3614879](https://doi.org/10.1145/3583780.3614879).
- [7] T. Protivinsky, “PyTDigest.” GitHub, 2023.