@twittereng

Only on Twitter

@Twitter
#OnlyOnTwitter

Insights

Embeddings@Twitter

By @twittereng
Thursday, 13 September 2018

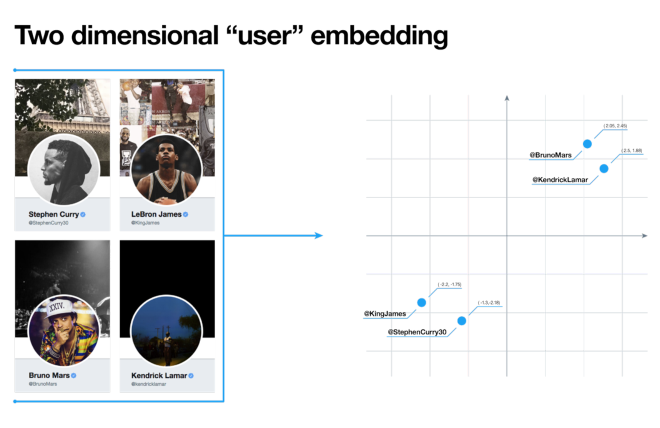
Machine-learning models are used across Twitter to enhance the product and serve the public conversation. The data that supports these models is often extremely large, complex, and constantly changing. At Twitter, we represent this information in the form of embeddings. Generating and sharing high-quality, up-to-date embeddings enables teams to effectively leverage various forms of data, improve the performance of ML models, and decrease redundant efforts.

In this blog post we discuss the commoditized tools, algorithms, and pipelines that we develop at Twitter to regularly generate embeddings for Twitter entities. This enables us to share them broadly across the company, thus making embeddings a first-class citizen of Twitter's ML infrastructure. We will also detail how creating reliable offline qualitative benchmarks helped us ensure high quality and achieve quick iteration speed.

What is an embedding?

Simply put, an embedding is a transformation of input data into a more useful representation — a list of real numbers, called a vector. Note that the usefulness of the representation can take on a different meaning depending on the domain. For example, if we are embedding words, then we want to ensure the embeddings contain some sense of semantic meaning. But if feature compression (explained below) is the goal, then an embedding will be useful if it is compact (low-dimensional) without losing too much information.

The image below illustrates the concept of representing users as two-dimensional vectors visualized as coordinates on a graph. We will revisit this concept later; take note that more similar users are closer together on the graph.



An embedding is usually associated with an entity, which we'll say is an instance of some discrete type of interest such as a user, Tweet, author, word, or phrase, etc. These entity embeddings (in the mapping sense) can be decomposed into two distinct classes: static and dynamic embeddings. A static embedding is an embedding of entities such that every entity has one and only one embedding value. A dynamic embedding, on the other hand, is an embedding of entities such that each entity can have more than one embedding value. We use the term dynamic here because the changing embeddings for an entity are often a function of time. Often people use embedding to refer to both the transformation (verb) as well as the particular values of the transformation (noun).

There are several practical uses of embeddings with which readers may already be familiar. In natural language processing (NLP), an embedding is a representation of a word, phrase, or document as a vector with length typically much smaller than the vocabulary, with the extra property that the embedding preserves some sense of semantic meaning. In image or video processing, an embedding can refer to a compression of the input media pixels into a smaller space. Finally, matrix factorization methods used in collaborative filtering assign each user and each item an embedding from a row or column of a matrix.

Why embeddings?

In this section we explain the leading benefits of using embeddings at Twitter.

Model features

Most ML algorithms understand one kind of input — a vector. However, data rarely fits nicely into that form: e.g., Twitter users. Traditionally, in order to represent a user as a vector, a ML practitioner will employ techniques such as one-hot encoding. In this case, each user is mapped to a different dimension of a vector space of fixed size. While this method works well for some cases, the resulting vector can have hundreds of millions of dimensions and only contain a small amount of meaningful information.

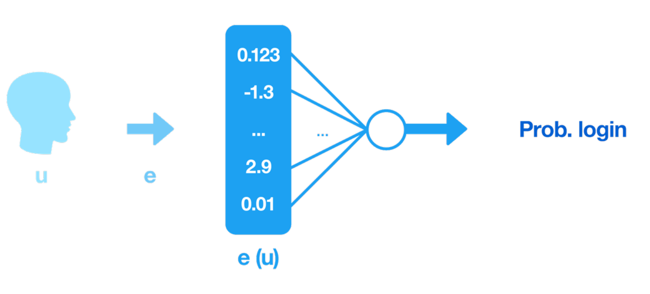
Entity embeddings, or learned representations, try to address this problem. Embeddings are themselves an output of other machine-learning models, trained directly on the sparse data.

Such models compress the high-dimensional feature space into a dense, lower-dimensional space while preserving salient information about the original entity. For a clarifying example, let's train user embeddings using follower relationship as input data as described above and take the embeddings corresponding to the users: Stephen Curry (@StephenCurry30), Lebron James (@KingJames), Bruno Mars (@BrunoMars), and Kendrick Lamar (@kendricklamar). We expect the distance between the embeddings of the NBA players to be smaller than the distance between the embeddings of a player and a musician. If we denote with e(user) the embedding of the user, then what we are saying is

```
dist(e(@StephenCurry30), e(@KingJames)) < dist(e(@KingJames), e(@BrunoMars))
```

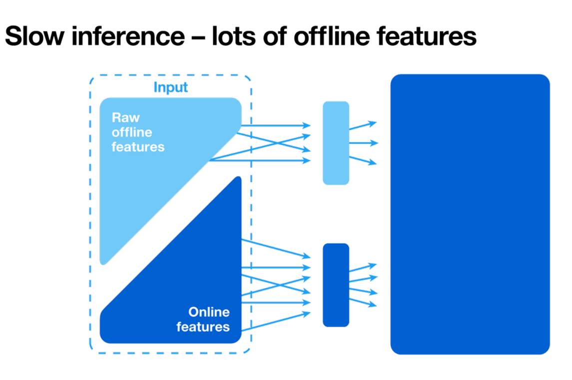
where dist is the Euclidean distance.

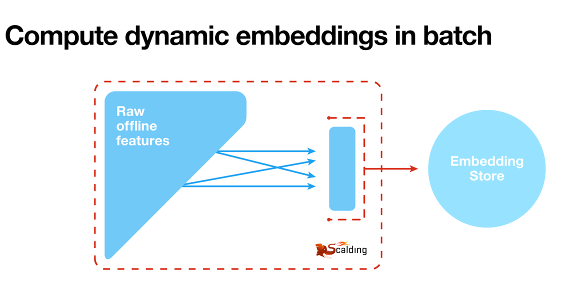
A model using embeddings as input features will benefit from their encoded knowledge, and therefore improve performance. On top of that, assuming compactness of the embeddings, the model itself will require fewer parameters, resulting in faster iteration speed and cost savings in terms of infrastructure during both training and serving.

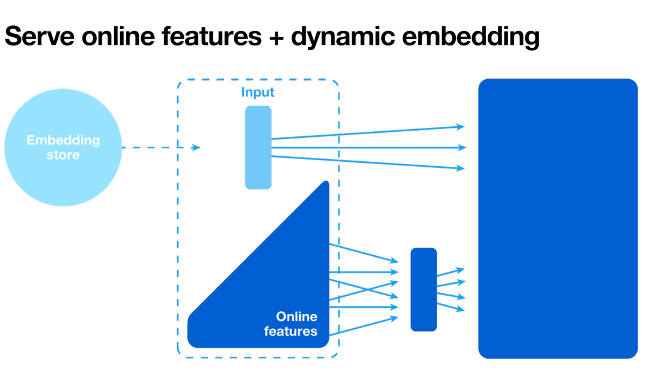


Feature compression

As ML engineers continue to refine and improve a model, the number of input features may grow to such a size that online inference slows to the point where adding more features is intractable. Precomputed embeddings offer a solution if some of the features are known prior to inference time and can be grouped together and passed through one or more layers of the network. This "embedding subnetwork" can be run in batch offline or in an online prediction service. The returned output embeddings will then be stored for online inference, where latencies will be much faster than if the full network had to be evaluated.







Nearest neighbor search

Many product surfaces at Twitter aim to make some kind of recommendation to users, whether they be other users to follow, Tweets to read, or videos to watch. Usually the first step in recommendations is to generate from the entire collection of items a smaller set of quality candidates (candidate generation). Being able to find similar items is an essential task for many candidate generation schemes, and one way to accomplish it is to find an embedding of items and a measure of distance between them such that similar items have embeddings that are close together. This search based on distance between items is called nearest neighbor search. This technique finds use in a variety of applications such as search query expansion and recommendation systems.

Transfer learning

Embeddings are also often used in the context of transfer learning, which is a general machine-learning strategy where a model trained for one task is used in another. A simple example of this is using a trained, generic image model (typically a convolutional neural net) on a new image task by using the parameters of the original network as the starting point for training the new one. Using trained model parameters as the initialization of a new network and then letting the parameters be updated via the learning method is called fine-tuning and can give faster convergence and better-performing algorithms.

We will also note quickly that from a business perspective transfer learning is a very attractive method since it can reduce the development time to a first shippable model and help leverage information learned from disparate areas of the product.

Goals:

As we develop the systems and processes to enable widespread use of embeddings at Twitter, it's worth spelling out goals for development of the same that we defined for ourselves, and realize that sometimes we will be forced to make trade-offs between them.

Our development goals:

- Quality and relevance
- Creation and consumption with ease
- Sharing and discoverability

Quality and relevance

We want embeddings that help us build great ML models. We want quality embeddings that provide meaningful entity representations. But we also must realize that the quality of an embedding may degrade over time. User behavior evolves, and so must their embedding representation. Similarly, the underlying meaning of words will change with time. We need to make sure our embeddings are relevant at the time of their application and, if required, that they are general enough to be of use across a variety of models (see "Embedding pipeline" section below).

Creation and consumption with ease

We should provide plug-and-play components that allow customers to learn embedding maps on their own and then generate the embeddings themselves at scale with ease. If a customer just wants to plug a widely applicable canonical embedding in their model they should have the tools to do so without having to care about how the embeddings were trained or stored.

Sharing and discoverability

Embeddings are not the end goal but an intermediate tool that, if sufficiently general, can be used in many tasks. Therefore, to maximize their usefulness they should be easily discovered and applied on new problems. We must consider from the outset how this sharing will take place and its consequences and side effects, both from engineering and modeling perspectives.

Embedding pipeline:

Since learned embeddings are machine-learning models, they require data to train and may be cumbersome to store and deploy. Furthermore, embeddings need to be regularly retrained and benchmarked — especially in a constantly changing system like Twitter. In order to address these issues and reap the benefits of embeddings, we have developed a series of tools that make it simple for teams throughout Twitter to customize, develop, access, and share embeddings.

Embedding generation pipelines often consist of a sequence of steps that can be difficult to maintain and reuse. In order to address this, we implement them within the Twitter ML Workflows platform — a system built on top of Apache Airflow that links data processing and ML components into reusable pipelines that can be configured with a web interface. This makes it much easier for teams to share steps between pipelines, keep embeddings up to date, and modify pipelines to publish customized embeddings. Each of our embedding generation pipelines consist of the following steps:

Item selection. In this step, we identify the set of items to generate embeddings for.

- In our word embedding pipelines, we use this step to select the tokens (hashtags, usernames, emojis, words, URLs, etc.) that we will assign embeddings to
- In some of our user embedding pipelines, we use this step to identify the users who have enough information to assign an embedding to.

Data preprocessing. In this step, we assemble a dataset of entity relationships that we can use to train the embedding model. We generally perform this step in Scalding, Twitter's Scala-based map-reduce engine.

- In our skip-gram word embedding pipeline, we use this step to form a set of *(word_1, word_2)* skip-gram pairs, where *word_1* appeared near *word_2* in a recent Tweet.
- In our user graph embeddings pipeline, we use this step to construct a dataset of *(user_1, user_2)* pairs where *user_1* and *user_2* are connected through Twitter's follow or engagement graph.

Model fitting. In this stage we fit a model on the data that we have collected. We use a variety of algorithms for model fitting, including matrix factorization, linear gradient-based approaches, and deep neural networks.

- In our skip-gram word embedding pipeline, we use a gradient-descent and negative-sampling based approach to assign embeddings to words from their skip-gram pairs.
- In our follow graph SVD pipeline, we use an SVD algorithm to convert the adjacency matrix that represents Twitter's follow graph into a set of embeddings for Twitter users.

Benchmarking. Unlike with a classification or regression model, it's notoriously difficult to measure the quality of an embedding. One of the reasons for this is that different teams use embeddings differently. For example, while some teams use user embeddings as model inputs, others use them in nearest neighbor systems. To mitigate this problem we have developed a variety of standard benchmarking tasks for each type of embedding.

- User topic prediction. During onboarding, Twitter users may indicate which topics interest them. The ROC-AUC of a logistic regression trained on user embeddings to predict those topics is a measure of that embedding's ability to represent user interests.
- Metadata prediction. Certain users provide their demographic information (such as gender, age, etc). The ROC-AUC of a logistic regression trained on user embeddings to predict this metadata is a measure of how well that embedding might perform on a downstream machine-learning task.
- User follow Jaccard. We can estimate the similarity of two users' tastes by the Jaccard index of the sets of accounts that the users follow. Over a set of user pairs, the rank order correlation between the users' embedding similarity (as determined by the cosine distance between the users' embeddings) and their follow sets' Jaccard index is a measure of how well the embedding groups users.

Feature store registration. In the final step of our embedding pipeline, we publish the embeddings to the "feature store," Twitter's shared feature repository. This enables machine-learning teams throughout Twitter to easily discover, access, and use freshly trained embeddings.

What's next?

As we continue to work toward enabling product teams to use embeddings-based ML solutions, we are focused on further developing more new reusable algorithms and scaling the existing ones. We also would like to see adoption increase among product teams learning and publishing their own embeddings to the centralized feature store, thus creating the flywheel that powers ML models across Twitter.

Additionally, we are investing in creating scalable nearest neighbor lookup infrastructure such that product teams can utilize the learned embeddings beyond the model feature use case. We have also achieved promising results experimenting with embeddings as a means for feature compression and are looking forward to building on those results. We will continue to share our progress and lessons learned along the way as the team continues making progress.

Acknowledgements:

This blog was authored by Dan Shiebler (@dshieble), Chris Green (@cmgreen210), Luca Belli (@__lucab) and Abhishek Tayal (@tayal_abhishek), all members of Cortex MLX (ML Extended Environment) team.

We would like to thank all the members of the MLX team for their contributions toward shaping our technical vision and executing on it: Nimalan Mahendran, Max Hansmire, Apoorv Sharma, Shivam Verma, and Yang Wu. Honorable mention to management for supporting us in this endeavor: Nicolas Koumchatzky, Jan Pedersen, and Sandeep Pandey. We would also like to extend our gratitude to the product management team consisting of Matthew Bleifer, Tim Sweeney, and Theodore Summe for helping us define the product offering and ensuring we deliver the desired experience for our customers. Special thanks to Hanchen Xiong for collaborating with us in the early days of the effort and helping us make big strides. And finally, a special thanks to all the people at Twitter who continue to work with us and provide valuable feedback during our beta offering.