An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors.

Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words. Ideally, an embedding capture some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models.

**Embeddings Motivation from Collaborative Filtering**

**Collaborative filtering** is the task of making predictions about the interests of a user based on interests of many other users.

As an example, let's look at the task of movie recommendation. Suppose we have 1,000,000 users, and a list of the movies each user has watched (from a catalog of 500,000 movies). Our goal is to recommend movies to users.

To solve this problem some method is needed to determine which movies are like each other. We can achieve this goal by embedding the movies into a low-dimensional space created such that similar movies are nearby.

Before describing how we can learn the embedding, we first explore the type of qualities we want the embedding to have, and how we will represent the training data for learning the embedding.

**Arrange Movies on a One-Dimensional Number Line**

To help develop intuition about embeddings, on a piece of paper, try to arrange the following movies on a one-dimensional number line so that the movies nearest each other are the most closely related:

**Graphical user interface, text, application, email

Description automatically generated**

**Graphical user interface, application, Word, website

Description automatically generated**

**Arrange Movies in a Two-Dimensional Space**

Try the same exercise as before, but this time arrange the same movies in a two-dimensional space.

A picture containing application

Description automatically generated

With this two-dimensional embedding we define a distance between movies such that movies are nearby (and thus inferred to be similar) if they are both alike in the extent to which they are geared towards children versus adults, as well as the extent to which they are blockbuster movies versus arthouse movies. These, of course, are just two of many characteristics of movies that might be important.

More generally, what we've done is mapped these movies into an embedding space, where each word is described by a two-dimensional set of coordinates. For example, in this space, "Shrek" maps to (-1.0, 0.95) and "Bleu" maps to (0.65, -0.2). In general, when learning a d-dimensional embedding, each movie is represented by d real-valued numbers, each one giving the coordinate in one dimension.

In this example, we have given a name to each dimension. When learning embeddings, the individual dimensions are not learned with names. Sometimes, we can look at the embeddings and assign semantic meanings to the dimensions, and other times we cannot. Often, each such dimension is called a latent dimension, as it represents a feature that is not explicit in the data but rather inferred from it.

Ultimately, it is the distances between movies in the embedding space that are meaningful, rather than a single movie's values along any given dimension.

**Embeddings: Categorical Input Data**

Categorical data refers to input features that represent one or more discrete items from a finite set of choices.

For example, it can be the set of movies a user has watched, the set of words in a document, or the occupation of a person.

Categorical data is most efficiently represented via sparse tensors, which are tensors with very few non-zero elements.

For example, if we're building a movie recommendation model, we can assign a unique ID to each possible movie, and then represent each user by a sparse tensor of the movies they have watched, as shown in Figure 3.

Table

Description automatically generated with low confidence

Each row of the matrix in Figure 3 is an example capturing a user's movie-viewing history and is represented as a sparse tensor because each user only watches a small fraction of all possible movies. The last row corresponds to the sparse tensor [1, 3, 999999], using the vocabulary indices shown above the movie icons.

Likewise, one can represent words, sentences, and documents as sparse vectors where each word in the vocabulary plays a role similar to the movies in our recommendation example.

In order to use such [representations](https://developers.google.com/machine-learning/crash-course/representation/video-lecture) within a machine learning system, we need a way to represent each sparse vector as a vector of numbers so that semantically similar items (movies or words) have similar distances in the vector space. But how do you represent a word as a vector of numbers?

The simplest way is to define a giant input layer with a node for every word in your vocabulary, or at least a node for every word that appears in your data. If 500,000 unique words appear in your data, you could represent a word with a length 500,000 vector and assign each word to a slot in the vector.

If you assign "horse" to index 1247, then to feed "horse" into your network you might copy a 1 into the 1247th input node and 0s into all the rest. This sort of representation is called a **one-hot encoding**, because only one index has a non-zero value.

More typically your vector might contain counts of the words in a larger chunk of text. This is known as a "**bag of words**" representation. In a bag-of-words vector, several of the 500,000 nodes would have non-zero value.

But however, you determine the non-zero values, one-node-per-word gives you very *sparse* input vectors—very large vectors with relatively few non-zero values**. Sparse representations** have a couple of problems that can make it hard for a model to learn effectively.

**Size of Network**

Huge input vectors mean a super-huge number of weights for a neural network.

If there are M words in your vocabulary and N nodes in the first layer of the network above the input, you have MxN weights to train for that layer. Many weights cause further problems:

* **Amount of data.** The more weights in your model, the more data you need to train effectively.
* **Amount of computation.** The more weights, the more computation required to train and use the model. It's easy to exceed the capabilities of your hardware.

**Lack of Meaningful Relations Between Vectors**

If you feed the pixel values of RGB channels into an image classifier, it makes sense to talk about "close" values. Reddish blue is close to pure blue, both semantically and in terms of the geometric distance between vectors. But a vector with a 1 at index 1247 for "horse" is not any closer to a vector with a 1 at index 50,430 for "antelope" than it is to a vector with a 1 at index 238 for "television".

**The Solution: Embeddings**

The solution to these problems is to use embeddings, which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships. We'll explore embeddings intuitively, conceptually, and programmatically in the following sections of this module.

**Key Terms**

|  |  |
| --- | --- |
|  [input layer](https://developers.google.com/machine-learning/glossary#input_layer) |  [one-hot encoding](https://developers.google.com/machine-learning/glossary#one-hot_encoding) |
|  [sparse features](https://developers.google.com/machine-learning/glossary#sparse_features) | |

**Embeddings: Translating to a Lower-Dimensional Space**

You can solve the core problems of sparse input data by mapping your high-dimensional data into a lower-dimensional space.

As you can see from the paper exercises, even a small multi-dimensional space provides the freedom to group semantically similar items together and keep dissimilar items far apart. Position (distance and direction) in the vector space can encode semantics in a good embedding. For example, the following visualizations of real embeddings show geometrical relationships that capture semantic relations like the relation between a country and its capital:

Chart, scatter chart

Description automatically generated

This sort of meaningful space gives your machine learning system opportunities to detect patterns that may help with the learning task.

**Shrinking the network**

While we want enough dimensions to encode rich semantic relations, we also want an embedding space that is small enough to allow us to train our system more quickly.

A useful embedding may be on the order of hundreds of dimensions. This is likely several orders of magnitude smaller than the size of your vocabulary for a natural language task.

**Embeddings as lookup tables**

An embedding is a matrix in which each column is the vector that corresponds to an item in your vocabulary. To get the dense vector for a single vocabulary item, you retrieve the column corresponding to that item.

But how would you translate a sparse bag of words vector? To get the dense vector for a sparse vector representing multiple vocabulary items (all the words in a sentence or paragraph, for example), you could retrieve the embedding for each individual item and then add them together.

If the sparse vector contains counts of the vocabulary items, you could multiply each embedding by the count of its corresponding item before adding it to the sum.

These operations may look familiar.

**Embedding lookup as matrix multiplication**

The lookup, multiplication, and addition procedure we've just described is equivalent to matrix multiplication. Given a 1×n sparse representation S and an n×m embedding table E, the matrix product SE gives you a 1×m dense vector.

But how do you get E in the first place? We'll take a look at how to obtain embeddings in the next section.

**Key Terms**

|  |
| --- |
|  [embeddings](https://developers.google.com/machine-learning/glossary#embeddings) |

**Embeddings: Obtaining Embeddings**

There are a number of ways to get an embedding, including a state-of-the-art algorithm created at Google.

**Standard Dimensionality Reduction Techniques**

There are many existing mathematical techniques for capturing the important structure of a high-dimensional space in a low dimensional space. In theory, any of these techniques could be used to create an embedding for a machine learning system.

For example, [principal component analysis](https://wikipedia.org/wiki/Principal_component_analysis) (PCA) has been used to create word embeddings. Given a set of instances like bag of words vectors, PCA tries to find highly correlated dimensions that can be collapsed into a single dimension.

**Word2vec**

Word2vec is an algorithm invented at Google for training word embeddings. Word2vec relies on **the distributional hypothesis** **to map semantically similar words to geometrically close embedding vectors.**

The **distributional hypothesis** states that words which often have the same neighbouring words tend to be semantically similar.

Both "dog" and "cat" frequently appear close to the word "vet", and this fact reflects their semantic similarity. As the linguist John Firth put it in 1957, "You shall know a word by the company it keeps".

Word2Vec exploits contextual information like this by training a neural net to *distinguish* **actually co-occurring groups of words** from **randomly grouped words**.

The input layer takes a sparse representation of a target word together with one or more context words. This input connects to a single, smaller hidden layer.

In one version of the algorithm, the system makes a negative example by substituting a random noise word for the target word. Given the positive example "the plane flies", the system might swap in "jogging" to create the contrasting negative example "the jogging flies"

The other version of the algorithm creates negative examples by pairing the true target word with randomly chosen context words. So, it might take the positive examples (the, plane), (flies, plane) and the negative examples (compiled, plane), (who, plane) and learn to identify which pairs actually appeared together in text.

The classifier is not the real goal for either version of the system, however.

After the model has been trained, you have an embedding. You can use the weights connecting the input layer with the hidden layer to map sparse representations of words to smaller vectors. This embedding can be reused in other classifiers.

For more information about word2vec, see the [tutorial on tensorflow.org](https://www.tensorflow.org/tutorials/word2vec/index.html)

**Training an Embedding as Part of a Larger Model**

You can also learn an embedding as part of the neural network for your target task. This approach gets you an embedding well customized for your particular system, but may take longer than training the embedding separately.

* In general, when you have sparse data (or dense data that you'd like to embed), you can create an embedding unit that is just a special type of hidden unit of size d.
* This embedding layer can be combined with any other features and hidden layers.
* As in any DNN, the final layer will be the loss that is being optimized.

For example, let's say we're performing collaborative filtering, where the goal is to predict a user's interests from the interests of other users.

We can model this as a supervised learning problem by randomly setting aside (or holding out) a small number of the movies that the user has watched as the positive labels, and then optimize a softmax loss.

Diagram

Description automatically generated

As another example if you want to create an embedding layer for the words in a real-estate ad as part of a DNN to predict housing prices then you'd optimize an L2 Loss using the known sale price of homes in your training data as the label.

When learning a d-dimensional embedding each item is mapped to a point in a d-dimensional space so that the similar items are nearby in this space.

Figure 6 helps to illustrate the relationship between the weights learned in the embedding layer and the geometric view. The edge weights between an input node and the nodes in the d-dimensional embedding layer correspond to the coordinate values for each of the d axes.

Diagram

Description automatically generated