



Word embeddings

 [Run in Google Colab \(https://colab.research.google.com/github/tensorflow/text/blob/master/docs/guide/word_embeddings.ipynb\)](https://colab.research.google.com/github/tensorflow/text/blob/master/docs/guide/word_embeddings.ipynb)

This tutorial contains an introduction to word embeddings. You will train your own word embeddings using a simple Keras model for a sentiment classification task, and then visualize them in the [Embedding Projector](http://projector.tensorflow.org) (<http://projector.tensorflow.org>) (shown in the image below).

 Screenshot of the embedding projector

Representing text as numbers

Machine learning models take vectors (arrays of numbers) as input. When working with text, the first thing you must do is come up with a strategy to convert strings to numbers (or to "vectorize" the text) before feeding it to the model. In this section, you will look at three strategies for doing so.

One-hot encodings

As a first idea, you might "one-hot" encode each word in your vocabulary. Consider the sentence "The cat sat on the mat". The vocabulary (or unique words) in this sentence is (cat, mat, on, sat, the). To represent each word, you will create a zero vector with length equal to the vocabulary, then place a one in the index that corresponds to the word. This approach is shown in the following diagram.

One-hot encoding

	cat	mat	on	sat	the
the =>	0	0	0	0	1
cat =>	1	0	0	0	0
sat =>	0	0	0	1	0
...					

To create a vector that contains the encoding of the sentence, you could then concatenate the one-hot vectors for each word.

Key Point: This approach is inefficient. A one-hot encoded vector is sparse (meaning, most indices are zero). Imagine you have 10,000 words in the vocabulary. To one-hot encode each word, you would create a vector where 99.99% of the elements are zero.

Encode each word with a unique number

A second approach you might try is to encode each word using a unique number. Continuing the example above, you could assign 1 to "cat", 2 to "mat", and so on. You could then encode the sentence "The cat sat on the mat" as a dense vector like [5, 1, 4, 3, 5, 2]. This approach is efficient. Instead of a sparse vector, you now have a dense one (where all elements are full).

There are two downsides to this approach, however:

- The integer-encoding is arbitrary (it does not capture any relationship between words).
- An integer-encoding can be challenging for a model to interpret. A linear classifier, for example, learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

Word embeddings

Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding. Importantly, you do not have to specify this encoding by hand. An embedding is a dense vector of floating point values (the length of the vector is a parameter you specify). Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer). It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words, but takes more data to learn.

A 4-dimensional embedding

cat =>	1.2	-0.1	4.3	3.2
mat =>	0.4	2.5	-0.9	0.5
on =>	2.1	0.3	0.1	0.4
...				

Above is a diagram for a word embedding. Each word is represented as a 4-dimensional vector of floating point values. Another way to think of an embedding is as "lookup table". After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.

Setup

```
import io
import os
import re
import shutil
```

```
import string
import tensorflow as tf

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Embedding, GlobalAveragePooling1D
from tensorflow.keras.layers import TextVectorization
```

```
2022-12-14 12:16:56.601690: W tensorflow/compiler/xla/stream_executor/platform/d
2022-12-14 12:16:56.601797: W tensorflow/compiler/xla/stream_executor/platform/d
2022-12-14 12:16:56.601808: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:
```

Download the IMDb Dataset

You will use the [Large Movie Review Dataset](http://ai.stanford.edu/%7Eamaas/data/sentiment/) (<http://ai.stanford.edu/%7Eamaas/data/sentiment/>) through the tutorial. You will train a sentiment classifier model on this dataset and in the process learn embeddings from scratch. To read more about loading a dataset from scratch, see the [Loading text tutorial](https://www.tensorflow.org/tutorials/load_data/text) (https://www.tensorflow.org/tutorials/load_data/text).

Download the dataset using Keras file utility and take a look at the directories.

```
url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

dataset = tf.keras.utils.get_file("aclImdb_v1.tar.gz", url,
                                  untar=True, cache_dir='.',
                                  cache_subdir='')

dataset_dir = os.path.join(os.path.dirname(dataset), 'aclImdb')
os.listdir(dataset_dir)
```

```
Downloading data from https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.t
84125825/84125825 [=====] - 3s 0us/step
['test', 'imdb.vocab', 'README', 'train', 'imdbEr.txt']
```

Take a look at the `train/` directory. It has `pos` and `neg` folders with movie reviews labelled as positive and negative respectively. You will use reviews from `pos` and `neg` folders to train a binary classification model.

```
train_dir = os.path.join(dataset_dir, 'train')
os.listdir(train_dir)
```

```
['pos',
 'neg',
 'unsupBow.feats',
 'urls_pos.txt',
 'unsup',
 'urls_neg.txt',
 'urls_unsup.txt',
 'labeledBow.feats']
```

The `train` directory also has additional folders which should be removed before creating training dataset.

```
remove_dir = os.path.join(train_dir, 'unsup')
shutil.rmtree(remove_dir)
```

Next, create a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) using `tf.keras.utils.text_dataset_from_directory` (https://www.tensorflow.org/api_docs/python/tf/keras/utils/text_dataset_from_directory). You can read more about using this utility in this [text classification tutorial](https://www.tensorflow.org/tutorials/keras/text_classification) (https://www.tensorflow.org/tutorials/keras/text_classification).

Use the `train` directory to create both train and validation datasets with a split of 20% for validation.

```
batch_size = 1024
seed = 123
train_ds = tf.keras.utils.text_dataset_from_directory(
```

```

    'aclImdb/train', batch_size=batch_size, validation_split=0.2,
    subset='training', seed=seed)
val_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train', batch_size=batch_size, validation_split=0.2,
    subset='validation', seed=seed)

```

```

Found 25000 files belonging to 2 classes.
Using 20000 files for training.
Found 25000 files belonging to 2 classes.
Using 5000 files for validation.

```

Take a look at a few movie reviews and their labels (1: positive, 0: negative) from the train dataset.

```

for text_batch, label_batch in train_ds.take(1):
    for i in range(5):
        print(label_batch[i].numpy(), text_batch.numpy()[i])

```

```

0 b"0h My God! Please, for the love of all that is holy, Do Not Watch This Movie
1 b'This movie is S0000 funny!!! The acting is WONDERFUL, the Ramones are sexy,
0 b"Alex D. Linz replaces Macaulay Culkin as the central figure in the third mov
0 b"There's a good movie lurking here, but this isn't it. The basic idea is good
0 b'I saw this movie at an actual movie theater (probably the \\\(2.00 one) with

```

Configure the dataset for performance

These are two important methods you should use when loading data to make sure that I/O does not become blocking.

`.cache()` keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.

`.prefetch()` overlaps data preprocessing and model execution while training.

You can learn more about both methods, as well as how to cache data to disk in the [data performance guide](https://www.tensorflow.org/guide/data_performance) (https://www.tensorflow.org/guide/data_performance).

```
AUTOTUNE = tf.data.AUTOTUNE
```

```
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)  
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Using the Embedding layer

Keras makes it easy to use word embeddings. Take a look at the [Embedding](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) layer.

The Embedding layer can be understood as a lookup table that maps from integer indices (which stand for specific words) to dense vectors (their embeddings). The dimensionality (or width) of the embedding is a parameter you can experiment with to see what works well for your problem, much in the same way you would experiment with the number of neurons in a Dense layer.

```
# Embed a 1,000 word vocabulary into 5 dimensions.  
embedding_layer = tf.keras.layers.Embedding(1000, 5)
```

When you create an Embedding layer, the weights for the embedding are randomly initialized (just like any other layer). During training, they are gradually adjusted via backpropagation. Once trained, the learned word embeddings will roughly encode similarities between words (as they were learned for the specific problem your model is trained on).

If you pass an integer to an embedding layer, the result replaces each integer with the vector from the embedding table:

```
result = embedding_layer(tf.constant([1, 2, 3]))  
result.numpy()
```

```
array([[ 0.03135875,  0.03640932, -0.00031054,  0.04873694, -0.03376802],
       [ 0.00243857, -0.02919209, -0.01841091, -0.03684188,  0.02765827],
       [-0.01245669, -0.01057661, -0.04422194, -0.0317696 , -0.00031216]],
      dtype=float32)
```

For text or sequence problems, the Embedding layer takes a 2D tensor of integers, of shape `(samples, sequence_length)`, where each entry is a sequence of integers. It can embed sequences of variable lengths. You could feed into the embedding layer above batches with shapes `(32, 10)` (batch of 32 sequences of length 10) or `(64, 15)` (batch of 64 sequences of length 15).

The returned tensor has one more axis than the input, the embedding vectors are aligned along the new last axis. Pass it a `(2, 3)` input batch and the output is `(2, 3, N)`

```
result = embedding_layer(tf.constant([[0, 1, 2], [3, 4, 5]]))
result.shape
```

```
TensorShape([2, 3, 5])
```

When given a batch of sequences as input, an embedding layer returns a 3D floating point tensor, of shape `(samples, sequence_length, embedding_dimensionality)`. To convert from this sequence of variable length to a fixed representation there are a variety of standard approaches. You could use an RNN, Attention, or pooling layer before passing it to a Dense layer. This tutorial uses pooling because it's the simplest. The [Text Classification with an RNN](https://www.tensorflow.org/text/tutorials/text_classification_rnn) (https://www.tensorflow.org/text/tutorials/text_classification_rnn) tutorial is a good next step.

Text preprocessing

Next, define the dataset preprocessing steps required for your sentiment classification model. Initialize a `TextVectorization` layer with the desired parameters to vectorize movie reviews. You

can learn more about using this layer in the [Text Classification](https://www.tensorflow.org/tutorials/keras/text_classification) (https://www.tensorflow.org/tutorials/keras/text_classification) tutorial.

```
# Create a custom standardization function to strip HTML break tags '<br />'.
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
    return tf.strings.regex_replace(stripped_html,
                                    ' [%s]' % re.escape(string.punctuation), '')

# Vocabulary size and number of words in a sequence.
vocab_size = 10000
sequence_length = 100

# Use the text vectorization layer to normalize, split, and map strings to
# integers. Note that the layer uses the custom standardization defined above.
# Set maximum_sequence length as all samples are not of the same length.
vectorize_layer = TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)

# Make a text-only dataset (no labels) and call adapt to build the vocabulary.
text_ds = train_ds.map(lambda x, y: x)
vectorize_layer.adapt(text_ds)
```

WARNING:tensorflow:From /tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/core/framework/lambda_fuctions.py:100: tf.nn.conv2d is deprecated and will be removed in a future version. Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they a

Create a classification model

Use the [Keras Sequential API](https://www.tensorflow.org/guide/keras/sequential_model) (https://www.tensorflow.org/guide/keras/sequential_model) to define the sentiment classification model. In this case it is a "Continuous bag of words" style model.

- The **TextVectorization** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/TextVectorization) layer transforms strings into vocabulary indices. You have already initialized `vectorize_layer` as a `TextVectorization` layer and built its vocabulary by calling `adapt` on `text_ds`. Now `vectorize_layer` can be used as the first layer of your end-to-end classification model, feeding transformed strings into the Embedding layer.
- The **Embedding** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) layer takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. These vectors are learned as the model trains. The vectors add a dimension to the output array. The resulting dimensions are: (batch, sequence, embedding).
- The **GlobalAveragePooling1D** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling1D) layer returns a fixed-length output vector for each example by averaging over the sequence dimension. This allows the model to handle input of variable length, in the simplest way possible.
- The fixed-length output vector is piped through a fully-connected (**Dense** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) layer with 16 hidden units.
- The last layer is densely connected with a single output node.

Caution: This model doesn't use masking, so the zero-padding is used as part of the input and hence the padding length may affect the output. To fix this, see the [masking and padding guide](https://www.tensorflow.org/guide/keras/masking_and_padding) (https://www.tensorflow.org/guide/keras/masking_and_padding).

```
embedding_dim=16
```

```
model = Sequential([
    vectorize_layer,
    Embedding(vocab_size, embedding_dim, name="embedding"),
    GlobalAveragePooling1D(),
    Dense(16, activation='relu'),
    Dense(1)
])
```

Compile and train the model

You will use [TensorBoard](https://www.tensorflow.org/tensorboard) (<https://www.tensorflow.org/tensorboard>) to visualize metrics including loss and accuracy. Create a `tf.keras.callbacks.TensorBoard` (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/TensorBoard).

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs")
```

Compile and train the model using the Adam optimizer and BinaryCrossentropy loss.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=15,
    callbacks=[tensorboard_callback])
```

```
Epoch 1/15
20/20 [=====] - 7s 206ms/step - loss: 0.6920 - accurac
Epoch 2/15
20/20 [=====] - 1s 50ms/step - loss: 0.6879 - accuracy
Epoch 3/15
20/20 [=====] - 1s 48ms/step - loss: 0.6815 - accuracy
Epoch 4/15
20/20 [=====] - 1s 51ms/step - loss: 0.6713 - accuracy
Epoch 5/15
20/20 [=====] - 1s 49ms/step - loss: 0.6566 - accuracy
Epoch 6/15
20/20 [=====] - 1s 49ms/step - loss: 0.6377 - accuracy
Epoch 7/15
```

With this approach the model reaches a validation accuracy of around 78% (note that the model is overfitting since training accuracy is higher).

Note: Your results may be a bit different, depending on how weights were randomly initialized before training the embedding layer.

You can look into the model summary to learn more about each layer of the model.

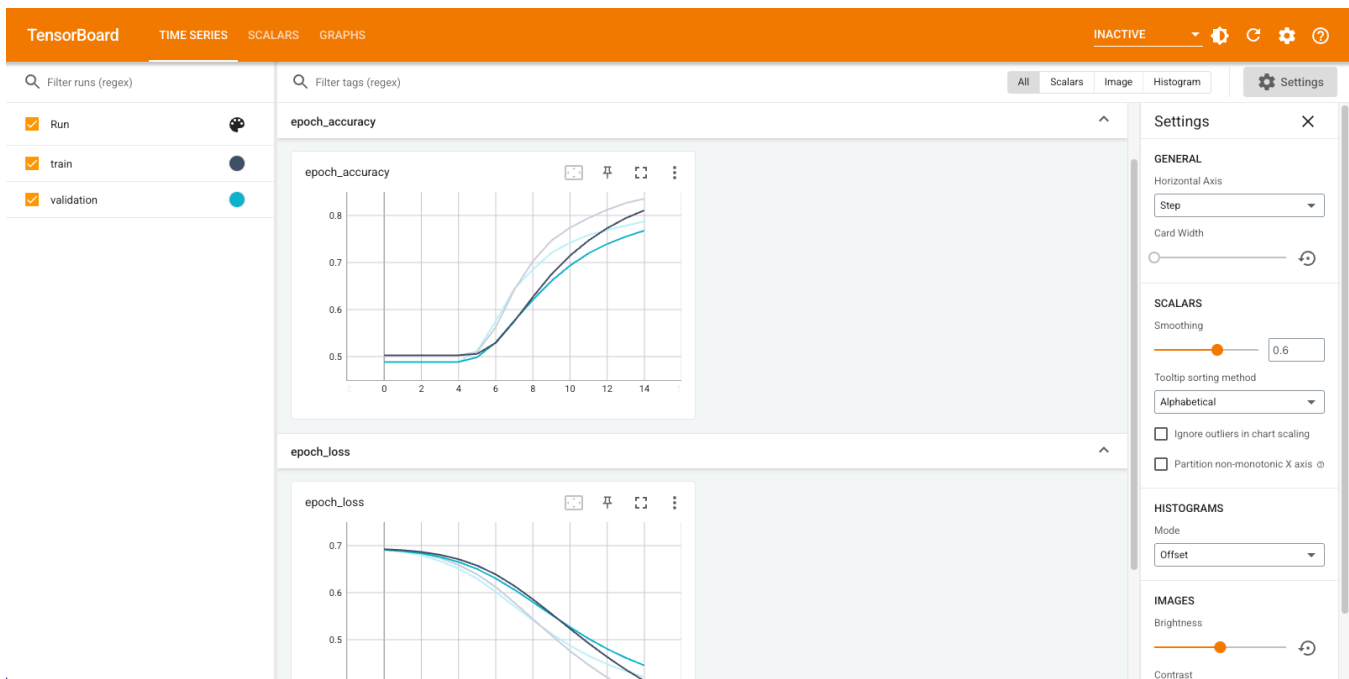
```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(None, 100)	0
embedding (Embedding)	(None, 100, 16)	160000
global_average_pooling1d (GlobalAveragePooling1D)	(None, 16)	0
dense (Dense)	(None, 16)	272

Visualize the model metrics in TensorBoard.

```
#docs_infra: no_execute
%load_ext tensorboard
%tensorboard --logdir logs
```



Retrieve the trained word embeddings and save them to disk

Next, retrieve the word embeddings learned during training. The embeddings are weights of the Embedding layer in the model. The weights matrix is of shape (vocab_size, embedding_dimension).

Obtain the weights from the model using `get_layer()` and `get_weights()`. The `get_vocabulary()` function provides the vocabulary to build a metadata file with one token per line.

```
weights = model.get_layer('embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()
```

Write the weights to disk. To use the [Embedding Projector](http://projector.tensorflow.org) (<http://projector.tensorflow.org>), you will upload two files in tab separated format: a file of vectors (containing the embedding), and a file of meta data (containing the words).

```
out_v = io.open('vectors.tsv', 'w', encoding='utf-8')
out_m = io.open('metadata.tsv', 'w', encoding='utf-8')
```

```
for index, word in enumerate(vocab):
    if index == 0:
        continue # skip 0, it's padding.
    vec = weights[index]
    out_v.write('\t'.join([str(x) for x in vec]) + "\n")
    out_m.write(word + "\n")
out_v.close()
out_m.close()
```

If you are running this tutorial in [Colaboratory](https://colab.research.google.com) (<https://colab.research.google.com>), you can use the following snippet to download these files to your local machine (or use the file browser, *View -> Table of contents -> File browser*).

```
try:
    from google.colab import files
    files.download('vectors.tsv')
    files.download('metadata.tsv')
except Exception:
    pass
```

Visualize the embeddings

To visualize the embeddings, upload them to the embedding projector.

Open the [Embedding Projector](http://projector.tensorflow.org/) (<http://projector.tensorflow.org/>) (this can also run in a local TensorBoard instance).

- Click on "Load data".
- Upload the two files you created above: `vecs.tsv` and `meta.tsv`.

The embeddings you have trained will now be displayed. You can search for words to find their closest neighbors. For example, try searching for "beautiful". You may see neighbors like "wonderful".

Note: Experimentally, you may be able to produce more interpretable embeddings by using a simpler model.

Try deleting the **Dense(16)** layer, retraining the model, and visualizing the embeddings again.

Note: Typically, a much larger dataset is needed to train more interpretable word embeddings. This tutorial uses a small IMDb dataset for the purpose of demonstration.

Next Steps

This tutorial has shown you how to train and visualize word embeddings from scratch on a small dataset.

- To train word embeddings using Word2Vec algorithm, try the [Word2Vec](https://www.tensorflow.org/tutorials/text/word2vec) (<https://www.tensorflow.org/tutorials/text/word2vec>) tutorial.
- To learn more about advanced text processing, read the [Transformer model for language understanding](https://www.tensorflow.org/text/tutorials/transformer) (<https://www.tensorflow.org/text/tutorials/transformer>).

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 4.0 License](https://creativecommons.org/licenses/by/4.0/) (<https://creativecommons.org/licenses/by/4.0/>), and code samples are licensed under the [Apache 2.0 License](https://www.apache.org/licenses/LICENSE-2.0) (<https://www.apache.org/licenses/LICENSE-2.0>). For details, see the [Google Developers Site Policies](https://developers.google.com/site-policies) (<https://developers.google.com/site-policies>). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2023-05-27 UTC.