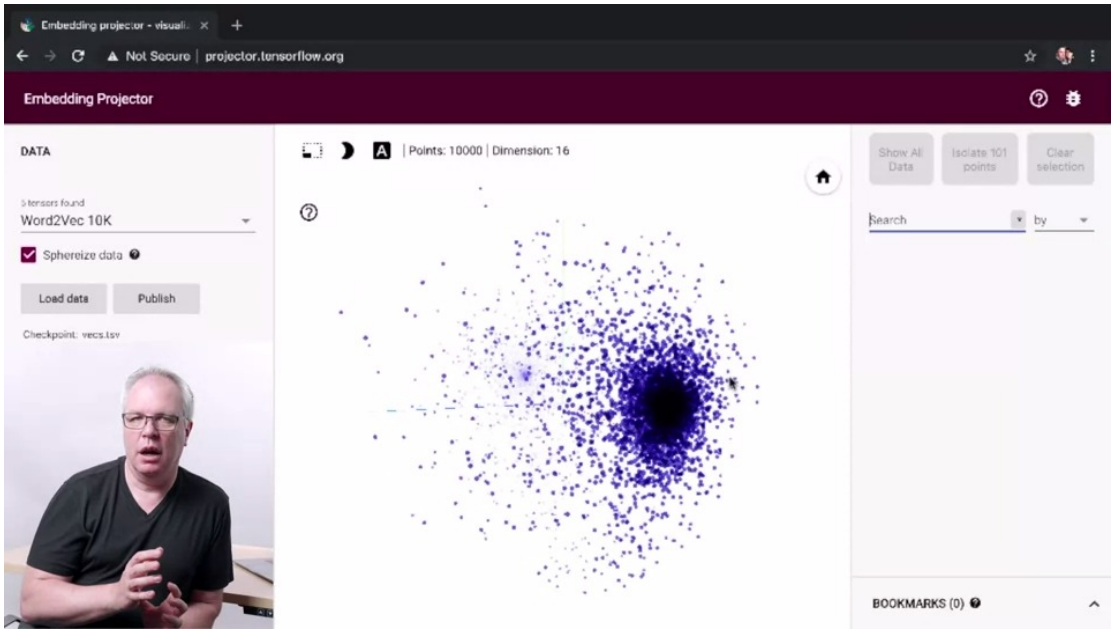
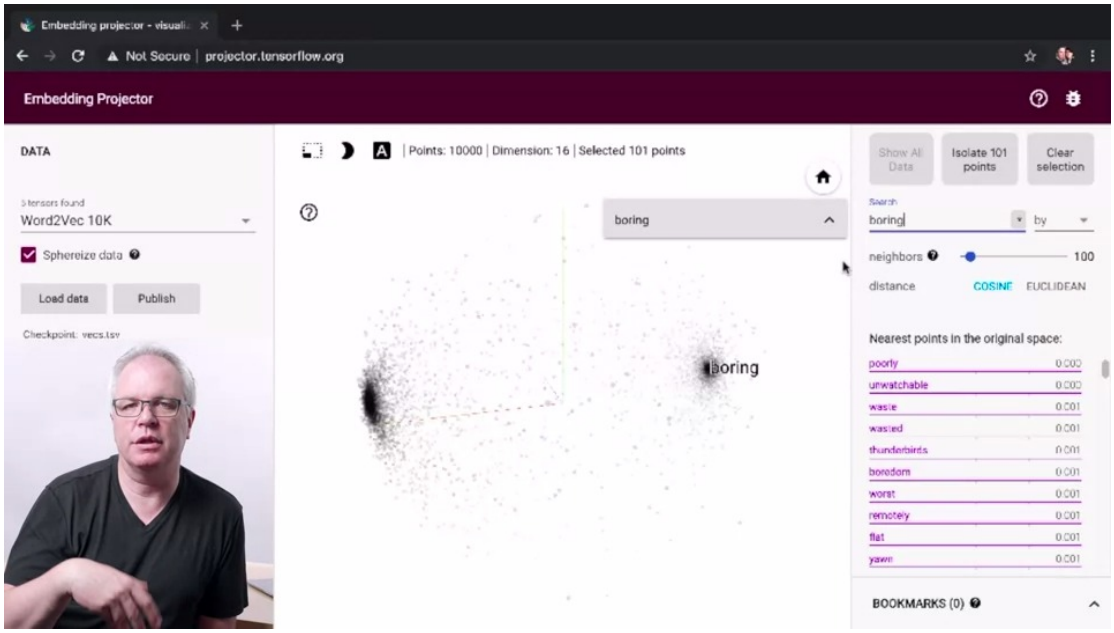
2.1 Introduction

Last week, you looked at tokenizing text. Where turn text into sequences of numbers with a number was the value of a key value pair with the key being the word. So for example, you could represent the word TensorFlow with the value nine, and then replace every instance of the word with a nine in a sequence. Using tools and TensorFlow, you are able to process strings to get indices of all the words in a corpus of strings and then convert the strings into matrices of numbers. This is the start of getting sentiment out of your sentences.
But right now, it's still just a string of numbers representing words. So from there, how would one actually get sentiment? Well, that's something that can be learned from a corpus of words in much the same way as features were extracted from images.



This process is called embedding, with the idea being that **words and associated words are clustered as vectors in a multi-dimensional space.** Here, I'm showing an embedding projector with classifications of movie reviews. This week, you'll learn how to build that. The reviews are in two main categories; positive and negative. So together with the labels, TensorFlow was able to build these embeddings showing a clear clustering of words that are distinct to both of these review types. I can actually search for words to see which ones match a classification.



So for example, if I search for boring, we can see that it lights up in one of the clusters and that associated words were clearly negative such as unwatchable. Similarly, if I search for a negative word like annoying, I'll find it along with annoyingly in the cluster that's clearly the negative reviews. Or if I search for fun, I'll find that fun and funny are positive, fundamental is neutral, and unfunny is of course, negative. This week, you'll learn how to use embeddings and how to build a classifier that gave that visualization. You're most of the way there already with the work that you've been doing with string tokenization. We'll get back to that later but first let's look at building the IMDB classification that you just visualized.

2.2 The IMBD dataset

Part of the vision of TensorFlow to make machine learning and deep learning easier to learn and easier to use, is the concept of **having built-in data sets**. You seen the little bit of a preview of the way back in the first course, when the fashion MNEs was available to you without you needing to download and split the data into training a test sets.

Expanding on this, there's a library called TensorFlow Data Services or TFTS for short, and that contains many data sets and lots of different categories. Here's some examples; and while we can see that there are many different data sets for different types, particularly image-based, there's also a few for text, and we'll be using the IMDB reviews dataset next. This dataset is ideal because it contains a large body of texts, 50,000 movie reviews which are categorized as positive or negative. It was authored by Andrew Mass et al at Stanford, and you can learn more about it at this link.

```
audio

"nsynth"

image

"abstract_reasoning"
"caltech101"
"cats_vs_dogs"
"celeb_a"
"celeb_a_hq"
"cifar10"
"cifar100"
"cifar10_corrupted"
"coco2014"
"colorectal_histology"
"cycle_gan"
"diabetic_retinopathy..."
"dsprites"
"dtd"
"emnist"
"fashion_mnist"
"horses_or_humans"
"image_label_folder"
"imagenet2012"
"imagenet2012_corrupted"
"kmnist"
"lsun"
"mnist"
"omniglot"
"open_images_v4"
"oxford_iiit_pet"
"quickdraw_bitmap"
"rock_paper_scissors"
"shapes3d"
"smallnorb"
"sun397"
"svhn_cropped"
"tf_flowers"

structured

"higgs"
"iris"
"titanic"

text

"cnn_dailymail"
"glue"
"imdb_reviews"
"lm1b"
"multi_nli"
"squad"
"wikipedia"
"xnli"

translate

"flores"
"para_crawl"
"ted_hrlr_translate"
"ted_multi_translate"
"wmt15_translate"
"wmt16_translate"
"wmt17_translate"
"wmt18_translate"
"wmt19_translate"
```

http://ai.stanford.edu/~amaas/data/sentiment/

```
@InProceedings{maas-EtAl:2011:ACL-HLT2011,
  author    = {Maas, Andrew L.  and  Daly, Raymond E.  and  Pham, Peter T.  and  Huang, Dan  and  Ng,
Andrew Y.  and  Potts, Christopher},
  title     = {Learning Word Vectors for Sentiment Analysis},
  booktitle = {Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics:
Human Language Technologies},
  month     = {June},
  year      = {2011},
  address   = {Portland, Oregon, USA},
  publisher = {Association for Computational Linguistics},
  pages     = {142--150},
  url       = {http://www.aclweb.org/anthology/P11-1015}
}
```

Please find the link to the IMDB reviews dataset here
You will find here 50,000 movie reviews which are classified as positive of negative.

2.3 Looking into the details

So let's start looking at it. There are a couple of things that you need to take into account before you start working with this week's code in TensorFlow. The first is the version of TensorFlow you're using. Use this code to determine it. Also, do note that all the code I'm using here is in Python 3. There are some differences if you use Python 2. So if you're using a Colab, you can set the environment to three. If you're doing this in your own environment, you may need to make some changes.

```
tf.enable_eager_execution()
```

If the previous code gave you TensorFlow 1.x, you'll need this line of code before you can go any further. If it gave you 2.x, then you won't need anything because eager execution is enabled by default in TensorFlow 2.0.

```
!pip install -q tensorflow-datasets
```

If you're using Google Colab, then you should have TensorFlow datasets already installed. Should you not have them, they're easily installed with this line of code.

```
import tensorflow_datasets as tfds
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)
```

Now, you can import TensorFlow datasets, and in this case I call them tfds. With imdb reviews, I can now call tfds.load, pass it the string imdb reviews, and it will return the data from imdb, and metadata about it with this code.

```
import numpy as np

train_data, test_data = imdb['train'], imdb['test']
```

The data is split into 25,000 samples for training and 25,000 samples for testing. I can split them out like this.

```
training_sentences = []
training_labels = []

testing_sentences = []
testing_labels = []

# str(s.tonumpy()) is needed in Python3 instead of just s.numpy()
for s,l in train_data:
  training_sentences.append(str(s.numpy()))
  training_labels.append(l.numpy())

for s,l in test_data:
  testing_sentences.append(str(s.numpy()))
  testing_labels.append(l.numpy())
```


Each of these are iterables containing the 25,000 respective sentences and labels as tensors. Up to this point, we've been using the Cara's tokenizers and padding tools on arrays of sentences, so we need to do a little converting. We'll do it like this. First of all, let's define the lists containing the sentences and labels for both training and testing data. Now, I can iterate over training data extracting the sentences and the labels. **The values for S and I are tensors, so by calling their NumPy method, I'll actually extract their value.** Then I'll do the same for the test set. Here's an example of a review.

```
tf.Tensor(b"As a lifelong fan of Dickens, I have invariably been disappointed
by adaptations of his novels.<br /><br />Although his works presented an
extremely accurate re-telling of human life at every level in Victorian Britain,
throughout them all was a pervasive thread of humour that could be both playful
or sarcastic as the narrative dictated. In a way, he was a literary
caricaturist and cartoonist. He could be serious and hilarious in the same
sentence. He pricked pride, lampooned arrogance, celebrated modesty,
and empathised with loneliness and poverty. It may be a clich\xc3\xa9, but
he was a people's writer.<br /><br />And it is the comedy that is so often
missing from his interpretations. At the time of writing, Oliver Twist
is being dramatised in serial form on BBC television. All of the misery
and cruelty is their, but non of the humour, irony, and savage lampoonery.",
shape=(), dtype=string)
```

I've truncated it to fit it on this slide, but you can see how it is stored as a tf.tensor. Similarly, here's a bunch of labels also stored as tensors.

```
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
```

The value 1 indicates a positive review and zero a negative one.

```
training_labels_final = np.array(training_labels)
testing_labels_final = np.array(testing_labels)
```

When training, my labels are expected to be NumPy arrays. So I'll turn the list of labels that I've just created into NumPy arrays with this code.

```
vocab_size = 10000
embedding_dim = 16
max_length = 120
trunc_type='post'
oov_tok = "<OOV>"

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences(training_sentences)
padded = pad_sequences(sequences,maxlen=max_length, truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences,maxlen=max_length)
```


Next up, we'll tokenize our sentences. Here's the code. I've put the hyperparameters at the top like this for the reason that it makes it easier to change and edit them, instead of phishing through function sequences for the literals and then changing those. Now, as before, we import the tokenizer and the pad sequences. We'll create an instance of tokenizer, giving it our vocab size and our desired out of vocabulary token. We'll now fit the tokenizer on our training

set of data. Once we have our word index, we can now replace the strings containing the words with the token value we created for them. This will be the list called sequences. As before, the sentences will have variant length. So we'll pad and or truncate the sequenced sentences until they're all the same length, determined by the maxlength parameter. Then we'll do the same for the testing sequences. Note that the word index is words that are derived from the training set, so you should expect to see a lot more out of vocabulary tokens in the test exam.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

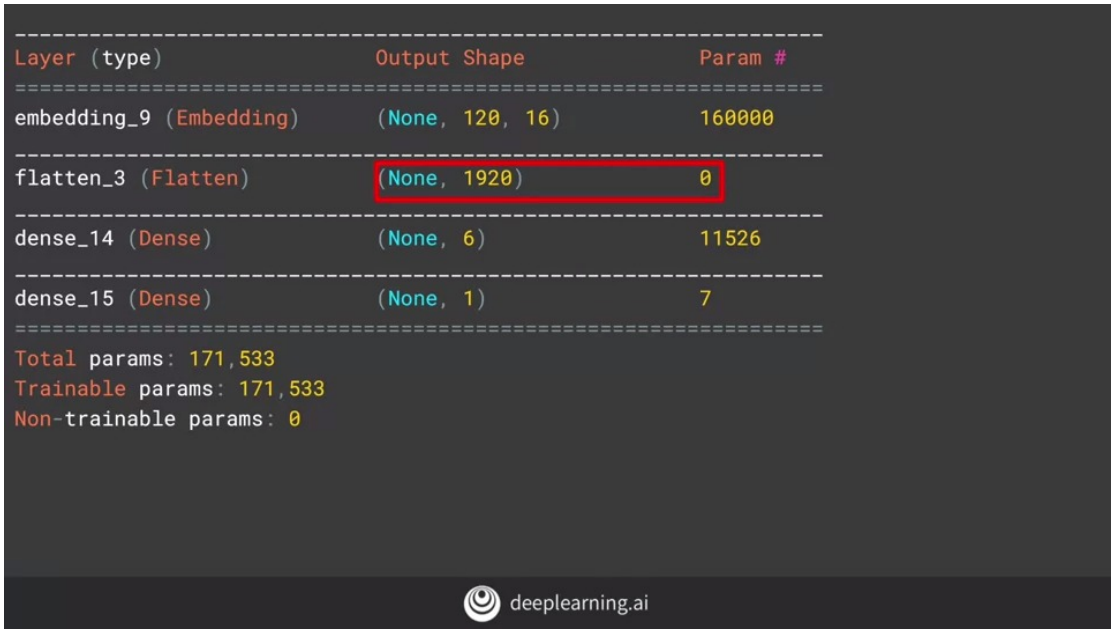
Now it's time to define our neural network. This should look very familiar by now, except for maybe this line, the embedding. This is the key to text sentiment analysis in TensorFlow, and this is where the magic really happens.

2.4 How can we use vectors?

The full scope of how embeddings work is beyond the scope of this course. But think of it like this. You have words in a sentence and often words that have similar meanings are close to each other. So in a movie review, it might say that the movie was dull and boring, or it might say that it was fun and exciting. So what if you could pick a vector in a higher-dimensional space say 16 dimensions, and words that are found together are given similar vectors. Then over time, words can begin to cluster together. The meaning of the words can come from the labeling of the dataset. So in this case, we say a negative review and the words dull and boring show up a lot in the negative review so that they have similar sentiments, and they are close to each other in the sentence. Thus their vectors will be similar. As the neural network trains, it can then learn these vectors associating them with the labels to come up with what's called an embedding i.e., the vectors for each word with their associated sentiment.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The results of the embedding will be a 2D array with the length of the sentence and the embedding dimension for example 16 as its size. So we need to flatten it out in much the same way as we needed to flatten out our images. We then feed that into a dense neural network to do the classification. Often in natural language processing, a different layer type than a flatten is used, and this is a global average pooling 1D. The reason for this is the size of the output vector being fed into the dance.



So for example, if I show the summary of the model with the flatten that we just saw, it will look like this. Or alternatively, you can use a Global Average Pooling 1D like this, which averages across the vector to flatten it out. Your model summary should look like this, which is simpler and should be a little faster.

Try it for yourself in colab and check the results. Over 10 epochs with global average pooling, I got an accuracy of 0.9664 on training and 0.8187 on test, taking about 6.2 seconds per epoch. With flatten, my accuracy was 1.0 and my validation about 0.83 taking about 6.5 seconds per epoch. So it was a little slower, but a bit more accurate.

2.5 More into the details

```
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

You can compile your model as before, and print out the summary with this code.

```
num_epochs = 10
model.fit(padded,
          training_labels_final,
          epochs=num_epochs,
          validation_data=(testing_padded, testing_labels_final))
```

Now training is the simplest passing padded and your training labels final as your training set, specifying the number of epochs, and passing the testing padded and testing labels final as your test set.

```
Epoch 8/10
25000/25000 [==============================] -
6s 256us/sample - loss: 5.2086e-04 - acc: 1.0000 - val_loss: 0.7252 - val_acc: 0.8270
Epoch 9/10
25000/25000 [==============================] -
6s 222us/sample - loss: 3.0199e-04 - acc: 1.0000 - val_loss: 0.7628 - val_acc: 0.8269
Epoch 10/10
25000/25000 [==============================] -
6s 224us/sample - loss: 1.7872e-04 - acc: 1.0000 - val_loss: 0.7997 - val_acc: 0.8259
```

Here's the results of training, with the training set giving us 1.00 accuracy and the validation set at 0.8259. So there's a good chance that we're overfitting. We'll look at some strategies to avoid this later, but you should expect results a little bit like this.

```
e = model.layers[0]
weights = e.get_weights()[0]
print(weights.shape) # shape: (vocab_size, embedding_dim)


(10000, 16)
```

Okay. Now we need to talk about and demonstrate the embeddings, so you can visualize them like you did right back at the beginning of this lesson. We'll start by getting the results of the embeddings layer, which is layer zero. We can get the weights, and print out their shape like this. **We can see that this is a 10,000 by 16 array, we have 10,000 words in our corpus, and we're working in a 16 dimensional array, so our embedding will have that shape.** To be able to plot it, we need a helper function to reverse our word index.

```
Hello : 1
World : 2
How   : 3
Are   : 4
You   : 5


reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])


1 : Hello
2 : World
3 : How
4 : Are
5 : You
```


As it currently stands, our word index has the key being the word, and the value being the token for the word. We'll need to flip this around, to l**ook through the padded list to decode the tokens back into the words**, so we've written this helper function.

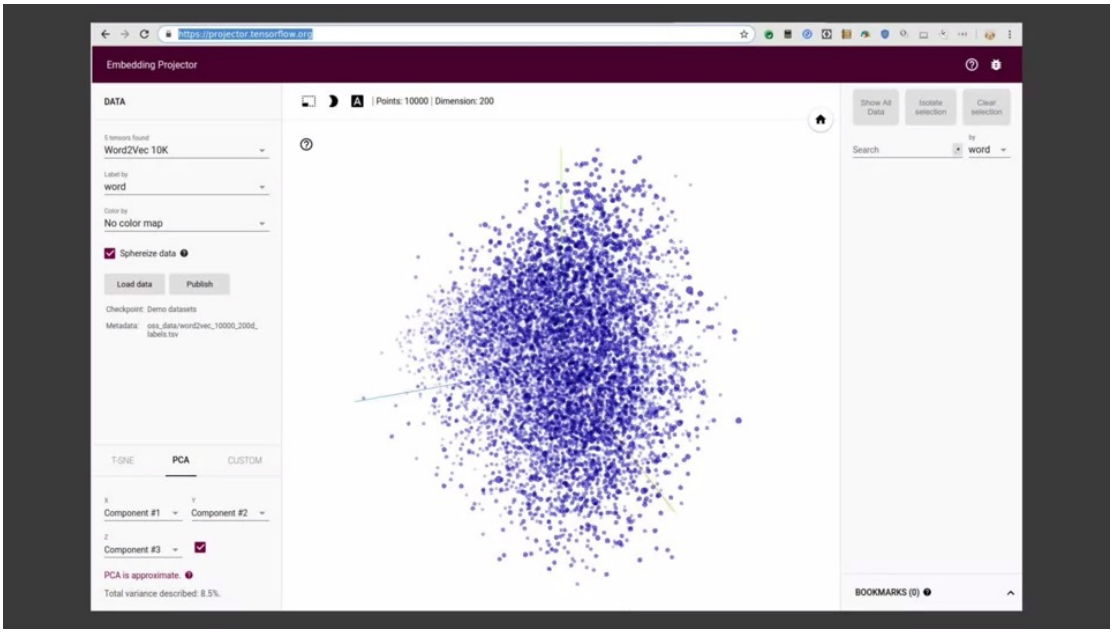
```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')
for word_num in range(1, vocab_size):
  word = reverse_word_index[word_num]
  embeddings = weights[word_num]
  out_m.write(word + "\n")
  out_v.write('\t'.join([str(x) for x in embeddings]) + "\n")
out_v.close()
out_m.close()
```

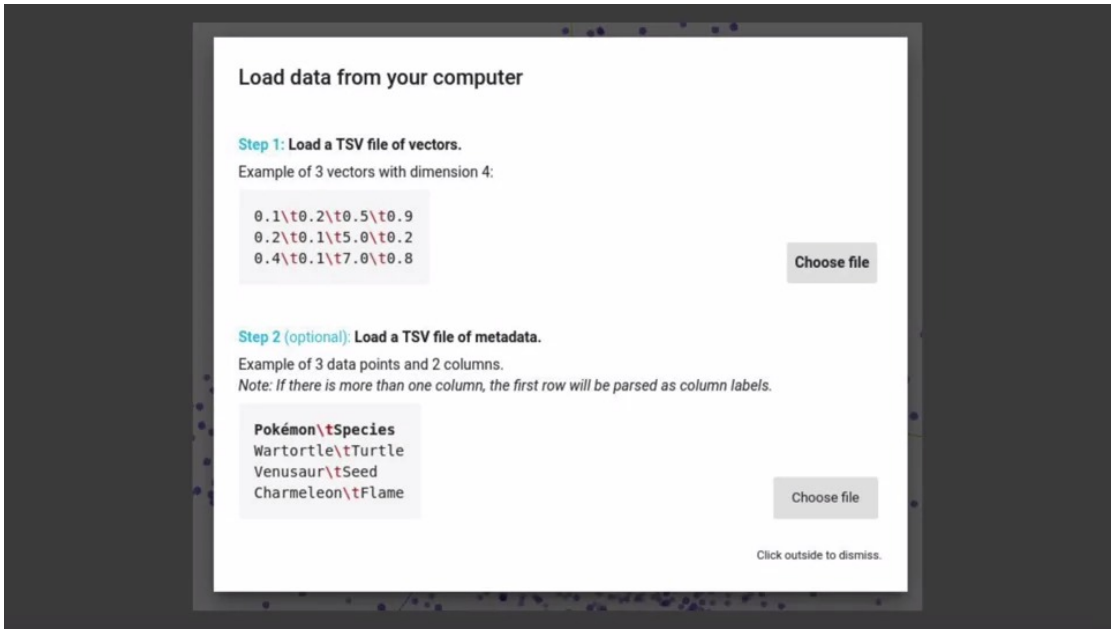
Now it's time to write the vectors and their metadata auto files. The TensorFlow Projector reads this file type and uses it to plot the vectors in 3D space so we can visualize them. To the vectors file, we simply write out the value of each of the items in the array of embeddings, i.e, the co-efficient of each dimension on the vector for this word. To the metadata array, we just write out the words.

```
try:
  from google.colab import files
except ImportError:
  pass
else:
  files.download('vecs.tsv')
  files.download('meta.tsv')
```

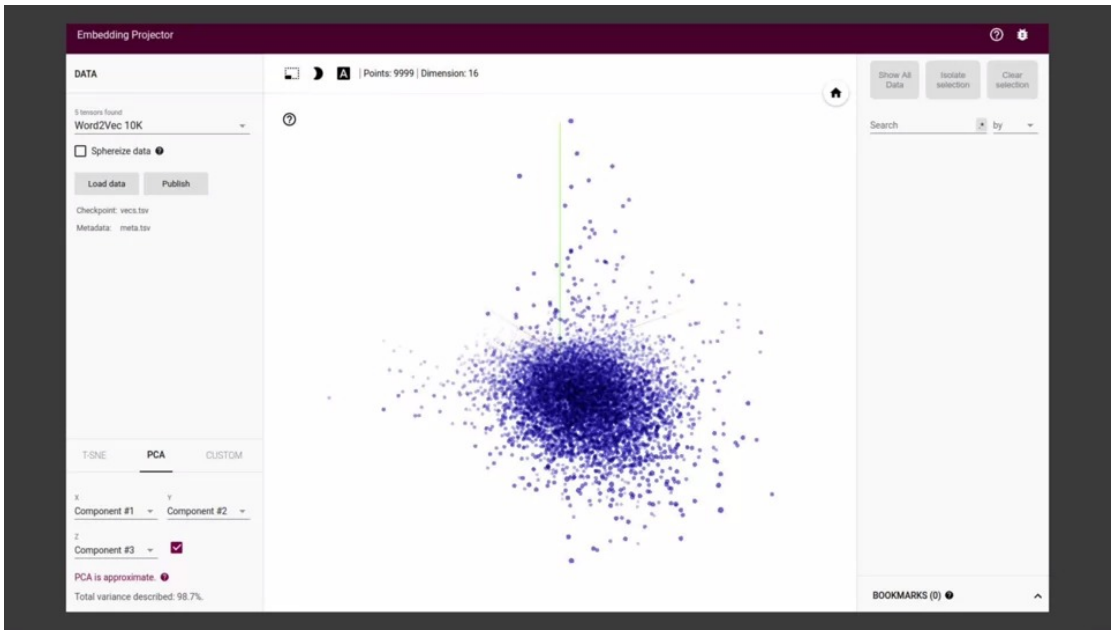
If you're working in Colab, this code will download the two files.



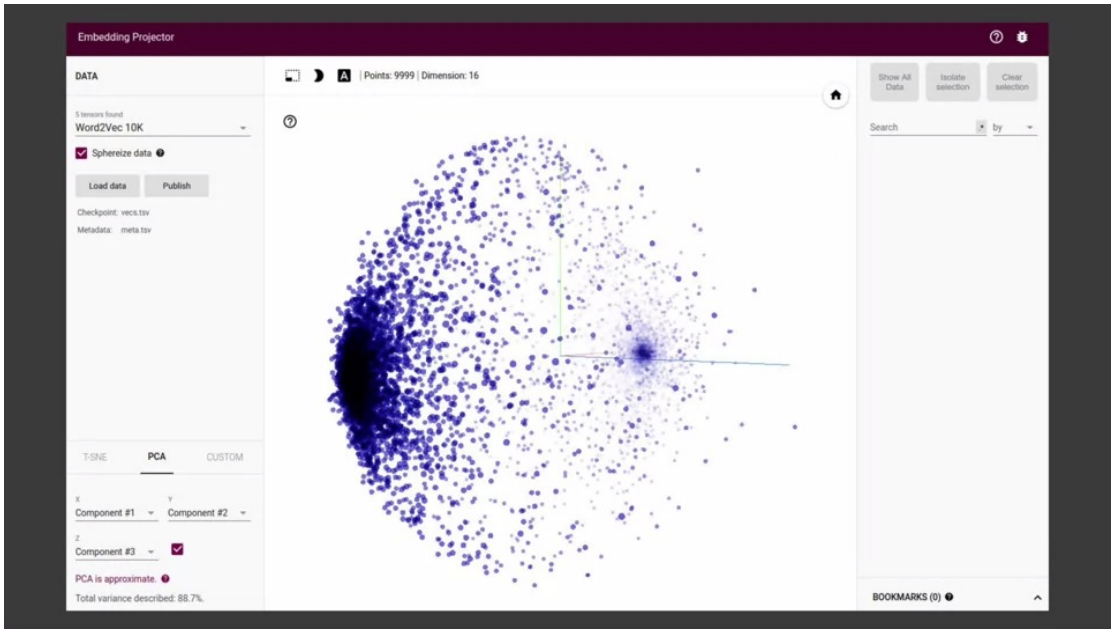
To now render the results, go to the TensorFlow Embedding Projector on projector.tensorflow.org, press the ''Load data'' button on the left.



You'll see a dialog asking you to load data from your computer. Use vector.TSV for the first one, and meta.TSV for the second.

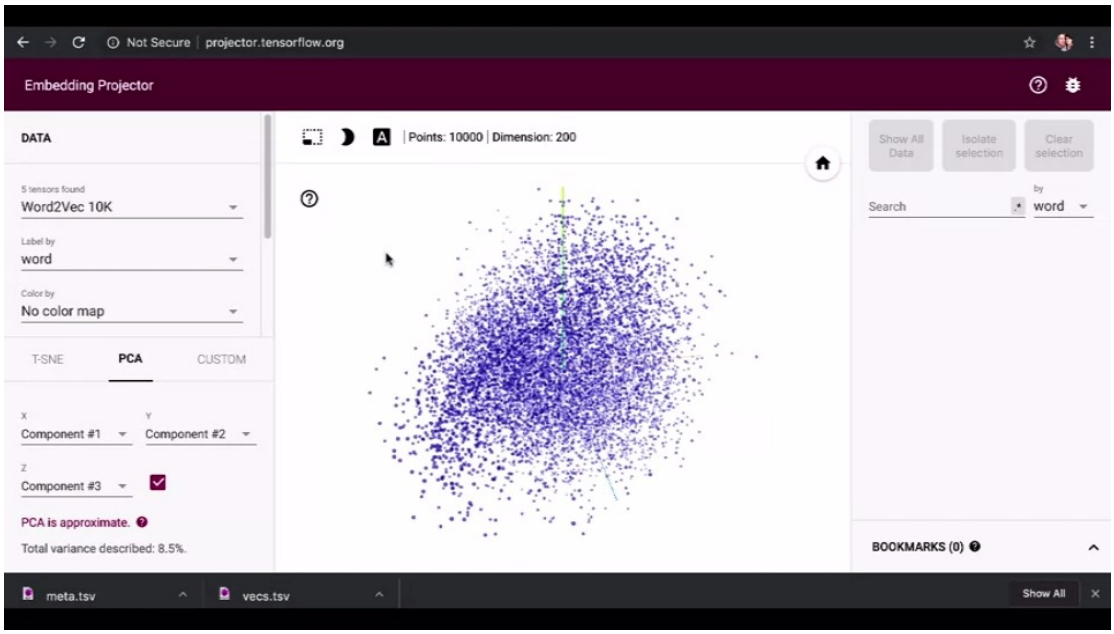


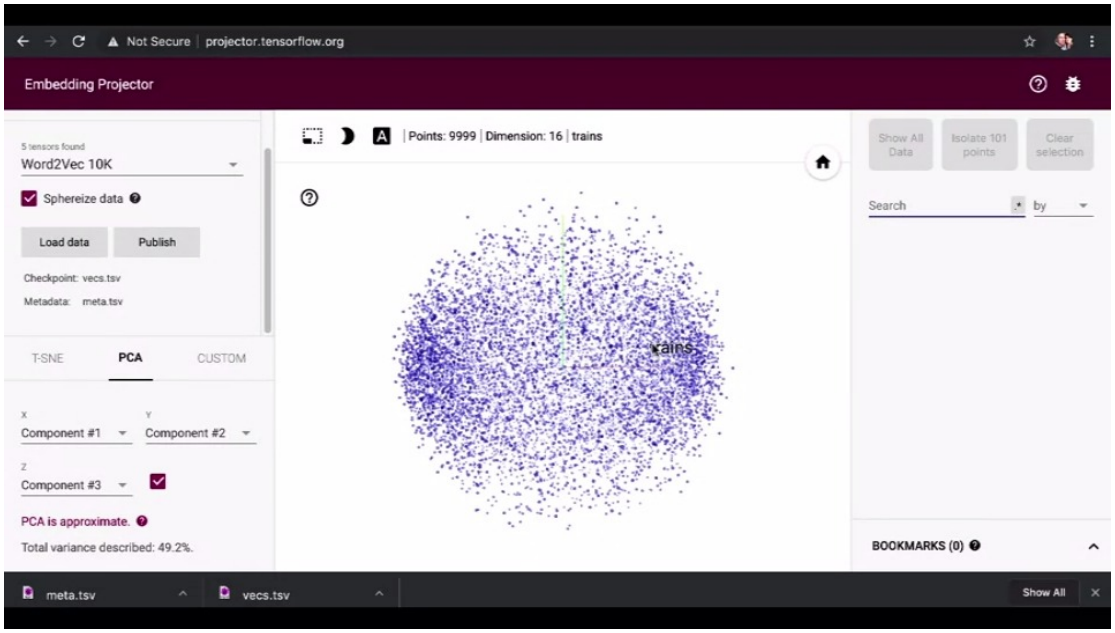
Once they're loaded, you should see something like this.

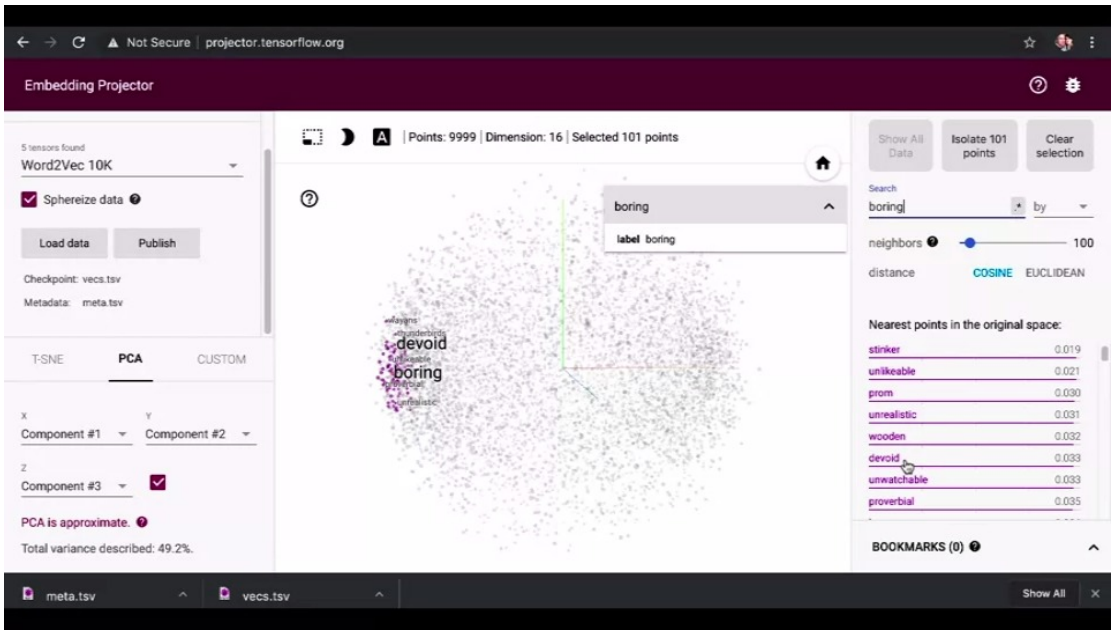


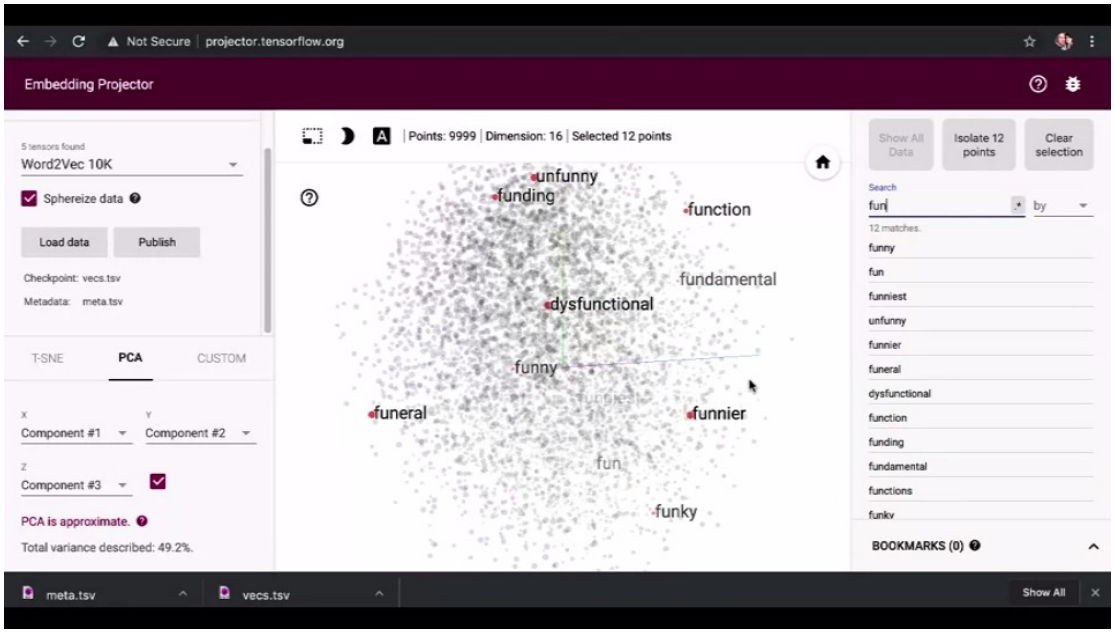
Click this ''sphereize data'' checkbox on the top left, and you'll see the binary clustering of the data. Experiment by searching for words, or clicking on the blue dots in the chart that represent words. Above all, have some fun with it. Next up, we'll step through a screencast of what you've just seen, so you can explore it in action. After that, you'll look at how TFTS has built in tokenizers that prevent you from writing a lot of the tokenizing code that we've just used.

2.6 Notebook for lesson 1




Not Secure | projector.tensorflow.org
Embedding Projector
5 tensors found
Word2Vec 10K
Sphereize data
Load data
Publish
Checkpoint: vecs.tsv
Metadata: meta.tsv
T-SNE
PCA
CUSTOM
X
Component #1
Y
Component #2
Z
Component #3
PCA is approximate.
Total variance described: 49.2%.
Points: 9999 | Dimension: 16 | trains
trains
Show All Data
Isolate 101 points
Clear selection
Search
by
BOOKMARKS (0)
meta.tsv
vecs.tsv
Show All


Not Secure | projector.tensorflow.org
Embedding Projector
5 tensors found
Word2Vec 10K
Sphereize data
Load data
Publish
Checkpoint: vecs.tsv
Metadata: meta.tsv
T-SNE
PCA
CUSTOM
X
Component #1
Y
Component #2
Z
Component #3
PCA is approximate.
Total variance described: 49.2%.
Points: 9999 | Dimension: 16 | Selected 101 points
boring
label boring
devoid
boring
Show All Data
Isolate 101 points
Clear selection
Search
boring
by
neighbors
100
distance
COSINE
EUCLIDEAN
Nearest points in the original space:
stinker 0.019
unlikeable 0.021
prom 0.030
unrealistic 0.031
wooden 0.032
devoid 0.033
unwatchable 0.033
proverbial 0.035
BOOKMARKS (0)
meta.tsv
vecs.tsv
Show All



2.7 Remember the sarcasm dataset?

So that was an example using the IMDB data set, where data is provided to you by the tfds API, which I hope you found helpful. Now I'd like to return to the sarcasm data set from last week, and let's look at building a classifier for that.

```
import json
import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

We'll start with importing tensorflow and json, as well as the tokenizer and pad sequences from pre-processing.

```
vocab_size = 10000
embedding_dim = 16
max_length = 32
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```

Now let's set up our hyper parameters; the vocabulary size, embedding dimensions, maximum length of sentences, and other stuff like the training size. This data set has about 27,000 records. So let's train on 20,000 and validate on the rest. The sarcasm data is stored at this URL, so you can download it to /tmp/sarcasm.json with this code.

```
with open("/tmp/sarcasm.json", 'r') as f:
    datastore = json.load(f)

sentences = []
labels = []

for item in datastore:
    sentences.append(item['headline'])
    labels.append(item['is_sarcastic'])
```

Now that you have the data set, you can open it and load it as an iterable with this code. You can create an array for sentences, and another for labels, and then iterate through the datastore, loading each headline as a sentence, and each is_sarcastic field, as your label.

2.8 Building a classifier for the sarcasm dataset

```
training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

To split the corpus into training and validation sets, we'll use this code. To get the training set, you take array items from zero to the training size, and to get the testing set, you can go from training size to the end of the array with code like this. To get the training and testing labels, you'll use similar codes to slice the labels array.

```
tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)

word_index = tokenizer.word_index

training_sequences = tokenizer.texts_to_sequences(training_sentences)
training_padded = pad_sequences(training_sequences, maxlen=max_length,
                                padding=padding_type, truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences, maxlen=max_length,
                               padding=padding_type, truncating=trunc_type)
```

Now that we have training and test sets of sequences and labels, it's time to sequence them. To pad those sequences, you'll do that with this code. You start with a tokenizer, passing it the number of words you want to tokenize on and the desired out of vocabulary token. Then fit that on the training set by calling fit on texts, passing it the training sentences array. Then you can use text to sequences to create the training sequence, replacing the words with their tokens. Then you can pad the training sequences to the desired length or truncate if they're too long. Next, you'll do the same but with a test set.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
```

Now, we can create our neural network in the usual way. We'll compile it with binary cross entropy, as we're classifying to different classes.

```
model.summary()



Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 32, 16)            160000
_________________________________________________________________
global_average_pooling1d_2 ( (None, 16)                0
_________________________________________________________________
dense_4 (Dense)              (None, 24)                408
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 25
=================================================================
Total params: 160,433
Trainable params: 160,433
Non-trainable params: 0
```

When we call a model's summary, we'll see that it looks like this, pretty much as we'd expect. It's pretty simple and embedding feeds into an average pooling, which then feeds our DNA.

```
num_epochs = 30

history = model.fit(training_padded, training_labels, epochs=num_epochs,
                    validation_data=(testing_padded, testing_labels), verbose=2)
```

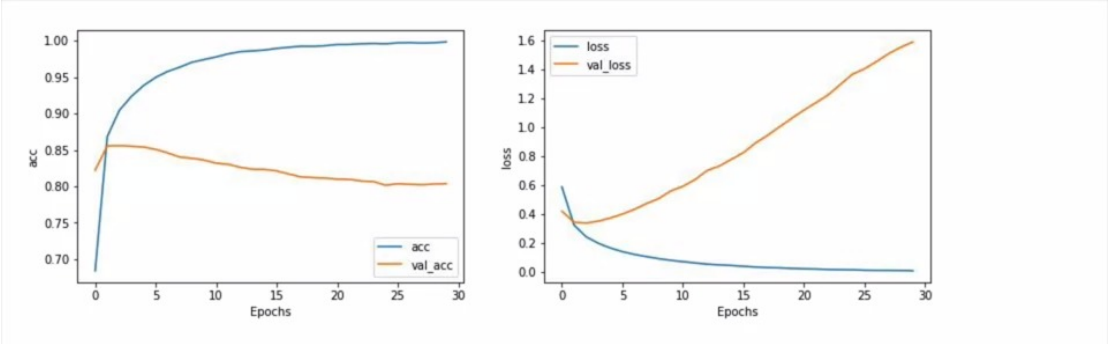
To train for 30 epochs, you pass in the padded data and labels. If you want to validate, you'll give the testing padded and labels to. After training for little while, you can plot the results.

```
import matplotlib.pyplot as plt


def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string)
  plt.legend([string, 'val_'+string])
  plt.show()

plot_graphs(history, "acc")
plot_graphs(history, "loss")
```

Here's the code for simple plot.

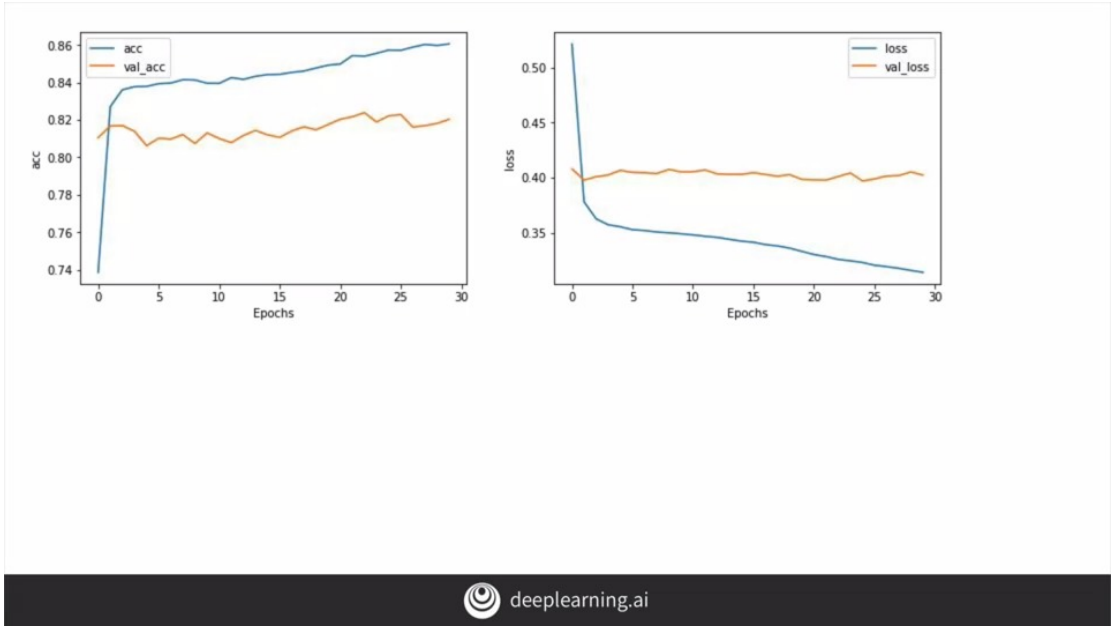


We can see accuracy increase nicely as we trained and the validation accuracy was okay, but not great. What's interesting is the loss values on the right, the training loss fall, but the validation loss increased. Well, why might that be?

2.9 Loss function

Think about loss in this context, as a confidence in the prediction. So while the number of accurate predictions increased over time, what was interesting was that the confidence per prediction effectively decreased. You may find this happening a lot with text data. So it's very important to keep an eye on it.
One way to do this is to explore the differences as you tweak the hyperparameters.

So for example, if you consider these changes, a decrease in vocabulary size, and taking shorter sentences, reducing the likelihood of padding, and then rerun, you may see results like this.



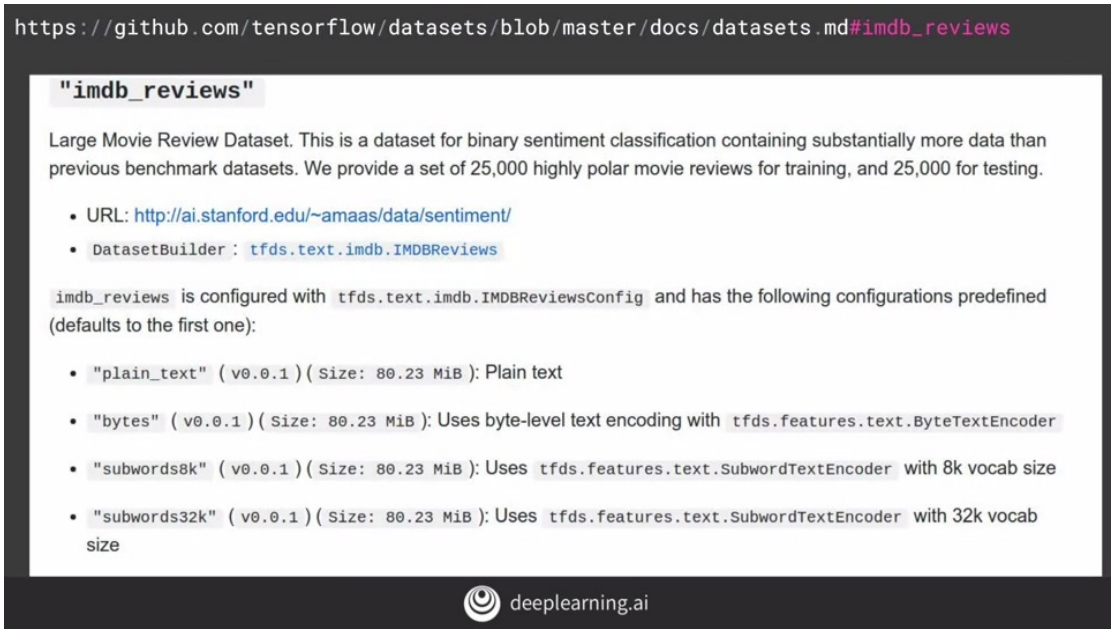
Here, you can see that the loss has flattened out which looks good, but of course, your accuracy is not as high.

Another tweak. Changing the number of dimensions using the embedding was also tried. Here, we can see that that had very little difference. Putting the hyperparameters as separate variables like this is a useful programming exercise, making it much easier for you to tweak and explore their impact on training. Keep working on them and see if you can find any combinations that give a 90 percent plus training accuracy without a cost of the lost function increasing sharply. In the next video, we'll also look at the impact of splitting our words into sub-tokens and how that might impact your training.

2.10 Pre-tokenized datasets

Earlier this week, we looked at using TensorFlow Data Services, or TFDS to load the reviews from the IMDb dataset and perform classification on them. In that video, you loaded the raw text for the reviews, and tokenized them yourself. However, often with prepackaged datasets like these, some data scientists have done the work for you already, and the IMDb dataset is no exception. In this video, we'll take a look at a version of the IMDb dataset that has been pre-tokenized for you, but the tokenization is done on sub words. We'll use that to demonstrate how text classification can have some unique issues, **namely that the sequence of words can be just as important as their existence.**

2.11 Diving into the code



So we'll start by looking at TensorFlow data-sets, you can find them at this URL. If you look at the IMDB reviews data-set, you'll see that there's a bunch of versions that you can use. These include,"plain_text" which we used in the last video,"bytes", where the text is encoded at byte level, and sub-word encoding which we'll look at in this video.

```
import tensorflow_datasets as tfds
imdb, info = tfds.load("imdb_reviews/subwords8k", with_info=True, as_supervised=True)
```

Once you're on TensorFlow 2, you can now start using the imdb subwords data-set. We'll use the 8k version today. Getting access to your training and test data is then as easy as this.

```
tokenizer = info.features['text'].encoder


tensorflow.org/datasets/api_docs/python/tfds/features/text/SubwordTextEncoder
```

Next, if you want to access the sub words tokenizer, you can do it with this code. You can learn all about the sub-words texts encoder at this URL.

```
print(tokenizer.subwords)

['the_', ', ', '. ', 'a_', 'and_', 'of_', 'to_', 's_', 'is_',
'br', 'in_', 'I_', 'that_', 'this_', 'it_', … ]
```

We have a pre-trained sub-words tokenizer now, so we can inspect its vocabulary by looking at its sub-words property.

```
sample_string = 'TensorFlow, from basics to mastery'

tokenized_string = tokenizer.encode(sample_string)
print ('Tokenized string is {}'.format(tokenized_string))

original_string = tokenizer.decode(tokenized_string)
print ('The original string: {}'.format(original_string))


Tokenized string is [6307, 2327, 4043, 2120, 2, 48, 4249, 4429,
7, 2652, 8050]

The original string: TensorFlow, from basics to mastery
```

If we want to see how it encodes or decode strings, we can do so with this code. So we can encode simply by calling the encode method passing it the string. Similarly, decode by calling the decode method. We can see the results of the tokenization when we print out the encoded and decoded strings.

```
for ts in tokenized_string:
  print ('{} ----> {}'.format(ts, tokenizer.decode([ts])))

6307 ----> Ten
2327 ----> sor
4043 ----> Fl
2120 ----> ow
2 ----> ,
48 ----> from
4249 ----> basi
4429 ----> cs
7 ----> to
2652 ----> master
8050 ----> y
```

If we want to see the tokens themselves, we can take each element and decode that, showing the value to token. Note that this is case sensitive and punctuation is maintained unlike the tokenizer we saw in the last video. You don't need to do anything with them yet, I just wanted to show you how sub-word tokenization works.

So now, let's take a look at classifying IMDB with it. What the results are going to be? Here's the model. Again, it should look very familiar at this point.

```
embedding_dim = 64
model = tf.keras.Sequential([
  tf.keras.layers.Embedding(tokenizer.vocab_size, embedding_dim),
  tf.keras.layers.GlobalAveragePooling1D(),
  tf.keras.layers.Dense(6, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.summary()
```

One thing to take into account though, is the shape of the vectors coming from the tokenizer through the embedding, and it's not easily flattened. So we'll use Global Average Pooling 1D instead. Trying to flatten them, will cause a TensorFlow crash.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
global_average_pooling1d_1 ( (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 6)                 390
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 7
=================================================================
Total params: 524,237
Trainable params: 524,237
Non-trainable params: 0
```

Here's the output of the model summary.

```
num_epochs = 10

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

history = model.fit(train_data,
                    epochs=num_epochs,
                    validation_data=test_data)
```

You can compile and train the model like this, it's pretty standard code. Training is dealing with a lot of hyper-parameters and sub-words, so expect it to be slow.

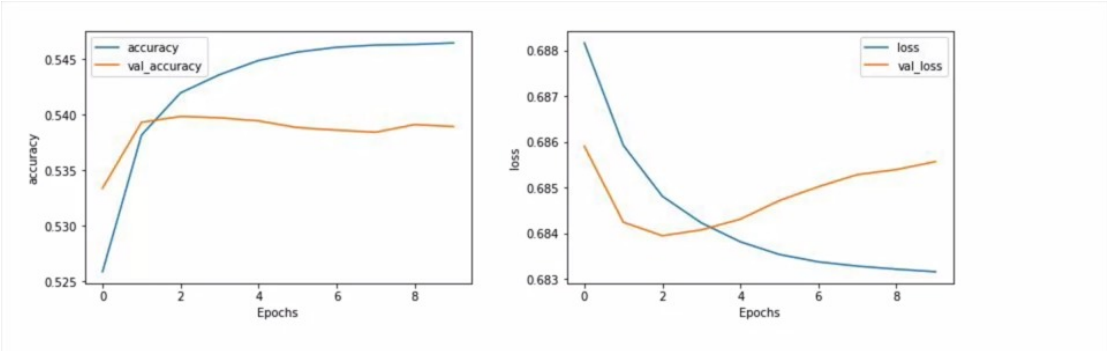
```
import matplotlib.pyplot as plt


def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string)
  plt.legend([string, 'val_'+string])
  plt.show()

plot_graphs(history, "acc")
plot_graphs(history, "loss")
```

deeplearning.ai

You can graph the results with this code, and your graphs will probably look something like this.



In my case, the accuracy was barely about 50 percent, which you could get with a random guess. While losses decreasing, it's decreasing in a very small way. So why do you think that might be?

Well, the keys in the fact that we're using **sub-words and not for-words**, sub-word meanings are often nonsensical and it's only when we put them together in sequences that they have meaningful semantics. Thus, some way from learning from sequences would be a great way forward, and that's exactly what you're going to do next week with recurrent neural networks

2.12 Notebook for lesson 3