audio

"crema_d"
"gtzan"
"gtzan_music_speech"
"libri_tts"
"ljspeech"
"nsynth"
"savee"
"speech_commands"
"tedlium"
"vctk"
"voxceleb"
"voxforge"

image

"abstract_reasoning"
"celeb_a"
"celeb_a_hq"
"cifar10"
"cifar100"
"cifar10_corrupted"
"coil100"
"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"
"svhn_cropped"
"tf_flowers"

structured

"bigearthnet"
"caltech_birds2010"
"caltech_birds2011"
"dmlab"
"kitti"
"mnist_corrupted"
"quickdraw_bitmap"
"sun397"

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"

audio

"imagenet2012"
"imagenet2012_corrupted"
"kmnist"
image
"abstract_reasoning"
"omniglot"
"open_images_v4"
"oxford_iiit_pet"
"quickdraw_bitmap"
"rock_paper_scissors"
"shapes3d"
"smallnorb"
"svhn_cropped"
"tf_flowers"
"diabetic_retinopathy..."
structured
"dsprites"
"mnist"
"fashion_mnist"
"horses_or_humans"
"image_label_folder"

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}
}
```

```
import tensorflow_datasets as tfds

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

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

single_example = list(imdb['train'].take(1))[0]

print(single_example[0])
```

```
tf.Tensor(b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great actors, but this must simply be their worst role in history. Even their great acting could not redeem this movie's ridiculous storyline. This movie is an early nineties US propaganda piece. The most pathetic scenes were those when the Columbian rebels were making their cases for revolutions. Maria Conchita Alonso appeared phony, and her pseudo-love affair with Walken was nothing but a pathetic emotional plug in a movie that was devoid of any real meaning. I am disappointed that there are movies like this, ruining actor's like Christopher Walken's good name. I could barely sit through it.", shape=(), dtype=string)
```

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

single_example = list(imdb['train'].take(1))[0]

print(single_example[0])
```

```
tf.Tensor(b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great actors, but this must simply be their worst role in history. Even their great acting could not redeem this movie's ridiculous storyline. This movie is an early nineties US propaganda piece. The most pathetic scenes were those when the Columbian rebels were making their cases for revolutions. Maria Conchita Alonso appeared phony, and her pseudo-love affair with Walken was nothing but a pathetic emotional plug in a movie that was devoid of any real meaning. I am disappointed that there are movies like this, ruining actor's like Christopher Walken's good name. I could barely sit through it.", shape=(), dtype=string)
```

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

single_example = list(imdb['train'].take(1))[0]

print(single_example[1])
```

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

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

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

train_reviews = train_dataset.map(lambda review, label: review)
train_labels = train_dataset.map(lambda review, label: label)

test_reviews = test_dataset.map(lambda review, label: review)
test_labels = test_dataset.map(lambda review, label: label)
```

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

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

train_reviews = train_dataset.map(lambda review, label: review)
train_labels = train_dataset.map(lambda review, label: label)

test_reviews = test_dataset.map(lambda review, label: review)
test_labels = test_dataset.map(lambda review, label: label)
```

```
import tensorflow_datasets as tfds

imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

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

train_reviews = train_dataset.map(lambda review, label: review)
train_labels = train_dataset.map(lambda review, label: label)

test_reviews = test_dataset.map(lambda review, label: review)
test_labels = test_dataset.map(lambda review, label: label)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
vectorize_layer = tf.keras.layers.TextVectorization(max_tokens=10000)

vectorize_layer.adapt(train_reviews)

def padding_func(sequences):
    sequences = sequences.ragged_batch(batch_size=sequences.cardinality())
    sequences = sequences.get_single_element()

    padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(), maxlen=120,
                                                    truncating='post', padding='pre')

    padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

    return padded_sequences

train_sequences = train_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
test_sequences = test_reviews.map(lambda text: vectorize_layer(text)).apply(padding_func)
```

```
train_dataset_vectorized = tf.data.Dataset.zip(train_sequences, train_labels)
test_dataset_vectorized = tf.data.Dataset.zip(test_sequences, test_labels)

SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

train_dataset_final = (train_dataset_vectorized
                       .cache()
                       .shuffle(SHUFFLE_BUFFER_SIZE)
                       .prefetch(PREFETCH_BUFFER_SIZE)
                       .batch(BATCH_SIZE)
                       )

test_dataset_final = (test_dataset_vectorized
                      .cache()
                      .prefetch(PREFETCH_BUFFER_SIZE)
                      .batch(BATCH_SIZE)
                      )
```

```
train_dataset_vectorized = tf.data.Dataset.zip(train_sequences, train_labels)
test_dataset_vectorized = tf.data.Dataset.zip(test_sequences, test_labels)

SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

train_dataset_final = (train_dataset_vectorized
                       .cache()
                       .shuffle(SHUFFLE_BUFFER_SIZE)
                       .prefetch(PREFETCH_BUFFER_SIZE)
                       .batch(BATCH_SIZE)
                       )

test_dataset_final = (test_dataset_vectorized
                      .cache()
                      .prefetch(PREFETCH_BUFFER_SIZE)
                      .batch(BATCH_SIZE)
                      )
```

```
train_dataset_vectorized = tf.data.Dataset.zip(train_sequences, train_labels)
test_dataset_vectorized = tf.data.Dataset.zip(test_sequences, test_labels)

SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

train_dataset_final = (train_dataset_vectorized
                       .cache()
                       .shuffle(SHUFFLE_BUFFER_SIZE)
                       .prefetch(PREFETCH_BUFFER_SIZE)
                       .batch(BATCH_SIZE)
                       )

test_dataset_final = (test_dataset_vectorized
                      .cache()
                      .prefetch(PREFETCH_BUFFER_SIZE)
                      .batch(BATCH_SIZE)
                      )
```

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(120,)),
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(120,)),
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(120,)),
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 120, 16)           160000
_________________________________________________________________
flatten_3 (Flatten)          (None, 1920)              0
_________________________________________________________________
dense_14 (Dense)             (None, 6)                 11526
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 7
=================================================================
Total params: 171,533
Trainable params: 171,533
Non-trainable params: 0
```

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(120,)),
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, 120, 16)           160000
_________________________________________________________________
global_average_pooling1d_3 ( (None, 16)                0
_________________________________________________________________
dense_16 (Dense)             (None, 6)                 102
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 7
=================================================================
Total params: 160,109
Trainable params: 160,109
Non-trainable params: 0
```

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

```
num_epochs = 10
model.fit(train_dataset_final,
          epochs=num_epochs,
          validation_data=test_dataset_final)
```

```
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
```

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

```
(10000, 16)
```

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

vocabulary = vectorize_layer.get_vocabulary()

for word_num in range(1, len(vocabulary)):
    word_name = vocabulary[word_num]
    word_embedding = embedding_weights[word_num]
    out_m.write(word_name + "\n")
    out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

out_v.close()
out_m.close()
```

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

vocabulary = vectorize_layer.get_vocabulary()

for word_num in range(1, len(vocabulary)):
    word_name = vocabulary[word_num]
    word_embedding = embedding_weights[word_num]
    out_m.write(word_name + "\n")
    out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

out_v.close()
out_m.close()
```

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

vocabulary = vectorize_layer.get_vocabulary()

for word_num in range(1, len(vocabulary)):
    word_name = vocabulary[word_num]
    word_embedding = embedding_weights[word_num]
    out_m.write(word_name + "\n")
    out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

out_v.close()
out_m.close()
```

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

vocabulary = vectorize_layer.get_vocabulary()

for word_num in range(1, len(vocabulary)):
    word_name = vocabulary[word_num]
    word_embedding = embedding_weights[word_num]
    out_m.write(word_name + "\n")
    out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

out_v.close()
out_m.close()
```

```
import io

out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

vocabulary = vectorize_layer.get_vocabulary()

for word_num in range(1, len(vocabulary)):
    word_name = vocabulary[word_num]
    word_embedding = embedding_weights[word_num]
    out_m.write(word_name + "\n")
    out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

out_v.close()
out_m.close()
```

https://projector.tensorflow.org

Embedding Projector

DATA

5 tensors found
Word2Vec 10K

Label by
word

Color by
No color map

Sphereize data

Load data | Publish

Checkpoint: Demo datasets

Metadata: oss_data/word2vec_10000_200d_labels.tsv

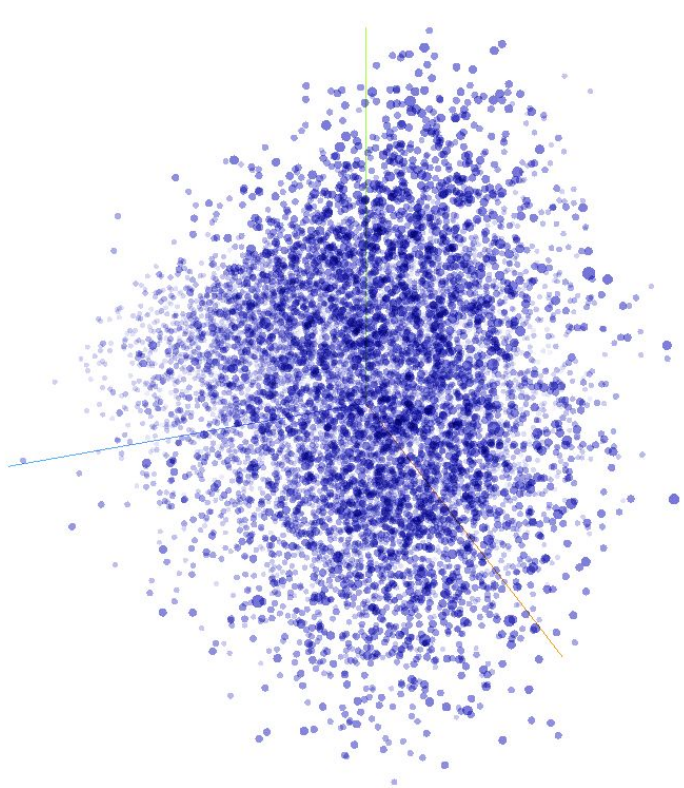
T-SNE | PCA | CUSTOM

X: Component #1
Y: Component #2
Z: Component #3

PCA is approximate.

Total variance described: 8.5%.

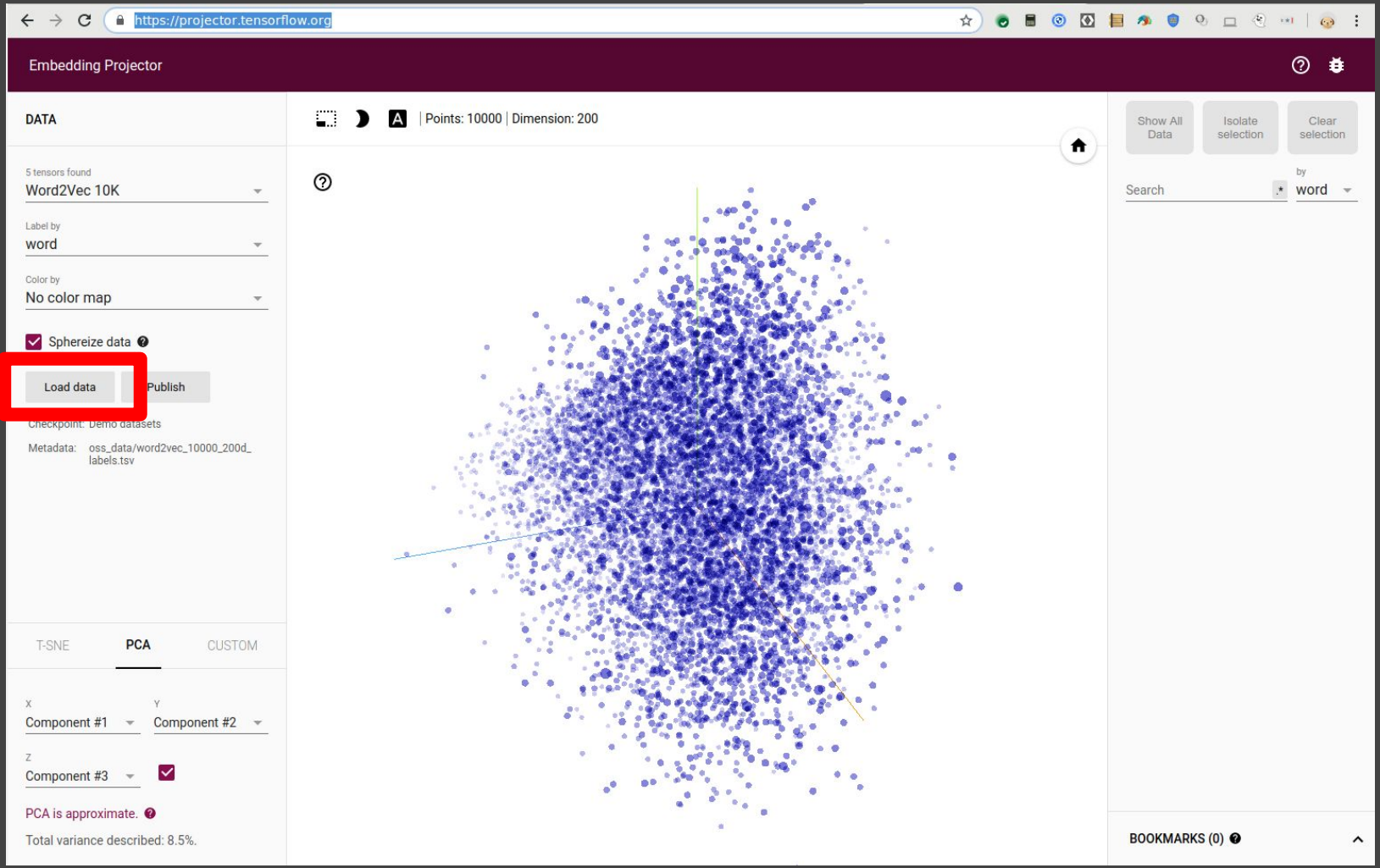
Points: 10000 | Dimension: 200



Show All Data | Isolate selection | Clear selection

Search | by word

BOOKMARKS (0)


https://projector.tensorflow.org
Embedding Projector
DATA
5 tensors found
Word2Vec 10K
Label by
word
Color by
No color map
Sphereize data
Load data
Publish
Checkpoint: Demo datasets
Metadata: oss_data/word2vec_10000_200d_labels.tsv
T-SNE
PCA
CUSTOM
X
Component #1
Y
Component #2
Z
Component #3
PCA is approximate.
Total variance described: 8.5%.
Points: 10000 | Dimension: 200
Show All Data
Isolate selection
Clear selection
Search
by
word
BOOKMARKS (0)

Load data from your computer

Step 1: **Load a TSV file of vectors.**

Example of 3 vectors with dimension 4:

```
0.1\t0.2\t0.5\t0.9
0.2\t0.1\t5.0\t0.2
0.4\t0.1\t7.0\t0.8
```

Choose file

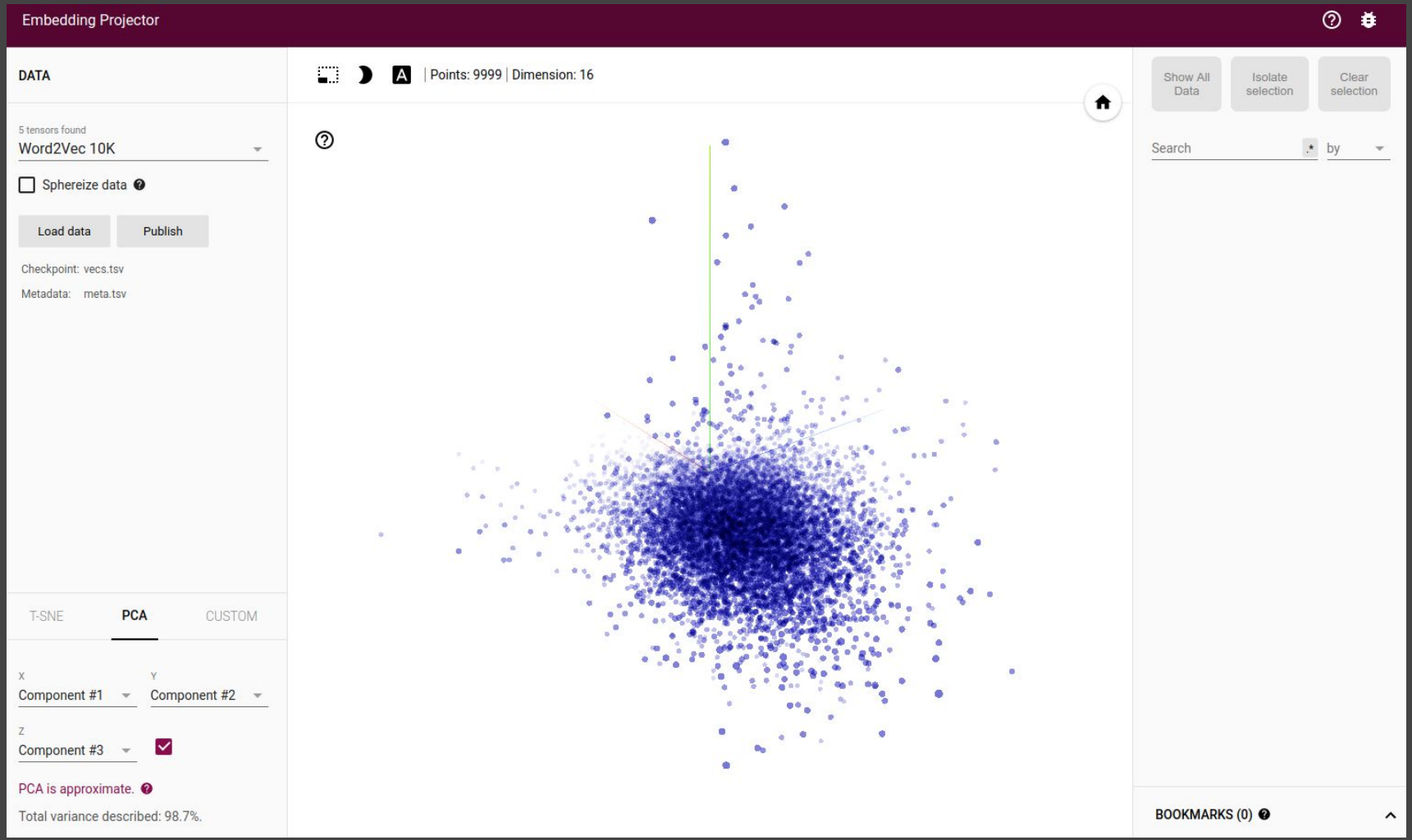
Step 2 (optional): **Load a TSV file of metadata.**

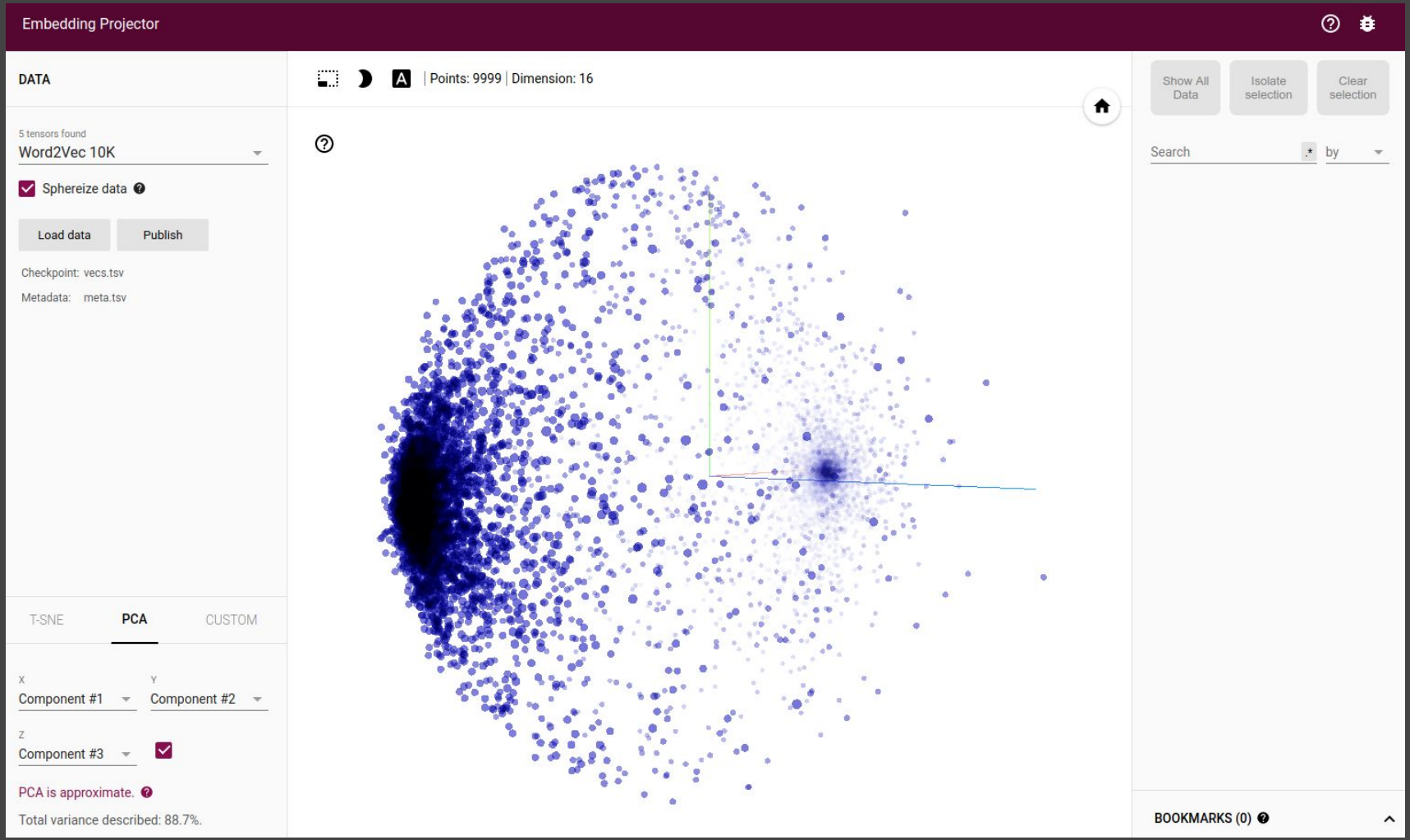
Example of 3 data points and 2 columns.
Note: If there is more than one column, the first row will be parsed as column labels.

```
Pokémon\tSpecies
Wartortle\tTurtle
Venusaur\tSeed
Charmeleon\tFlame
```

Choose file

Click outside to dismiss.


Embedding Projector
DATA
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: 98.7%.
Points: 9999 | Dimension: 16
Show All Data
Isolate selection
Clear selection
Search
by
BOOKMARKS (0)


Embedding Projector
DATA
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: 88.7%.
| Points: 9999 | Dimension: 16
Show All Data
Isolate selection
Clear selection
Search
.*
by
BOOKMARKS (0)

```
TRAINING_SIZE = 20000

VOCAB_SIZE = 10000

MAX_LENGTH = 32

EMBEDDING_DIM = 16
```

```
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'])
```

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

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

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

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

```
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_sequence_length=MAX_LENGTH)

vectorize_layer.adapt(train_sentences)

train_sequences = vectorize_layer(train_sentences)
test_sequences = vectorize_layer(test_sentences)

train_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (train_sequences, train_labels))
test_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (test_sequences, test_labels))
```

```
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_sequence_length=MAX_LENGTH)

vectorize_layer.adapt(train_sentences)

train_sequences = vectorize_layer(train_sentences)
test_sequences = vectorize_layer(test_sentences)

train_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (train_sequences, train_labels))
test_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (test_sequences, test_labels))
```

```
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_sequence_length=MAX_LENGTH)

vectorize_layer.adapt(train_sentences)

train_sequences = vectorize_layer(train_sentences)
test_sequences = vectorize_layer(test_sentences)

train_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (train_sequences, train_labels))
test_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (test_sequences, test_labels))
```

```
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_sequence_length=MAX_LENGTH)

vectorize_layer.adapt(train_sentences)

train_sequences = vectorize_layer(train_sentences)
test_sequences = vectorize_layer(test_sentences)

train_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (train_sequences, train_labels))
test_dataset_vectorized = tf.data.Dataset.from_tensor_slices(
    (test_sequences, test_labels))
```

```
SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

train_dataset_final = (train_dataset_vectorized
                       .cache()
                       .shuffle(SHUFFLE_BUFFER_SIZE)
                       .prefetch(PREFETCH_BUFFER_SIZE)
                       .batch(BATCH_SIZE)
                       )

test_dataset_final = (test_dataset_vectorized
                      .cache()
                      .prefetch(PREFETCH_BUFFER_SIZE)
                      .batch(BATCH_SIZE)
                      )
```

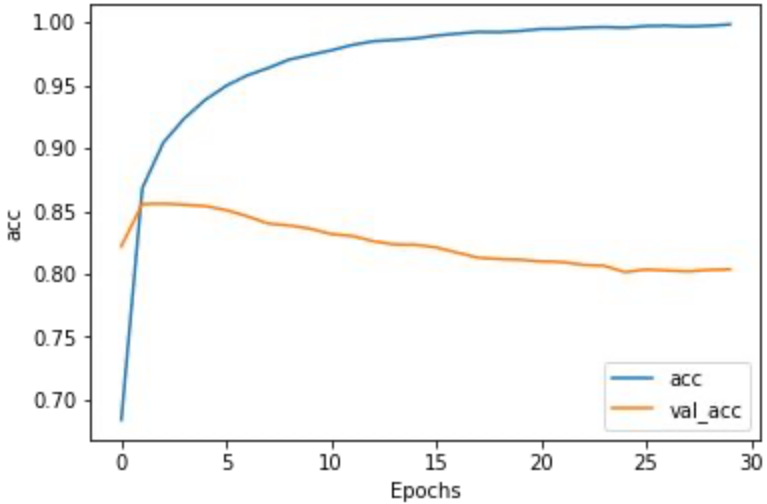
```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    tf.keras.layers.Embedding(VOCAB_SIZE, EMBEDDING_DIM),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

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

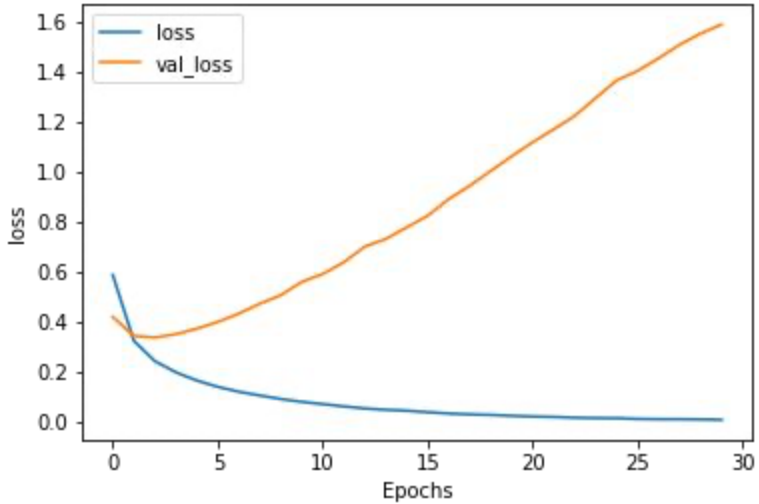
```
model.summary()
```

```
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
```

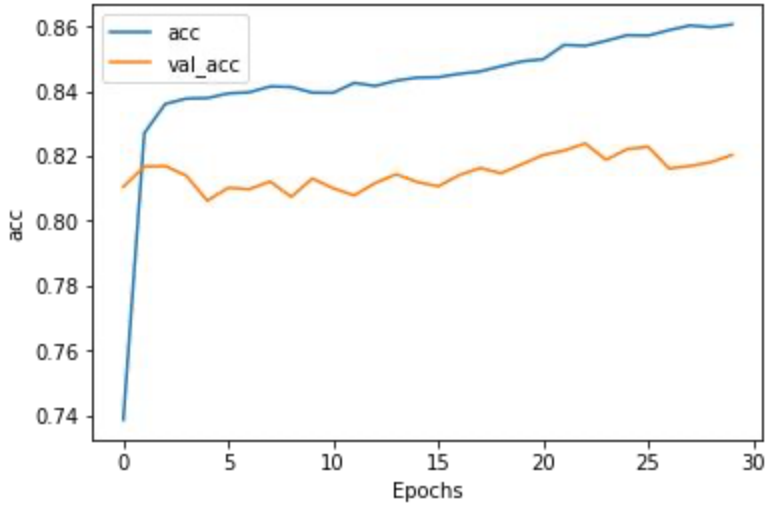
```
num_epochs = 30

history = model.fit(train_dataset_final, epochs=num_epochs,
                    validation_data=test_dataset_final, verbose=2)
```

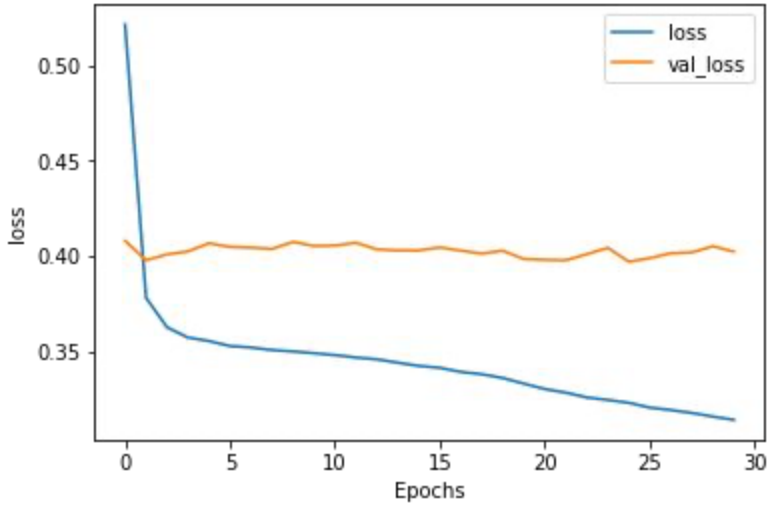
```
import matplotlib.pyplot as plt


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


1.00
0.95
0.90
0.85
0.80
0.75
0.70
acc
0
5
10
15
20
25
30
Epochs
acc
val_acc

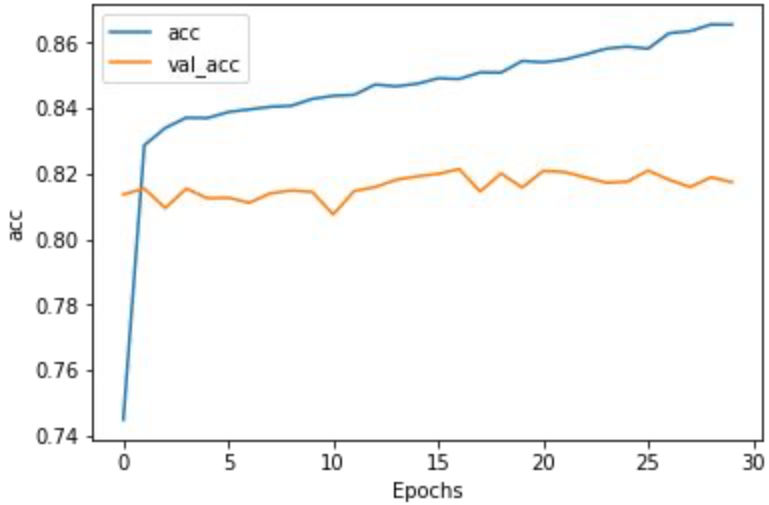

1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
5
10
15
20
25
30
Epochs
loss
val_loss

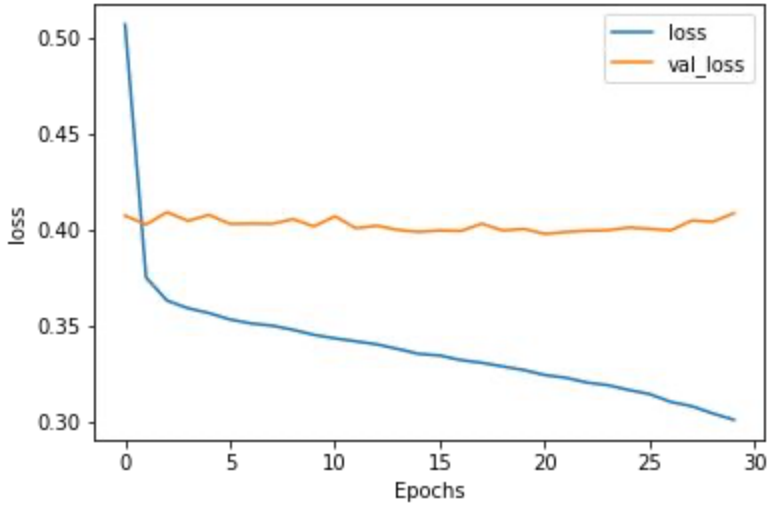
```
TRAINING_SIZE = 20000

VOCAB_SIZE = 1000 (was 10000)

MAX_LENGTH = 16   (was 32)

EMBEDDING_DIM = 16
```


acc
val_acc
0.86
0.84
0.82
0.80
0.78
0.76
0.74
acc
0
5
10
15
20
25
30
Epochs

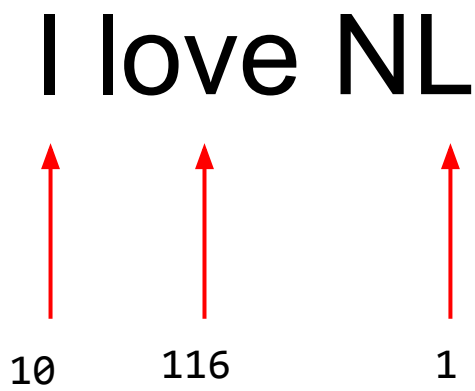

loss
val_loss
0.50
0.45
0.40
0.35
loss
0
5
10
15
20
25
30
Epochs

```
TRAINING_SIZE = 20000

VOCAB_SIZE = 1000  (was 10000)

MAX_LENGTH = 16    (was 32)

EMBEDDING_DIM = 32 (was 16)
```

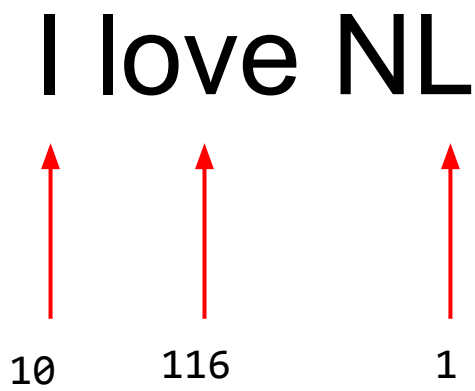

acc
val_acc
0.86
0.84
0.82
0.80
0.78
0.76
0.74
acc
0
5
10
15
20
25
30
Epochs


loss
val_loss
0.50
0.45
0.40
0.35
0.30
loss
0
5
10
15
20
25
30
Epochs

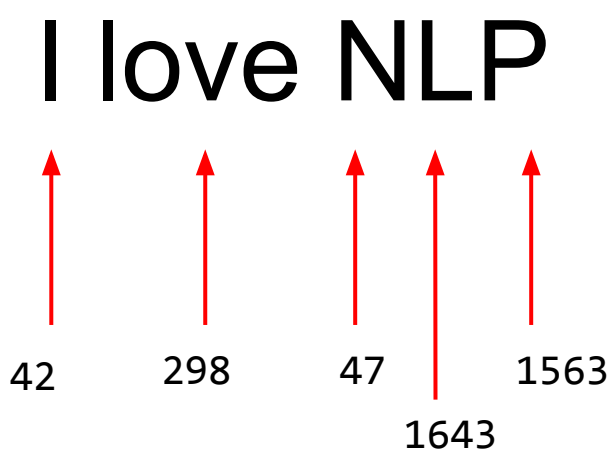
Word tokenization



Word tokenization



Subword tokenization



https://github.com/tensorflow/datasets/tree/master/docs/catalog

https://github.com/tensorflow/datasets/blob/master/docs/catalog/imdb_reviews.md

imdb_reviews

- **Description:**

Large Movie Review Dataset. This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. We provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well.

- **Homepage**: http://ai.stanford.edu/~amaas/data/sentiment/
- **Source code**: `tfds.text.IMDBReviews`
- **Versions:**
  - `1.0.0` (default): New split API (https://tensorflow.org/datasets/splits)
- **Download size**: `80.23 MiB`
- **Dataset size**: `Unknown size`
- **Auto-cached** (documentation): Unknown
- **Splits:**

Split	Examples
`'test'`	25,000
`'train'`	25,000
`'unsupervised'`	50,000

https://github.com/tensorflow/datasets/blob/master/docs/catalog/imdb_reviews.md

275 lines (209 sloc) | 9.65 KB

Raw Blame

imdb_reviews/plain_text (default config)

- **Config description**: Plain text
- **Features**:

```
FeaturesDict({
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    'text': Text(shape=(), dtype=tf.string),
})
```

- **Examples** (tfds.as_dataframe):

{% framebox %}

Display examples...

<script> const url = "https://storage.googleapis.com/tfds-data/visualization/dataframe/imdb_reviews-plain_text-1.0.0.html"; const dataButton = document.getElementById('displaydataframe'); dataButton.addEventListener('click', async () => { // Disable the button after clicking (dataframe loaded only once). dataButton.disabled = true;
const contentPane = document.getElementById('dataframecontent'); try { const response = await fetch(url); // Error response codes don't throw an error, so force an error to show // the error message. if (!response.ok) throw Error(response.statusText);

```
const data = await response.text();
contentPane.innerHTML = data;
```

} catch (e) { contentPane.innerHTML = 'Error loading examples. If the error persist, please open ' + 'a new issue.'; } }); </script>

{% endframebox %}

https://github.com/tensorflow/datasets/blob/master/docs/catalog/imdb_reviews.md

imdb_reviews/bytes

- **Config description**: Uses byte-level text encoding with `tfds.deprecated.text.ByteTextEncoder`
- **Features**:

```
FeaturesDict({
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    'text': Text(shape=(None,), dtype=tf.int64, encoder=<ByteTextEncoder vocab_size=257>),
})
```

- **Examples** (tfds.as_dataframe):

{% framebox %}

Display examples...

<script> const url = "https://storage.googleapis.com/tfds-data/visualization/dataframe/imdb_reviews-bytes-1.0.0.html"; const dataButton = document.getElementById('displaydataframe'); dataButton.addEventListener('click', async () => { // Disable the button after clicking (dataframe loaded only once). dataButton.disabled = true;
const contentPane = document.getElementById('dataframecontent'); try { const response = await fetch(url); // Error response codes don't throw an error, so force an error to show // the error message. if (!response.ok) throw Error(response.statusText);

```
const data = await response.text();
contentPane.innerHTML = data;
```

} catch (e) { contentPane.innerHTML = 'Error loading examples. If the error persist, please open ' + 'a new issue.'; } }); </script>

{% endframebox %}

https://github.com/tensorflow/datasets/blob/master/docs/catalog/imdb_reviews.md

275 lines (209 sloc) | 9.65 KB

Raw Blame

imdb_reviews/subwords8k

- **Config description**: Uses `tfds.deprecated.text.SubwordTextEncoder` with 8k vocab size
- **Features**:

```
FeaturesDict({
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    'text': Text(shape=(None,), dtype=tf.int64, encoder=<SubwordTextEncoder vocab_size=8185>),
})
```

- **Examples** (tfds.as_dataframe):

{% framebox %}

Display examples...

<script> const url = "https://storage.googleapis.com/tfds-data/visualization/dataframe/imdb_reviews-subwords8k-1.0.0.html"; const dataButton = document.getElementById('displaydataframe'); dataButton.addEventListener('click', async () => { // Disable the button after clicking (dataframe loaded only once). dataButton.disabled = true;
const contentPane = document.getElementById('dataframecontent'); try { const response = await fetch(url); // Error response codes don't throw an error, so force an error to show // the error message. if (!response.ok) throw Error(response.statusText);

```
const data = await response.text();
contentPane.innerHTML = data;
```

} catch (e) { contentPane.innerHTML = 'Error loading examples. If the error persist, please open ' + 'a new issue.'; } }); </script>

{% endframebox %}

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

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

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

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

```
import keras_nlp

imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir='./data', download=False)

train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

keras_nlp.tokenizers.compute_word_piece_vocabulary(
    train_reviews,
    vocabulary_size=8000,
    reserved_tokens=["[PAD]", "[UNK]"],
    vocabulary_output_file='imdb_vocab_subwords.txt'
)

subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

```
import keras_nlp

imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir='./data', download=False)

train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

keras_nlp.tokenizers.compute_word_piece_vocabulary(
    train_reviews,
    vocabulary_size=8000,
    reserved_tokens=["[PAD]", "[UNK]"],
    vocabulary_output_file='imdb_vocab_subwords.txt'
)

subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

```
import keras_nlp

imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir='./data', download=False)

train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

keras_nlp.tokenizers.compute_word_piece_vocabulary(
    train_reviews,
    vocabulary_size=8000,
    reserved_tokens=["[PAD]", "[UNK]"],
    vocabulary_output_file='imdb_vocab_subwords.txt'
)

subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

```
import keras_nlp

imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir='./data', download=False)

train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

keras_nlp.tokenizers.compute_word_piece_vocabulary(
    train_reviews,
    vocabulary_size=8000,
    reserved_tokens=["[PAD]", "[UNK]"],
    vocabulary_output_file='imdb_vocab_subwords.txt'
)

subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

```
import keras_nlp

imdb = tfds.load("imdb_reviews", as_supervised=True, data_dir='./data', download=False)

train_reviews = imdb['train'].map(lambda review, label: review)
train_labels = imdb['train'].map(lambda review, label: label)

keras_nlp.tokenizers.compute_word_piece_vocabulary(
    train_reviews,
    vocabulary_size=8000,
    reserved_tokens=["[PAD]", "[UNK]"],
    vocabulary_output_file='imdb_vocab_subwords.txt'
)

subword_tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary='./imdb_vocab_subwords.txt'
)
```

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

tokenized_string = subword_tokenizer.tokenize(sample_string)
print('Tokenized string is {}'.format(tokenized_string))

original_string = subword_tokenizer.detokenize(tokenized_string).numpy().decode("utf-8")
print('The original string: {}'.format(original_string))
```

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

tokenized_string = subword_tokenizer.tokenize(sample_string)
print('Tokenized string is {}'.format(tokenized_string))

original_string = subword_tokenizer.detokenize(tokenized_string).numpy().decode("utf-8")
print('The original string: {}'.format(original_string))
```

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

tokenized_string = subword_tokenizer.tokenize(sample_string)
print('Tokenized string is {}'.format(tokenized_string))

original_string = subword_tokenizer.detokenize(tokenized_string).numpy().decode("utf-8")
print('The original string: {}'.format(original_string))
```

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

tokenized_string = subword_tokenizer.tokenize(sample_string)
print('Tokenized string is {}'.format(tokenized_string))

original_string = subword_tokenizer.detokenize(tokenized_string).numpy().decode("utf-8")
print('The original string: {}'.format(original_string))
```

```
Tokenized string is [  53 2235  543 1827 3024   13  198 1659  174  167 2220  238]
The original string: TensorFlow , from basics to mastery
```

```
for i in range(len(tokenized_string)):
    subword = subword_tokenizer.detokenize(tokenized_string[i:i+1]).numpy().decode("utf-8")
    print(subword)
```

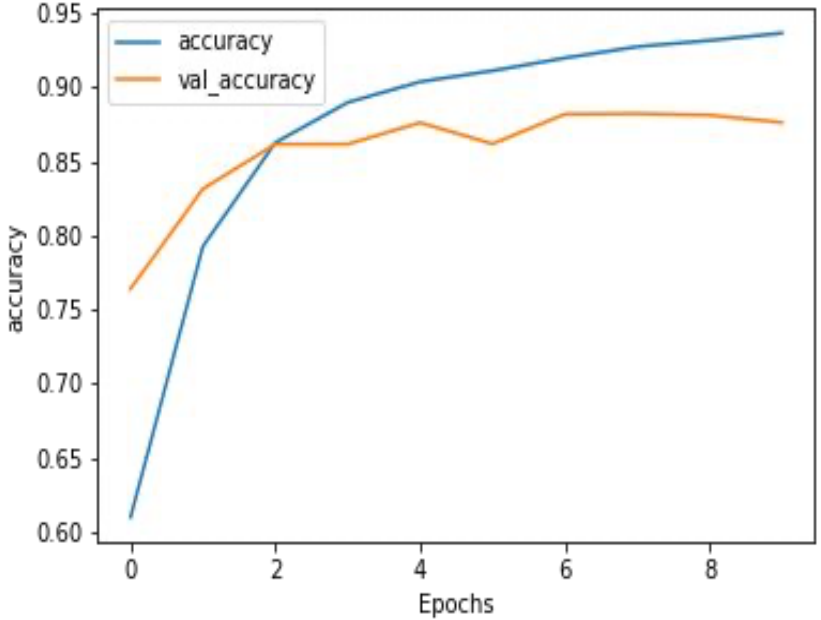
```
for i in range(len(tokenized_string)):
    subword = subword_tokenizer.detokenize(tokenized_string[i:i+1]).numpy().decode("utf-8")
    print(subword)
```

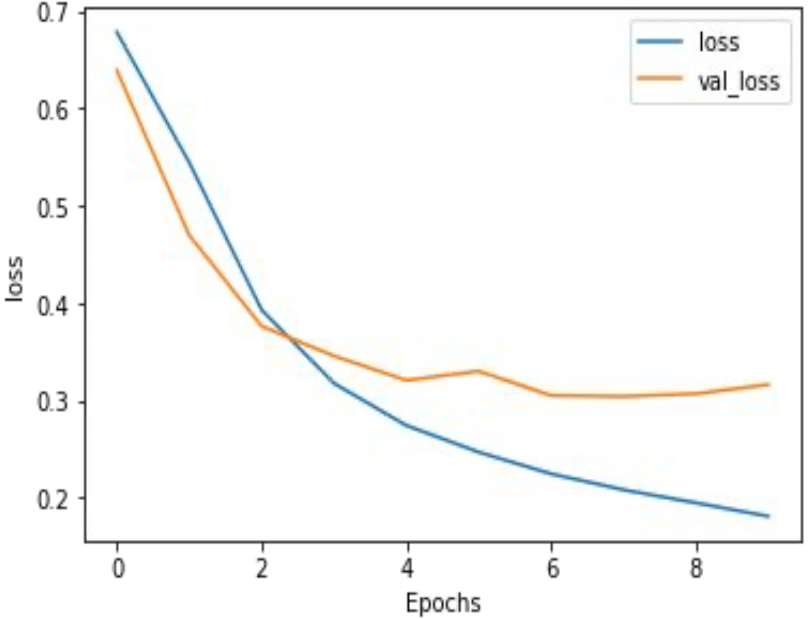
```
T
##ens
##or
##F
##low
,
from
basic
##s
to
master
##y
```

```
embedding_dim = 64
model = tf.keras.Sequential([
  tf.keras.Input(shape=(MAX_LENGTH,)),
  tf.keras.layers.Embedding(subword_tokenizer.vocabulary_size(), EMBEDDING_DIM),
  tf.keras.layers.GlobalAveragePooling1D(),
  tf.keras.layers.Dense(6, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 120, 64)           488,460
_________________________________________________________________
global_average_pooling1d_1 ( (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 6)                 390
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 7
=================================================================
Total params: 489,037
Trainable params: 489,037
Non-trainable params: 0
```

```
num_epochs = 10

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

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


accuracy
val_accuracy
accuracy
Epochs
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0
2
4
6
8


loss
val_loss
loss
Epochs
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8