Ungraded Lab: Training a binary classifier with the IMDB Reviews Dataset

In this lab, you will be building a sentiment classification model to distinguish between positive and negative movie reviews. You will train it on the IMDB Reviews dataset and visualize the word embeddings generated after training.

Imports

As usual, you will start by importing the necessary packages.

In [1]:
```
import tensorflow_datasets as tfds
import tensorflow as tf
import io
```

Download the Dataset

You will load the dataset via Tensorflow Datasets, a collection of prepared datasets for machine learning. If you're running this notebook on your local machine, make sure to have the `tensorflow-datasets` package installed before importing it. You can install it via `pip` as shown in the commented cell below.

In [ ]:
```
# Install this package if running on your local machine
# !pip install -q tensorflow-datasets
```

The `tfds.load()` method downloads the dataset into your working directory. You can set the `with_info` parameter to `True` if you want to see the description of the dataset. The `as_supervised` parameter, on the other hand, is set to load the data as `(input, label)` pairs.

To ensure smooth operation, the data was pre-downloaded and saved in the data folder. When you have the data already downloaded, you can read it by passing two additional arguments. With `data_dir="./data/"`, you specify the folder where the data is located (if different than default) and by setting `download=False` you explicitly tell the method to read the data from the folder, rather than downloading it.

In [2]:
```
# Load the IMDB Reviews dataset
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True, data_dir="./data/", download=False)
```

In [3]:
```
# Print information about the dataset
print(info)
```

```
tfds.core.DatasetInfo(
    name='imdb_reviews',
    full_name='imdb_reviews/plain_text/1.0.0',
    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.
    """,
    config_description="""
    Plain text
    """,
    homepage='http://ai.stanford.edu/~amaas/data/sentiment/',
    data_dir='./data/imdb_reviews/plain_text/1.0.0',
    file_format=tfrecord,
    download_size=80.23 MiB,
    dataset_size=129.83 MiB,
    features=FeaturesDict({
        'label': ClassLabel(shape=(), dtype=int64, num_classes=2),
        'text': Text(shape=(), dtype=string),
    }),
    supervised_keys=('text', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=25000, num_shards=1>,
        'train': <SplitInfo num_examples=25000, num_shards=1>,
        'unsupervised': <SplitInfo num_examples=50000, num_shards=1>,
    },
    citation="""@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 Lan
guage 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}
    }""",
)
```

As you can see in the output above, there is a total of 100,000 examples in the dataset and it is split into `train`, `test` and `unsupervised` sets. For this lab, you will only use `train` and `test` sets because you will need labeled examples to train your model.

Split the dataset

The `imdb` dataset that you downloaded earlier contains a dictionary pointing to `tf.data.Dataset` objects.

In [4]:
```python
# Print the contents of the dataset
print(imdb)
```

```
{'train': <_PrefetchDataset element_spec=(TensorSpec(shape=(), dtype=tf.string, name=None), TensorSpec(shape=(), dtyp
e=tf.int64, name=None))>, 'test': <_PrefetchDataset element_spec=(TensorSpec(shape=(), dtype=tf.string, name=None), T
ensorSpec(shape=(), dtype=tf.int64, name=None))>, 'unsupervised': <_PrefetchDataset element_spec=(TensorSpec(shape=
(), dtype=tf.string, name=None), TensorSpec(shape=(), dtype=tf.int64, name=None))>}
```

You can preview the raw format of a few examples by using the `take()` method and iterating over it as shown below:

In [5]:
```python
# View 4 training examples
for example in imdb['train'].take(4):
  print(example)
```

```
(<tf.Tensor: shape=(), dtype=string, numpy=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 gr
eat acting could not redeem this movie's ridiculous storyline. This movie is an early nineties US propaganda piece. T
he 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 th
at was devoid of any real meaning. I am disappointed that there are movies like this, ruining actor's like Christophe
r Walken's good name. I could barely sit through it.">, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
(<tf.Tensor: shape=(), dtype=string, numpy=b'I have been known to fall asleep during films, but this is usually due t
o a combination of things including, really tired, being warm and comfortable on the sette and having just eaten a lo
t. However on this occasion I fell asleep because the film was rubbish. The plot development was constant. Constantly
slow and boring. Things seemed to happen, but with no explanation of what was causing them or why. I admit, I may hav
e missed part of the film, but i watched the majority of it and everything just seemed to happen of its own accord wi
thout any real concern for anything else. I cant recommend this film at all.'>, <tf.Tensor: shape=(), dtype=int64, nu
mpy=0>)
(<tf.Tensor: shape=(), dtype=string, numpy=b'Mann photographs the Alberta Rocky Mountains in a superb fashion, and Ji
mmy Stewart and Walter Brennan give enjoyable performances as they always seem to do. <br /><br />But come on Hollywo
od - a Mountie telling the people of Dawson City, Yukon to elect themselves a marshal (yes a marshal!) and to enforce
the law themselves, then gunfighters battling it out on the streets for control of the town? <br /><br />Nothing even
remotely resembling that happened on the Canadian side of the border during the Klondike gold rush. Mr. Mann and comp
any appear to have mistaken Dawson City for Deadwood, the Canadian North for the American Wild West.<br /><br />Canad
ian viewers be prepared for a Reefer Madness type of enjoyable howl with this ludicrous plot, or, to shake your head
in disgust.'>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
(<tf.Tensor: shape=(), dtype=string, numpy=b'This is the kind of film for a snowy Sunday afternoon when the rest of t
he world can go ahead with its own business as you descend into a big arm-chair and mellow for a couple of hours. Won
derful performances from Cher and Nicolas Cage (as always) gently row the plot along. There are no rapids to cross, n
o dangerous waters, just a warm and witty paddle through New York life at its best. A family film in every sense and
one that deserves the praise it received.'>, <tf.Tensor: shape=(), dtype=int64, numpy=1>)
```

You can see that each example is a 2-element tuple of tensors containing the text first, then the label (shown in the `numpy()` property). The next cell below will take all the `train` and `test` sentences and labels into separate lists so you can preprocess the text and feed it to the model later.

In [6]:
```python
# Get the train and test sets
train_dataset, test_dataset = imdb['train'], imdb['test']
```

Generate Padded Sequences

Now you can do the text preprocessing steps you've learned last week. You will convert the strings to integer sequences, then pad them to a uniform length. The parameters are separated into its own code cell below so it will be easy for you to tweak it later if you want.

In [7]:
```python
# Parameters

VOCAB_SIZE = 10000
MAX_LENGTH = 120
EMBEDDING_DIM = 16
PADDING_TYPE = 'pre'
TRUNC_TYPE = 'post'
```

An important thing to note here is you should generate the vocabulary based only on the training set. You should not include the test set because that is meant to represent data that the model hasn't seen before. With that, you can expect more unknown tokens (i.e. the value `1`) in the integer sequences of the test data. Also for clarity in demonstrating the transformations, you will first separate the reviews and labels. You will see other ways to implement the data pipeline in the next labs.

In [8]:
```python
# Instantiate the vectorization layer
vectorize_layer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
)

# Get the string inputs and integer outputs of the training set
train_reviews = train_dataset.map(lambda review, label: review)
train_labels = train_dataset.map(lambda review, label: label)

# Get the string inputs and integer outputs of the test set
test_reviews = test_dataset.map(lambda review, label: review)
test_labels = test_dataset.map(lambda review, label: label)

# Generate the vocabulary based only on the training set
vectorize_layer.adapt(train_reviews)
```

You will define a padding function to generate the padded sequences. Note that the `pad_sequences()` function expects an iterable (e.g. list) while the input to this function is a `tf.data.Dataset`. Here's one way to do the conversion:

- Put all the elements in a single ragged batch (i.e. batch with elements that have different lengths).
  - You will need to specify the batch size and it has to match the number of all elements in the dataset. From the output of the dataset info earlier, you know that this should be 25000.
  - Instead of specifying a specific number, you can also use the cardinality() method. This computes the number of elements in a `tf.data.Dataset`.
- Use the get_single_element() method on the single batch to output a Tensor.
- Convert back to a `tf.data.Dataset`. You'll see why this is needed in the next cell.

In [9]:
```
def padding_func(sequences):
  '''Generates padded sequences from a tf.data.Dataset'''

  # Put all elements in a single ragged batch
  sequences = sequences.ragged_batch(batch_size=sequences.cardinality())

  # Output a tensor from the single batch
  sequences = sequences.get_single_element()

  # Pad the sequences
  padded_sequences = tf.keras.utils.pad_sequences(sequences.numpy(),
                                                  maxlen=MAX_LENGTH,
                                                  truncating=TRUNC_TYPE,
                                                  padding=PADDING_TYPE
                                                 )

  # Convert back to a tf.data.Dataset
  padded_sequences = tf.data.Dataset.from_tensor_slices(padded_sequences)

  return padded_sequences
```

This is the pipeline to convert the raw string inputs to padded integer sequences:

- Use the map() method to pass each string to the `TextVectorization` layer defined earlier.
- Use the apply() method to use the padding function on the entire dataset.

The difference between `map()` and `apply()` is the mapping function in `map()` expects its input to be single elements (i.e. element-wise transformations), while the transformation function in `apply()` expects its input to be the entire dataset in the pipeline.

In [10]:
```
# Apply the layer to the train and test data
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)
```

You can take a few examples from the results and observe that the raw strings are now converted to padded integer sequences.

In [11]:
```
# View 2 training sequences
for example in train_sequences.take(2):
  print(example)
  print()
```

```
tf.Tensor(
[   0    0    0    0   11   14   34  412  384   18   90   28    1    8
   33 1322 3560   42  487    1  191   24   85  152   19   11  217  316
   28   65  240  214    8  489   54   65   85  112   96   22 5596   11
   93  642  743   11   18    7   34  394 9522  170 2464  408    2   88
 1216  137   66  144   51    2    1 7558   66  245   65 2870   16    1
 2860    1    1 1426 5050    3   40    1 1579   17 3560   14  158   19
    4 1216  891 8040    8    4   18   12   14 4059    5   99  146 1241
   10  237  704   12   48   24   93   39   11 7339  152   39 1322    1
   50  398   10   96 1155  851  141    9], shape=(120,), dtype=int32)

tf.Tensor(
[   0    0    0    0    0    0    0    0   10   26   75  617    6  776
 2355  299   95   19   11    7  604  662    6    4 2129    5  180  571
   63 1403  107 2410    3 3905   21    2    1    3  252   41 4781    4
  169  186   21   11 4259   10 1507 2355   80    2   20   14 1973    2
  114  943   14 1740 1300  594    3  356  180  446    6  596   19   17
   57 1775    5   49   14 4002   98   42  134   10  934   10  194   26
 1026  171    5    2   20   19   10  284    2 2065    5    9    3  279
   41  446    6  596    5   30  200    1  201   99  146 4525   16  229
  329   10  175  368   11   20   31   32], shape=(120,), dtype=int32)
```

You will now re-combine the sequences with the labels to prepare it for training.

In [12]:
```python
train_dataset_vectorized = tf.data.Dataset.zip(train_sequences,train_labels)
test_dataset_vectorized = tf.data.Dataset.zip(test_sequences,test_labels)
```

In [13]:
```python
# View 2 training sequences and its labels
for example in train_dataset_vectorized.take(2):
  print(example)
  print()
```

```
(<tf.Tensor: shape=(120,), dtype=int32, numpy=
array([   0,    0,    0,    0,   11,   14,   34,  412,  384,   18,   90,
         28,    1,    8,   33, 1322, 3560,   42,  487,    1,  191,   24,
         85,  152,   19,   11,  217,  316,   28,   65,  240,  214,    8,
        489,   54,   65,   85,  112,   96,   22, 5596,   11,   93,  642,
        743,   11,   18,    7,   34,  394, 9522,  170, 2464,  408,    2,
         88, 1216,  137,   66,  144,   51,    2,    1, 7558,   66,  245,
         65, 2870,   16,    1, 2860,    1,    1, 1426, 5050,    3,   40,
          1, 1579,   17, 3560,   14,  158,   19,    4, 1216,  891, 8040,
          8,    4,   18,   12,   14, 4059,    5,   99,  146, 1241,   10,
        237,  704,   12,   48,   24,   93,   39,   11, 7339,  152,   39,
       1322,    1,   50,  398,   10,   96, 1155,  851,  141,    9],
      dtype=int32)>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)

(<tf.Tensor: shape=(120,), dtype=int32, numpy=
array([   0,    0,    0,    0,    0,    0,    0,    0,   10,   26,   75,
        617,    6,  776, 2355,  299,   95,   19,   11,    7,  604,  662,
          6,    4, 2129,    5,  180,  571,   63, 1403,  107, 2410,    3,
       3905,   21,    2,    1,    3,  252,   41, 4781,    4,  169,  186,
         21,   11, 4259,   10, 1507, 2355,   80,    2,   20,   14, 1973,
          2,  114,  943,   14, 1740, 1300,  594,    3,  356,  180,  446,
          6,  596,   19,   17,   57, 1775,    5,   49,   14, 4002,   98,
         42,  134,   10,  934,   10,  194,   26, 1026,  171,    5,    2,
         20,   19,   10,  284,    2, 2065,    5,    9,    3,  279,   41,
        446,    6,  596,    5,   30,  200,    1,  201,   99,  146, 4525,
         16,  229,  329,   10,  175,  368,   11,   20,   31,   32],
      dtype=int32)>, <tf.Tensor: shape=(), dtype=int64, numpy=0>)
```

Lastly, you will optimize and batch the datasets.

In [14]:
```python
SHUFFLE_BUFFER_SIZE = 1000
PREFETCH_BUFFER_SIZE = tf.data.AUTOTUNE
BATCH_SIZE = 32

# Optimize the datasets for training
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)
                      )
```

Build and Compile the Model

With the data already preprocessed, you can proceed to building your sentiment classification model. The input will be an `Embedding` layer. The main idea here is to represent each word in your vocabulary with vectors. These vectors have trainable weights so as your neural network learns, words that are most likely to appear in a positive tweet will converge towards similar weights. Similarly, words in negative tweets will be clustered more closely together. You can read more about word embeddings here.

After the `Embedding` layer, you will flatten its output and feed it into a `Dense` layer. You will explore other architectures for these hidden layers in the next labs.

The output layer would be a single neuron with a sigmoid activation to distinguish between the 2 classes. As is typical with binary classifiers, you will use the `binary_crossentropy` as your loss function while training.

In [15]:
```
# Build the model
model = tf.keras.Sequential([
    tf.keras.Input(shape=(MAX_LENGTH,)),
    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')
])

# Setup the training parameters
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

# Print the model summary
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 120, 16)	160,000
flatten (Flatten)	(None, 1920)	0
dense (Dense)	(None, 6)	11,526
dense_1 (Dense)	(None, 1)	7

Total params: 171,533 (670.05 KB)
Trainable params: 171,533 (670.05 KB)
Non-trainable params: 0 (0.00 B)

Train the Model

Next, of course, is to train your model. With the current settings, you will get near perfect training accuracy after just 5 epochs but the validation accuracy will only be at around 80%. See if you can still improve this by adjusting some of the parameters earlier (e.g. the `VOCAB_SIZE`, number of `Dense` neurons, number of epochs, etc.).

In [16]:
```
NUM_EPOCHS = 5

# Train the model
model.fit(train_dataset_final, epochs=NUM_EPOCHS, validation_data=test_dataset_final)
```

```
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.6313 - loss: 0.6126 - val_accuracy: 0.8241 - val_loss: 0.3865
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.8748 - loss: 0.2954 - val_accuracy: 0.8122 - val_loss: 0.4339
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9554 - loss: 0.1379 - val_accuracy: 0.8111 - val_loss: 0.5095
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9928 - loss: 0.0365 - val_accuracy: 0.8048 - val_loss: 0.6001
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9986 - loss: 0.0095 - val_accuracy: 0.8032 - val_loss: 0.6790
```

Out[16]:
```
<keras.src.callbacks.history.History at 0x79778147a5d0>
```

Visualize Word Embeddings

After training, you can visualize the trained weights in the `Embedding` layer to see words that are clustered together. The Tensorflow Embedding Projector is able to reduce the 16-dimension vectors you defined earlier into fewer components so it can be plotted in the projector. First, you will need to get these weights and you can do that with the cell below:

In [17]:
```
# Get the embedding layer from the model (i.e. first layer)
embedding_layer = model.layers[0]

# Get the weights of the embedding layer
embedding_weights = embedding_layer.get_weights()[0]

# Print the shape. Expected is (vocab_size, embedding_dim)
print(embedding_weights.shape)
```

```
(10000, 16)
```

You will need to generate two files:

- `vecs.tsv` - contains the vector weights of each word in the vocabulary
- `meta.tsv` - contains the words in the vocabulary

You will get the word list from the `TextVectorization` layer you adapted earler, then start the loop to generate the files. You will loop `vocab_size-1` times, skipping the `0` key because it is just for the padding.

In [18]:
```
# Open writeable files
out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

# Get the word list
vocabulary = vectorize_layer.get_vocabulary()

# Initialize the loop. Start counting at `1` because `0` is just for the padding
for word_num in range(1, len(vocabulary)):

  # Get the word associated withAttributeError the current index
  word_name = vocabulary[word_num]

  # Get the embedding weights associated with the current index
  word_embedding = embedding_weights[word_num]

  # Write the word name
  out_m.write(word_name + "\n")

  # Write the word embedding
  out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

# Close the files
out_v.close()
out_m.close()
```

You can find the files in your current working directory and download them. Now you can go to the Tensorflow Embedding Projector and load the two files you downloaded to see the visualization. You can search for words like `worst` and `fantastic` and see the other words closely located to these.

Wrap Up

In this lab, you were able build a simple sentiment classification model and train it on preprocessed text data. In the next lessons, you will revisit the Sarcasm Dataset you used in Week 1 and build a model to train on it.