5.

Sentiment Analysis

IMDB Dataset

- The IMDb dataset consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing), along with a simple binary target for each review indicating whether it is negative (0) or positive (1).

```
import tensorflow_datasets as tfds

raw_train_set, raw_valid_set, raw_test_set = tfds.load(
    name="imdb_reviews",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
tf.random.set_seed(42)
train_set = raw_train_set.shuffle(5000, seed=42).batch(32).prefetch(1)
valid_set = raw_valid_set.batch(32).prefetch(1)
test_set = raw_test_set.batch(32).prefetch(1)
```

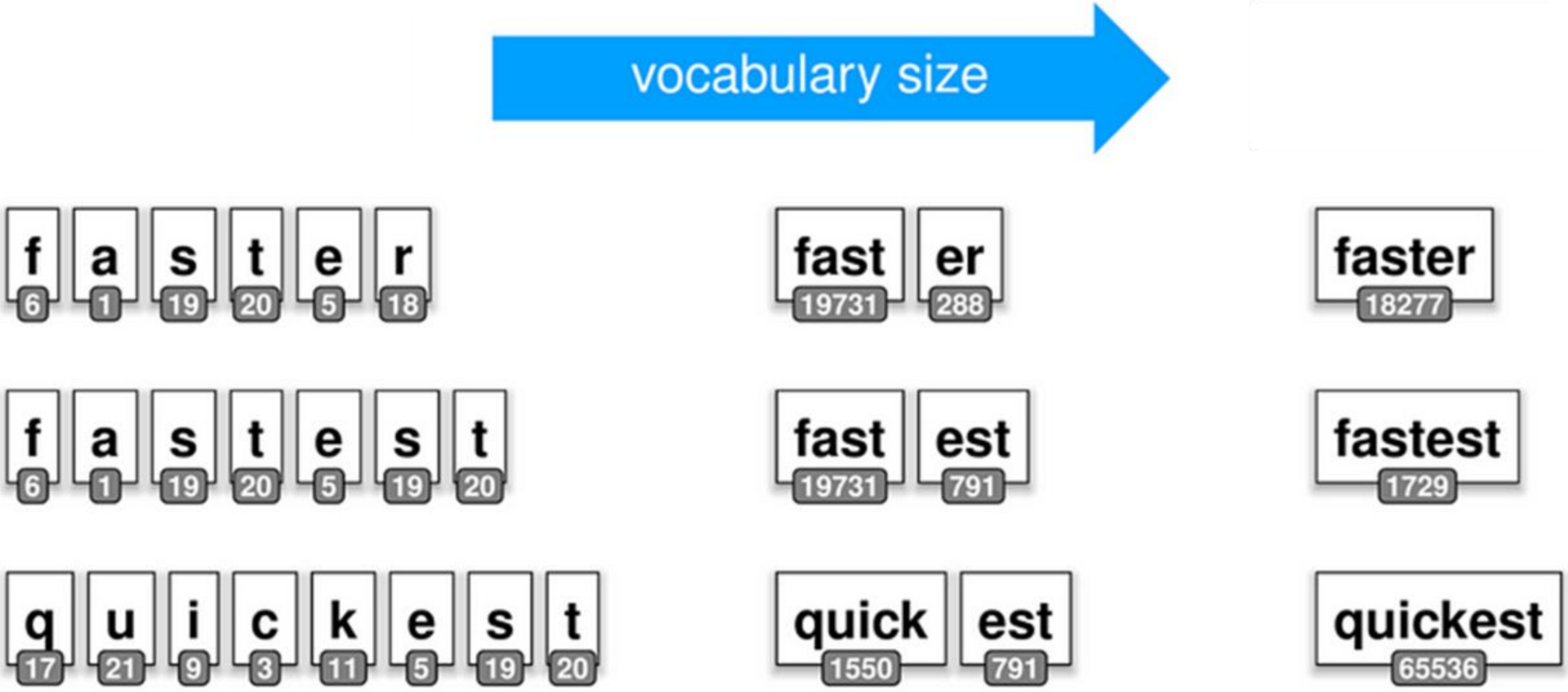
Data Sample

```
for review, label in raw_train_set.take(4):
    print(review.numpy().decode("utf-8")[:100], "[...]")
    print("Label:", label.numpy())
```

```
This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside.  [...]
Label: 0
I have been known to fall asleep during films, but this is usually due to a combination of things in [...]
Label: 0
Mann photographs the Alberta Rocky Mountains in a superb fashion, and Jimmy Stewart and Walter Brenn [...]
Label: 0
This is the kind of film for a snowy Sunday afternoon when the rest of the world can go ahead with i [...]
Label: 1
```

- To build a model for this task, we need to preprocess the text, but this time we will chop it into words instead of characters using the `TextVectorization` layer again.
- Note that it uses spaces to identify word boundaries, which will not work well in some languages. Even in English, spaces are not always the best way to tokenize text: think of “San Francisco” or “#ILoveDeepLearning”.

Tokenization



Character Level Subword Level Word Level

Preprocessing

- Create a `TextVectorization` layer and adapt it to the training set.
- We limit the vocabulary to 1,000 tokens, including the most frequent 998 words plus a padding token and a token for unknown words.
  - It's unlikely that very rare words will be important for this task.
  - Limiting the vocabulary size will reduce the number of parameters the model needs to learn.

```
vocab_size = 1000
text_vec_layer = tf.keras.layers.TextVectorization(max_tokens=vocab_size)
text_vec_layer.adapt(train_set.map(lambda reviews, labels: reviews))
```

Building the Model

- Create the model and train it:

```
embed_size = 128
tf.random.set_seed(42)
model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size),
    tf.keras.layers.GRU(128),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=2)
```

```
Epoch 1/2
704/704 [==============================] - 255s 359ms/step - loss: 0.6934 - accuracy: 0.4990 - val_loss: 0.6931 - val_accuracy: 0.5016
Epoch 2/2
704/704 [==============================] - 250s 355ms/step - loss: 0.6934 - accuracy: 0.5042 - val_loss: 0.6942 - val_accuracy: 0.5008
```

Why it didn't work

- The reviews have different lengths, so when the `TextVectorization` layer converts them to sequences of token IDs, it pads the shorter sequences using the padding token (with ID 0) to make them as long as the longest sequence in the batch.
  - Most sequences end with many padding tokens.

- Though we're using a `GRU` layer, which is much better than a `SimpleRNN` layer, its short-term memory is still not great, so when it goes through many padding tokens, it ends up forgetting what the review was about!
  - Solution 1: feed the model with batches of equal-length sentences (which also speeds up training).
  - Solution 2: make the RNN ignore the padding tokens (masking).

Masking

- Add `mask_zero=True` when creating the `Embedding` layer. This means that padding tokens (whose ID is 0) will be ignored by all downstream layers.

```
embed_size = 128
tf.random.set_seed(42)
model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size, mask_zero=True),
    tf.keras.layers.GRU(128),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=5)
```

```
Epoch 1/5
704/704 [==============================] - 303s 426ms/step - loss: 0.5296 - accuracy: 0.7234 - val_loss: 0.4045 - val_accuracy: 0.8244
Epoch 2/5
704/704 [==============================] - 295s 419ms/step - loss: 0.3702 - accuracy: 0.8418 - val_loss: 0.3390 - val_accuracy: 0.8532
```

Using Pretrained Embeddings

- We can reuse word embeddings trained on some other (very) large text corpus (e.g., Amazon reviews, available on TensorFlow Datasets), even if it is not composed of movie reviews.
  - “amazing” has the same meaning whether used about movies or anything else.
- Other embeddings would be useful for sentiment analysis even if they were trained on another task: since words like “awesome” and “amazing” have a similar meaning, they will likely cluster in the embedding space even for tasks such as predicting the next word in a sentence.
  - If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis.
- Instead of training word embeddings, we could just download and use pretrained embeddings, such as Google’s Word2vec, Stanford’s GloVe, or Facebook’s FastText.

Limitation of Pretrained Embeddings

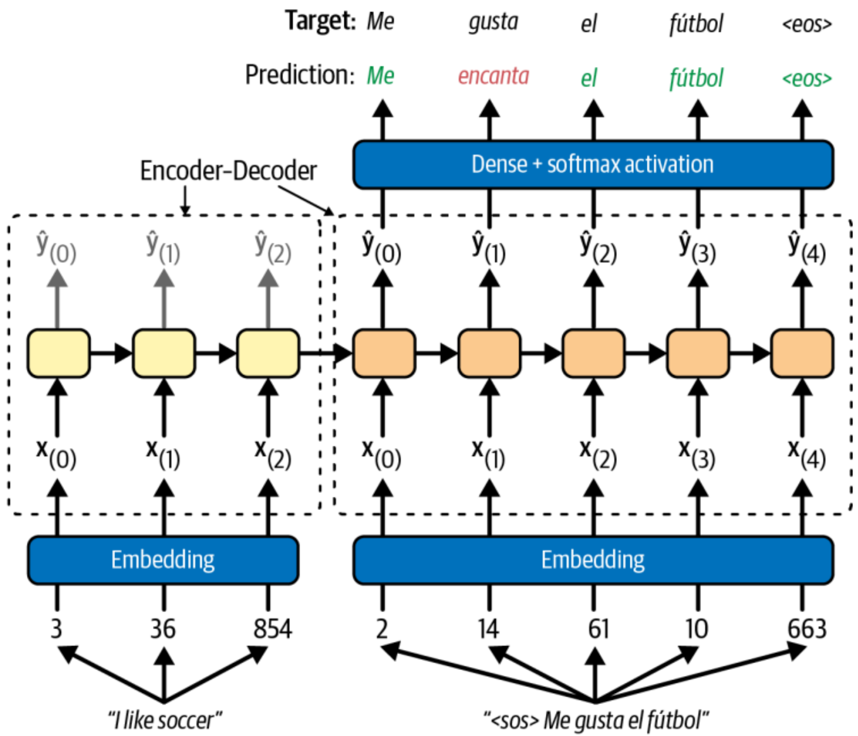
- A word has a single representation, no matter the context.
  - For example, the word "right" is encoded the same way in "left and right" and "right and wrong", even though it means two very different things.
- To address this limitation, a 2018 paper by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model.
- Predicting the word "are" from both left and right contexts:

- Instead of just using pretrained embeddings in your model, you reuse part of a pretrained bidirectional language model.

6.
Encoder-Decoder Networks

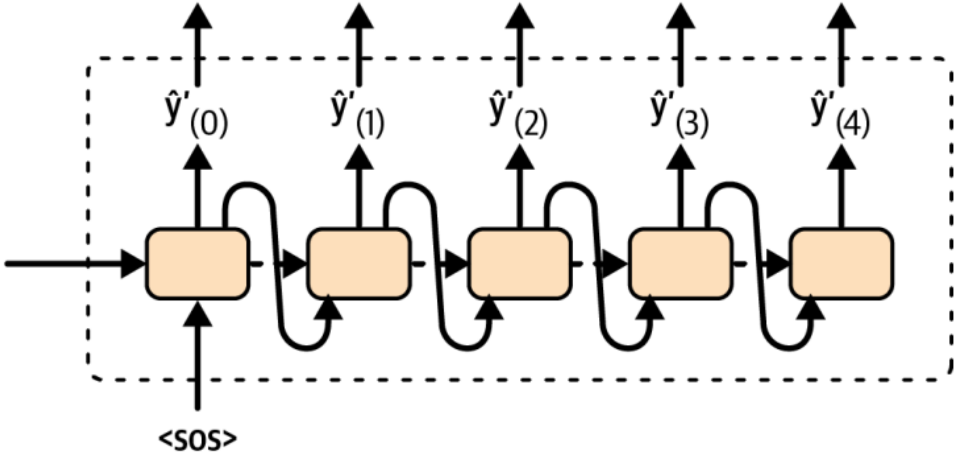
Encoder-Decoder Network

- A *neural machine translation* (NMT) model that translates English to Spanish:



Decoder at Inference Time

- At inference time (after training), you will not have the target sentence to feed to the decoder. Instead, you need to feed it the word that it has just output at the previous step.



Get the Dataset

- Download the dataset of English/Spanish sentence pairs:

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip", origin=url, cache_dir="datasets",
                               extract=True)
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text()
```

- Preprocessing the data:

```
for i in range(3):
    print(sentences_en[i], "=>", sentences_es[i])
```

```
How boring! => Qué aburrimiento!
I love sports. => Adoro el deporte.
Would you like to swap jobs? => Te gustaría que intercambiemos los trabajos?
```

TextVectorization **Layer**

- Create two `TextVectorization` layers—one per language—and adapt them to the text:

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in sentences_es])
```

```
text_vec_layer_en.get_vocabulary()[:10]
```

```
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
```

```
text_vec_layer_es.get_vocabulary()[:10]
```

```
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

Training/Validation Sets

- Create the training set and the validation set:

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[100_000:]])
```

- To build our model, we use the functional API since the model is not sequential. It requires two inputs for the encoder and the decoder:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

Encoding the Sentences

- We need to encode these sentences using the `TextVectorization` layers we prepared earlier, followed by an Embedding layer for each language, with `mask_zero=True` to ensure masking is handled.

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```

Creating the Encoder

- Create the encoder and pass it the embedded inputs:

```
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
```

- To keep things simple, we just used a single LSTM layer, but we could stack several of them.
  - We set `return_state=True` to get a reference to the layer's final state.
  - Since we're using an LSTM layer, there are actually two states: the short-term state and the long-term state.
- The layer returns these states separately, which is why we had to write `*encoder_state` to group both states in a list.
  - In Python, if you run `a, *b = [1, 2, 3, 4]`, then `a` equals `1` and `b` equals `[2, 3, 4]`.

Creating the Decoder

- Use the encoder's (double) state as the initial state of the decoder:

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

- Next, pass the decoder's outputs through a `Dense` layer with the softmax activation function to get the word probabilities for each step:

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

- We can use *sampled softmax* during training and use the normal softmax function at inference time.
  - Sampled softmax cannot be used at inference time because it requires knowing the target.

Creating the Keras Model

- Create the Keras Model, compile it, and train it:

```
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
```

- Translating a sentence is not as simple as calling `model.predict`(), because the decoder expects as input the word that was predicted at the previous time step.
  - Write a `translate` utility function that calls the model multiple times, predicting one extra word at each round.

The Model’s Performance

- The model works on short sentences:

```
translate("I like soccer")
```

```
'me gusta el fútbol'
```

- But it struggles with longer sentences:

```
translate("I like soccer and also going to the beach")
```

```
'me gusta el fútbol y a veces mismo al bus'
```

- How can you improve it?
  - Increase the training set size
  - Add more LSTM layers in both the encoder and the decoder
  - Use more sophisticated techniques, such as bidirectional recurrent layers