
Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous **Turing test**¹ in 1950, his objective was to evaluate a machine's ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.² This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords; it could pretend that it is joking or drunk, to get a pass on its weirdest answers; or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability. Can we build a machine that can read and write natural language?

A common approach for natural language tasks is to use recurrent neural networks. We will therefore continue to explore RNNs (introduced in **Chapter 15**), starting with a *character RNN*, trained to predict the next character in a sentence. This will allow us to generate some original text, and in the process we will see how to build a TensorFlow Dataset on a very long sequence. We will first use a *stateless RNN* (which learns

1 Alan Turing, "Computing Machinery and Intelligence," *Mind* 49 (1950): 433–460.

2 Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

on random portions of text at each iteration, without any information on the rest of the text), then we will build a *stateful RNN* (which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns). Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater’s feeling about the movie), this time treating sentences as sequences of words, rather than characters. Then we will show how RNNs can be used to build an Encoder–Decoder architecture capable of performing neural machine translation (NMT). For this, we will use the seq2seq API provided by the TensorFlow Addons project.

In the second part of this chapter, we will look at *attention mechanisms*. As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step. First we will see how to boost the performance of an RNN-based Encoder–Decoder architecture using attention, then we will drop RNNs altogether and look at a very successful attention-only architecture called the *Transformer*. Finally, we will take a look at some of the most important advances in NLP in 2018 and 2019, including incredibly powerful language models such as GPT-2 and BERT, both based on Transformers.

Let’s start with a simple and fun model that can write like Shakespeare (well, sort of).

Generating Shakespearean Text Using a Character RNN

In a famous [2015 blog post](#) titled “The Unreasonable Effectiveness of Recurrent Neural Networks,” Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *Char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a Char-RNN model after it was trained on all of Shakespeare’s work:

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain’d into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. Let’s look at how to build a Char-RNN, step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, let's download all of Shakespeare's work, using Keras's handy `get_file()` function and downloading the data from Andrej Karpathy's [Char-RNN project](#):

```
shakespeare_url = "https://hml.info/shakespeare" # shortcut URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Next, we must encode every character as an integer. One option is to create a custom preprocessing layer, as we did in [Chapter 13](#). But in this case, it will be simpler to use Keras's `Tokenizer` class. First we need to fit a tokenizer to the text: it will find all the characters used in the text and map each of them to a different character ID, from 1 to the number of distinct characters (it does not start at 0, so we can use that value for masking, as we will see later in this chapter):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

We set `char_level=True` to get character-level encoding rather than the default word-level encoding. Note that this tokenizer converts the text to lowercase by default (but you can set `lower=False` if you do not want that). Now the tokenizer can encode a sentence (or a list of sentences) to a list of character IDs and back, and it tells us how many distinct characters there are and the total number of characters in the text:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # number of distinct characters
>>> dataset_size = tokenizer.document_count # total number of characters
```

Let's encode the full text so each character is represented by its ID (we subtract 1 to get IDs from 0 to 38, rather than from 1 to 39):

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Before we continue, we need to split the dataset into a training set, a validation set, and a test set. We can't just shuffle all the characters in the text, so how do you split a sequential dataset?

How to Split a Sequential Dataset

It is very important to avoid any overlap between the training set, the validation set, and the test set. For example, we can take the first 90% of the text for the training set, then the next 5% for the validation set, and the final 5% for the test set. It would also

be a good idea to leave a gap between these sets to avoid the risk of a paragraph overlapping over two sets.

When dealing with time series, you would in general split across time,: for example, you might take the years 2000 to 2012 for the training set, the years 2013 to 2015 for the validation set, and the years 2016 to 2018 for the test set. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2000 to 2018, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

So, it is often safer to split across time—but this implicitly assumes that the patterns the RNN can learn in the past (in the training set) will still exist in the future. In other words, we assume that the time series is *stationary* (at least in a wide sense).³ For many time series this assumption is reasonable (e.g., chemical reactions should be fine, since the laws of chemistry don't change every day), but for many others it is not (e.g., financial markets are notoriously not stationary since patterns disappear as soon as traders spot them and start exploiting them). To make sure the time series is indeed sufficiently stationary, you can plot the model's errors on the validation set across time: if the model performs much better on the first part of the validation set than on the last part, then the time series may not be stationary enough, and you might be better off training the model on a shorter time span.

In short, splitting a time series into a training set, a validation set, and a test set is not a trivial task, and how it's done will depend strongly on the task at hand.

Now back to Shakespeare! Let's take the first 90% of the text for the training set (keeping the rest for the validation set and the test set), and create a `tf.data.Dataset` that will return each character one by one from this set:

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Chopping the Sequential Dataset into Multiple Windows

The training set now consists of a single sequence of over a million characters, so we can't just train the neural network directly on it: the RNN would be equivalent to a

³ By definition, a stationary time series's mean, variance, and *autocorrelations* (i.e., correlations between values in the time series separated by a given interval) do not change over time. This is quite restrictive; for example, it excludes time series with trends or cyclical patterns. RNNs are more tolerant in that they can learn trends and cyclical patterns.

deep net with over a million layers, and we would have a single (very long) instance to train it. Instead, we will use the dataset's `window()` method to convert this long sequence of characters into many smaller windows of text. Every instance in the dataset will be a fairly short substring of the whole text, and the RNN will be unrolled only over the length of these substrings. This is called *truncated backpropagation through time*. Let's call the `window()` method to create a dataset of short text windows:

```
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



You can try tuning `n_steps`: it is easier to train RNNs on shorter input sequences, but of course the RNN will not be able to learn any pattern longer than `n_steps`, so don't make it too small.

By default, the `window()` method creates nonoverlapping windows, but to get the largest possible training set we use `shift=1` so that the first window contains characters 0 to 100, the second contains characters 1 to 101, and so on. To ensure that all windows are exactly 101 characters long (which will allow us to create batches without having to do any padding), we set `drop_remainder=True` (otherwise the last 100 windows will contain 100 characters, 99 characters, and so on down to 1 character).

The `window()` method creates a dataset that contains windows, each of which is also represented as a dataset. It's a *nested dataset*, analogous to a list of lists. This is useful when you want to transform each window by calling its dataset methods (e.g., to shuffle them or batch them). However, we cannot use a nested dataset directly for training, as our model will expect tensors as input, not datasets. So, we must call the `flat_map()` method: it converts a nested dataset into a *flat dataset* (one that does not contain datasets). For example, suppose `{1, 2, 3}` represents a dataset containing the sequence of tensors 1, 2, and 3. If you flatten the nested dataset `{{1, 2}, {3, 4, 5, 6}}`, you get back the flat dataset `{1, 2, 3, 4, 5, 6}`. Moreover, the `flat_map()` method takes a function as an argument, which allows you to transform each dataset in the nested dataset before flattening. For example, if you pass the function `lambda ds: ds.batch(2)` to `flat_map()`, then it will transform the nested dataset `{{1, 2}, {3, 4, 5, 6}}` into the flat dataset `{[1, 2], [3, 4], [5, 6]}`: it's a dataset of tensors of size 2. With that in mind, we are ready to flatten our dataset:

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Notice that we call `batch(window_length)` on each window: since all windows have exactly that length, we will get a single tensor for each of them. Now the dataset contains consecutive windows of 101 characters each. Since Gradient Descent works best

when the instances in the training set are independent and identically distributed (see [Chapter 4](#)), we need to shuffle these windows. Then we can batch the windows and separate the inputs (the first 100 characters) from the target (the last character):

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

Figure 16-1 summarizes the dataset preparation steps discussed so far (showing windows of length 11 rather than 101, and a batch size of 3 instead of 32).

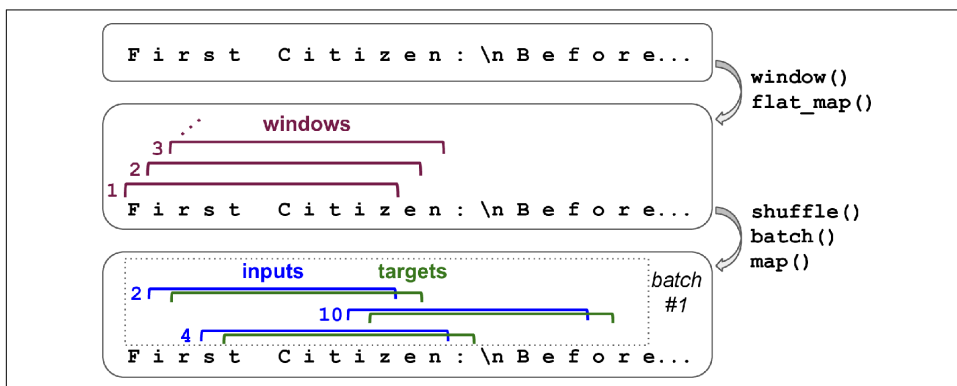


Figure 16-1. Preparing a dataset of shuffled windows

As discussed in [Chapter 13](#), categorical input features should generally be encoded, usually as one-hot vectors or as embeddings. Here, we will encode each character using a one-hot vector because there are fairly few distinct characters (only 39):

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Finally, we just need to add prefetching:

```
dataset = dataset.prefetch(1)
```

That's it! Preparing the dataset was the hardest part. Now let's create the model.

Building and Training the Char-RNN Model

To predict the next character based on the previous 100 characters, we can use an RNN with 2 GRU layers of 128 units each and 20% dropout on both the inputs (dropout) and the hidden states (recurrent_dropout). We can tweak these hyperparameters later, if needed. The output layer is a time-distributed Dense layer like we saw in [Chapter 15](#). This time this layer must have 39 units (`max_id`) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The output probabilities should sum up to 1 at each time step, so we apply the softmax activation function to the outputs of the Dense

layer. We can then compile this model, using the "sparse_categorical_crossentropy" loss and an Adam optimizer. Finally, we are ready to train the model for several epochs (this may take many hours, depending on your hardware):

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                      dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                      dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                       activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

Using the Char-RNN Model

Now we have a model that can predict the next character in text written by Shakespeare. To feed it some text, we first need to preprocess it like we did earlier, so let's create a little function for this:

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Now let's use the model to predict the next letter in some text:

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
'u'
```

Success! The model guessed right. Now let's use this model to generate new text.

Generating Fake Shakespearean Text

To generate new text using the Char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it at the end of the text, then give the extended text to the model to guess the next letter, and so on. But in practice this often leads to the same words being repeated over and over again. Instead, we can pick the next character randomly, with a probability equal to the estimated probability, using TensorFlow's `tf.random.categorical()` function. This will generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish: a temperature close to 0 will favor the high-probability characters, while a very high temperature will give all characters an equal probability. The following `next_char()` function uses this approach to pick the next character to add to the input text:

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Next, we can write a small function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

We are now ready to generate some text! Let's try with different temperatures:

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

Apparently our Shakespeare model works best at a temperature close to 1. To generate more convincing text, you could try using more GRU layers and more neurons per layer, train for longer, and add some regularization (for example, you could set `recurrent_dropout=0.3` in the GRU layers). Moreover, the model is currently incapable of learning patterns longer than `n_steps`, which is just 100 characters. You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences. Alternatively, you could use a stateful RNN.

Stateful RNN

Until now, we have used only *stateless* RNNs: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away, as it is not needed anymore. What if we told the RNN to preserve this final state after processing one training batch and use it as the initial state for the next training batch? This way the model can learn long-term patterns despite only backpropagating through short sequences. This is called a *stateful* RNN. Let's see how to build one.

First, note that a stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing we need to do to build a stateful RNN is to use sequential and nonoverlap-

ping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs). When creating the Dataset, we must therefore use `shift=n_steps` (instead of `shift=1`) when calling the `window()` method. Moreover, we must obviously *not* call the `shuffle()` method. Unfortunately, batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if we were to call `batch(32)`, then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these window where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use “batches” containing a single window:

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

Figure 16-2 summarizes the first steps.

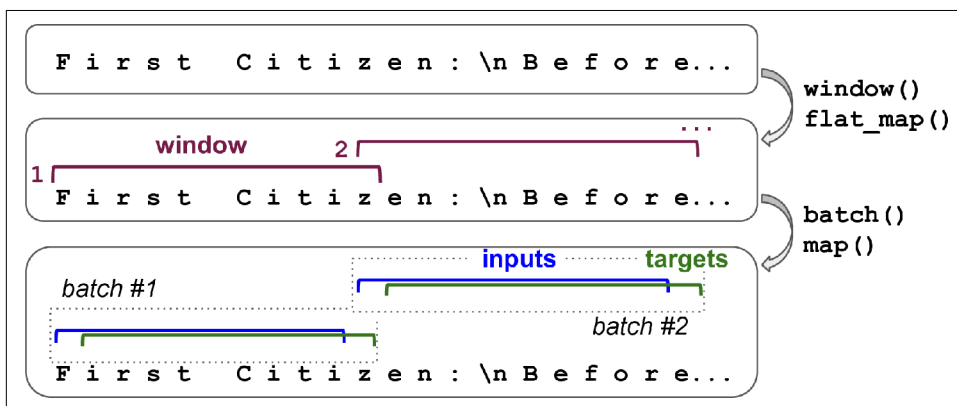


Figure 16-2. Preparing a dataset of consecutive sequence fragments for a stateful RNN

Batching is harder, but it is not impossible. For example, we could chop Shakespeare’s text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` to create proper consecutive batches, where the n^{th} input sequence in a batch starts off exactly where the n^{th} input sequence ended in the previous batch (see the notebook for the full code).

Now let's create the stateful RNN. First, we need to set `stateful=True` when creating every recurrent layer. Second, the stateful RNN needs to know the batch size (since it will preserve a state for each input sequence in the batch), so we must set the `batch_input_shape` argument in the first layer. Note that we can leave the second dimension unspecified, since the inputs could have any length:

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                       activation="softmax"))
])
```

At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small callback:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And now we can compile and fit the model (for more epochs, because each epoch is much shorter than earlier, and there is only one instance per batch):

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```



After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical *stateless* model, and copy the stateful model's weights to this model.

Now that we have built a character-level model, it's time to look at word-level models and tackle a common natural language processing task: *sentiment analysis*. In the process we will learn how to handle sequences of variable lengths using masking.

Sentiment Analysis

If MNIST is the “hello world” of computer vision, then the IMDB reviews dataset is the “hello world” of natural language processing: it consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous [Internet Movie Database](#), along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDB reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount

of time, but challenging enough to be fun and rewarding. Keras provides a simple function to load it:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Where are the movie reviews? Well, as you can see, the dataset is already preprocessed for you: `X_train` consists of a list of reviews, each of which is represented as a NumPy array of integers, where each integer represents a word. All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words). The integers 0, 1, and 2 are special: they represent the padding token, the *start-of-sequence* (SSS) token, and unknown words, respectively. If you want to visualize a review, you can decode it like this:

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
...     id_to_word[id_] = token
...
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

In a real project, you will have to preprocess the text yourself. You can do that using the same `Tokenizer` class we used earlier, but this time setting `char_level=False` (which is the default). When encoding words, it filters out a lot of characters, including most punctuation, line breaks, and tabs (but you can change this by setting the `filters` argument). Most importantly, it uses spaces to identify word boundaries. This is OK for English and many other scripts (written languages) that use spaces between words, but not all scripts use spaces this way. Chinese does not use spaces between words, Vietnamese uses spaces even within words, and languages such as German often attach multiple words together, without spaces. Even in English, spaces are not always the best way to tokenize text: think of “San Francisco” or “#ILoveDeepLearning.”

Fortunately, there are better options! The [2018 paper](#)⁴ by Taku Kudo introduced an unsupervised learning technique to tokenize and detokenize text at the subword level in a language-independent way, treating spaces like other characters. With this approach, even if your model encounters a word it has never seen before, it can still reasonably guess what it means. For example, it may never have seen the word “smartest” during training, but perhaps it learned the word “smart” and it also learned that the suffix “est” means “the most,” so it can infer the meaning of

⁴ Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” arXiv preprint arXiv:1804.10959 (2018).

“smartest.” Google’s *SentencePiece* project provides an open source implementation, described in a [paper](#)⁵ by Taku Kudo and John Richardson.

Another option was proposed in an earlier [paper](#)⁶ by Rico Sennrich et al. that explored other ways of creating subword encodings (e.g., using *byte pair encoding*). Last but not least, the TensorFlow team released the *TF.Text* library in June 2019, which implements various tokenization strategies, including *WordPiece*⁷ (a variant of byte pair encoding).

If you want to deploy your model to a mobile device or a web browser, and you don’t want to have to write a different preprocessing function every time, then you will want to handle preprocessing using only TensorFlow operations, so it can be included in the model itself. Let’s see how. First, let’s load the original IMDB reviews, as text (byte strings), using TensorFlow Datasets (introduced in [Chapter 13](#)):

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Next, let’s write the preprocessing function:

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"^[a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

It starts by truncating the reviews, keeping only the first 300 characters of each: this will speed up training, and it won’t impact performance too much because you can generally tell whether a review is positive or not in the first sentence or two. Then it uses *regular expressions* to replace `
` tags with spaces, and to replace any characters other than letters and quotes with spaces. For example, the text "Well, I can't
" will become "Well I can't". Finally, the `preprocess()` function splits the reviews by the spaces, which returns a ragged tensor, and it converts this ragged tensor to a dense tensor, padding all reviews with the padding token "`<pad>`" so that they all have the same length.

5 Taku Kudo and John Richardson, “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing,” arXiv preprint arXiv:1808.06226 (2018).

6 Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units,” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.

7 Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation,” arXiv preprint arXiv:1609.08144 (2016).

Next, we need to construct the vocabulary. This requires going through the whole training set once, applying our `preprocess()` function, and using a `Counter` to count the number of occurrences of each word:

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Let's look at the three most common words:

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Great! We probably don't need our model to know all the words in the dictionary to get good performance, though, so let's truncate the vocabulary, keeping only the 10,000 most common words:

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Now we need to add a preprocessing step to replace each word with its ID (i.e., its index in the vocabulary). Just like we did in [Chapter 13](#), we will create a lookup table for this, using 1,000 out-of-vocabulary (oov) buckets:

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

We can then use this table to look up the IDs of a few words:

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,  12,  11, 10054]])>
```

Note that the words “this,” “movie,” and “was” were found in the table, so their IDs are lower than 10,000, while the word “faaaaaantastic” was not found, so it was mapped to one of the oov buckets, with an ID greater than or equal to 10,000.



TF Transform (introduced in [Chapter 13](#)) provides some useful functions to handle such vocabularies. For example, check out the `tft.compute_and_apply_vocabulary()` function: it will go through the dataset to find all distinct words and build the vocabulary, and it will generate the TF operations required to encode each word using this vocabulary.

Now we are ready to create the final training set. We batch the reviews, then convert them to short sequences of words using the `preprocess()` function, then encode

these words using a simple `encode_words()` function that uses the table we just built, and finally prefetch the next batch:

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

At last we can create the model and train it:

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

The first layer is an `Embedding` layer, which will convert word IDs into embeddings (introduced in [Chapter 13](#)). The embedding matrix needs to have one row per word ID (`vocab_size + num_oov_buckets`) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune). Whereas the inputs of the model will be 2D tensors of shape *[batch size, time steps]*, the output of the `Embedding` layer will be a 3D tensor of shape *[batch size, time steps, embedding size]*.

The rest of the model is fairly straightforward: it is composed of two `GRU` layers, with the second one returning only the output of the last time step. The output layer is just a single neuron using the sigmoid activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie. We then compile the model quite simply, and we fit it on the dataset we prepared earlier, for a few epochs.

Masking

As it stands, the model will need to learn that the padding tokens should be ignored. But we already know that! Why don't we tell the model to ignore the padding tokens, so that it can focus on the data that actually matters? It's actually trivial: simply add

`mask_zero=True` when creating the Embedding layer. This means that padding tokens (whose ID is 0)⁸ will be ignored by all downstream layers. That's all!

The way this works is that the Embedding layer creates a *mask tensor* equal to `K.not_equal(inputs, 0)` (where `K = keras.backend`): it is a Boolean tensor with the same shape as the inputs, and it is equal to `False` anywhere the word IDs are 0, or `True` otherwise. This mask tensor is then automatically propagated by the model to all subsequent layers, as long as the time dimension is preserved. So in this example, both GRU layers will receive this mask automatically, but since the second GRU layer does not return sequences (it only returns the output of the last time step), the mask will not be transmitted to the Dense layer. Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is `False`). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step. If the mask propagates all the way to the output (in models that output sequences, which is not the case in this example), then it will be applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0).



The LSTM and GRU layers have an optimized implementation for GPUs, based on Nvidia's cuDNN library. However, this implementation does not support masking. If your model uses a mask, then these layers will fall back to the (much slower) default implementation. Note that the optimized implementation also requires you to use the default values for several hyperparameters: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias`, and `reset_after`.

All layers that receive the mask must support masking (or else an exception will be raised). This includes all recurrent layers, as well as the `TimeDistributed` layer and a few other layers. Any layer that supports masking must have a `supports_masking` attribute equal to `True`. If you want to implement your own custom layer with masking support, you should add a `mask` argument to the `call()` method (and obviously make the method use the mask somehow). Additionally, you should set `self.supports_masking = True` in the constructor. If your layer does not start with an Embedding layer, you may use the `keras.layers.Masking` layer instead: it sets the mask to `K.any(K.not_equal(inputs, 0), axis=-1)`, meaning that time steps where the last dimension is full of zeros will be masked out in subsequent layers (again, as long as the time dimension exists).

⁸ Their ID is 0 only because they are the most frequent “words” in the dataset. It would probably be a good idea to ensure that the padding tokens are always encoded as 0, even if they are not the most frequent.

Using masking layers and automatic mask propagation works best for simple `Sequential` models. It will not always work for more complex models, such as when you need to mix `Conv1D` layers with recurrent layers. In such cases, you will need to explicitly compute the mask and pass it to the appropriate layers, using either the Functional API or the Subclassing API. For example, the following model is identical to the previous model, except it is built using the Functional API and handles masking manually:

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

After training for a few epochs, this model will become quite good at judging whether a review is positive or not. If you use the `TensorBoard()` callback, you can visualize the embeddings in TensorBoard as they are being learned: it is fascinating to see words like “awesome” and “amazing” gradually cluster on one side of the embedding space, while words like “awful” and “terrible” cluster on the other side. Some words are not as positive as you might expect (at least with this model), such as the word “good,” presumably because many negative reviews contain the phrase “not good.” It’s impressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews. Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately we don’t, but perhaps we can reuse word embeddings trained on some other large text corpus (e.g., Wikipedia articles), even if it is not composed of movie reviews? After all, the word “amazing” generally has the same meaning whether you use it to talk about movies or anything else. Moreover, perhaps 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 other tasks (e.g., 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. So instead of using so many parameters to learn word embeddings, let’s see if we can’t just reuse pretrained embeddings.

Reusing Pretrained Embeddings

The TensorFlow Hub project makes it easy to reuse pretrained model components in your own models. These model components are called *modules*. Simply browse the [TF Hub repository](#), find the one you need, and copy the code example into your project, and the module will be automatically downloaded, along with its pretrained weights, and included in your model. Easy!

For example, let's use the `nnlm-en-dim50` sentence embedding module, version 1, in our sentiment analysis model:

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

The `hub.KerasLayer` layer downloads the module from the given URL. This particular module is a *sentence encoder*: it takes strings as input and encodes each one as a single vector (in this case, a 50-dimensional vector). Internally, it parses the string (splitting words on spaces) and embeds each word using an embedding matrix that was pretrained on a huge corpus: the Google News 7B corpus (seven billion words long!). Then it computes the mean of all the word embeddings, and the result is the sentence embedding.⁹ We can then add two simple `Dense` layers to create a good sentiment analysis model. By default, a `hub.KerasLayer` is not trainable, but you can set `trainable=True` when creating it to change that so that you can fine-tune it for your task.



Not all TF Hub modules support TensorFlow 2, so make sure you choose a module that does.

Next, we can just load the IMDB reviews dataset—no need to preprocess it (except for batching and prefetching)—and directly train the model:

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

Note that the last part of the TF Hub module URL specified that we wanted version 1 of the model. This versioning ensures that if a new module version is released, it will not break our model. Conveniently, if you just enter this URL in a web browser, you

⁹ To be precise, the sentence embedding is equal to the mean word embedding multiplied by the square root of the number of words in the sentence. This compensates for the fact that the mean of n vectors gets shorter as n grows.

will get the documentation for this module. By default, TF Hub will cache the downloaded files into the local system's temporary directory. You may prefer to download them into a more permanent directory to avoid having to download them again after every system cleanup. To do that, set the `TFHUB_CACHE_DIR` environment variable to the directory of your choice (e.g., `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

So far, we have looked at time series, text generation using Char-RNN, and sentiment analysis using word-level RNN models, training our own word embeddings or reusing pretrained embeddings. Let's now look at another important NLP task: *neural machine translation* (NMT), first using a pure Encoder–Decoder model, then improving it with attention mechanisms, and finally looking the extraordinary Transformer architecture.

An Encoder–Decoder Network for Neural Machine Translation

Let's take a look at a simple **neural machine translation model**¹⁰ that will translate English sentences to French (see [Figure 16-3](#)).

In short, the English sentences are fed to the encoder, and the decoder outputs the French translations. Note that the French translations are also used as inputs to the decoder, but shifted back by one step. In other words, the decoder is given as input the word that it *should* have output at the previous step (regardless of what it actually output). For the very first word, it is given the start-of-sequence (SOS) token. The decoder is expected to end the sentence with an end-of-sequence (EOS) token.

Note that the English sentences are reversed before they are fed to the encoder. For example, “I drink milk” is reversed to “milk drink I.” This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that's generally the first thing that the decoder needs to translate.

Each word is initially represented by its ID (e.g., 288 for the word “milk”). Next, an embedding layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.

¹⁰ Ilya Sutskever et al., “Sequence to Sequence Learning with Neural Networks,” arXiv preprint arXiv:1409.3215 (2014).

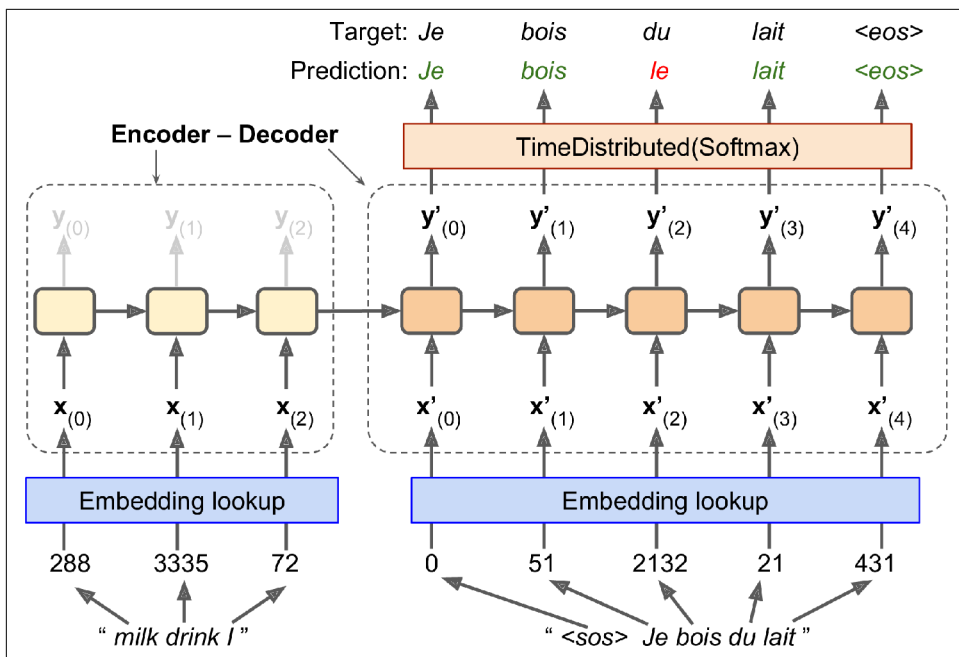


Figure 16-3. A simple machine translation model

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities. For example, at the first step the word "Je" may have a probability of 20%, "Tu" may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, so you can train the model using the "sparse_categorical_crossentropy" loss, much like we did in the Char-RNN model.

Note that at inference time (after training), you will not have the target sentence to feed to the decoder. Instead, simply feed the decoder the word that it output at the previous step, as shown in Figure 16-4 (this will require an embedding lookup that is not shown in the diagram).

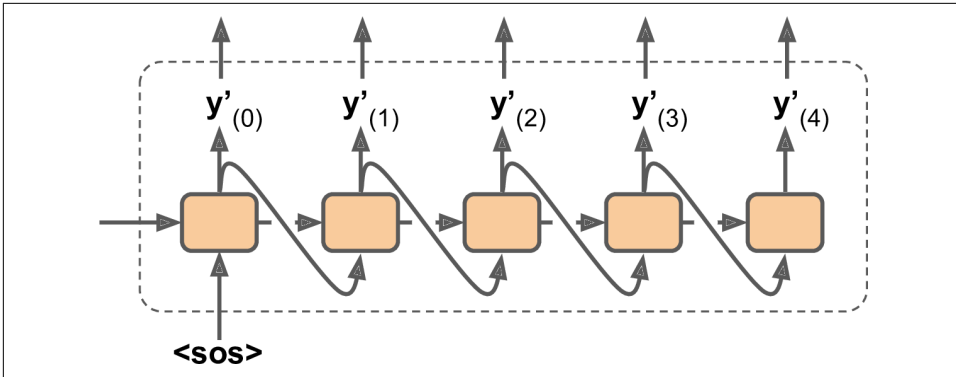


Figure 16-4. Feeding the previous output word as input at inference time

OK, now you have the big picture. Still, there are a few more details to handle if you implement this model:

- So far we have assumed that all input sequences (to the encoder and to the decoder) have a constant length. But obviously sentence lengths vary. Since regular tensors have fixed shapes, they can only contain sentences of the same length. You can use masking to handle this, as discussed earlier. However, if the sentences have very different lengths, you can't just crop them like we did for sentiment analysis (because we want full translations, not cropped translations). Instead, group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on), using padding for the shorter sequences to ensure all sentences in a bucket have the same length (check out the `tf.data.experimental.bucket_by_sequence_length()` function for this). For example, "I drink milk" becomes "<pad> <pad> <pad> milk drink I."
- We want to ignore any output past the EOS token, so these tokens should not contribute to the loss (they must be masked out). For example, if the model outputs "Je bois du lait <eos> oui," the loss for the last word should be ignored.
- When the output vocabulary is large (which is the case here), outputting a probability for each and every possible word would be terribly slow. If the target vocabulary contains, say, 50,000 French words, then the decoder would output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This *sampled softmax* technique was **introduced in**

2015 by Sébastien Jean et al.¹¹ In TensorFlow you can use the `tf.nn.sampled_softmax_loss()` function for this 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).

The TensorFlow Addons project includes many sequence-to-sequence tools to let you easily build production-ready Encoder–Decoders. For example, the following code creates a basic Encoder–Decoder model, similar to the one represented in Figure 16-3:

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                output_layer=output_layer)

final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                    outputs=[Y_proba])
```

The code is mostly self-explanatory, but there are a few points to note. First, we set `return_state=True` when creating the LSTM layer so that we can get its final hidden state and pass it to the decoder. Since we are using an LSTM cell, it actually returns two hidden states (short term and long term). The `TrainingSampler` is one of several samplers available in TensorFlow Addons: their role is to tell the decoder at each step what it should pretend the previous output was. During inference, this should be the

11 Sébastien Jean et al., “On Using Very Large Target Vocabulary for Neural Machine Translation,” *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.

embedding of the token that was actually output. During training, it should be the embedding of the previous target token: this is why we used the `TrainingSampler`. In practice, it is often a good idea to start training with the embedding of the target of the previous time step and gradually transition to using the embedding of the actual token that was output at the previous step. This idea was introduced in a [2015 paper](#)¹² by Samy Bengio et al. The `ScheduledEmbeddingTrainingSampler` will randomly choose between the target or the actual output, with a probability that you can gradually change during training.

Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is “causal,” meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, but for many NLP tasks, such as Neural Machine Translation, it is often preferable to look ahead at the next words before encoding a given word. For example, consider the phrases “the Queen of the United Kingdom,” “the queen of hearts,” and “the queen bee”: to properly encode the word “queen,” you need to look ahead. To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then simply combine their outputs at each time step, typically by concatenating them. This is called a *bidirectional recurrent layer* (see [Figure 16-5](#)).

To implement a bidirectional recurrent layer in Keras, wrap a recurrent layer in a `keras.layers.Bidirectional` layer. For example, the following code creates a bidirectional GRU layer:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```



The `Bidirectional` layer will create a clone of the GRU layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the GRU layer has 10 units, the `Bidirectional` layer will output 20 values per time step.

¹² Samy Bengio et al., “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks,” arXiv preprint arXiv:1506.03099 (2015).

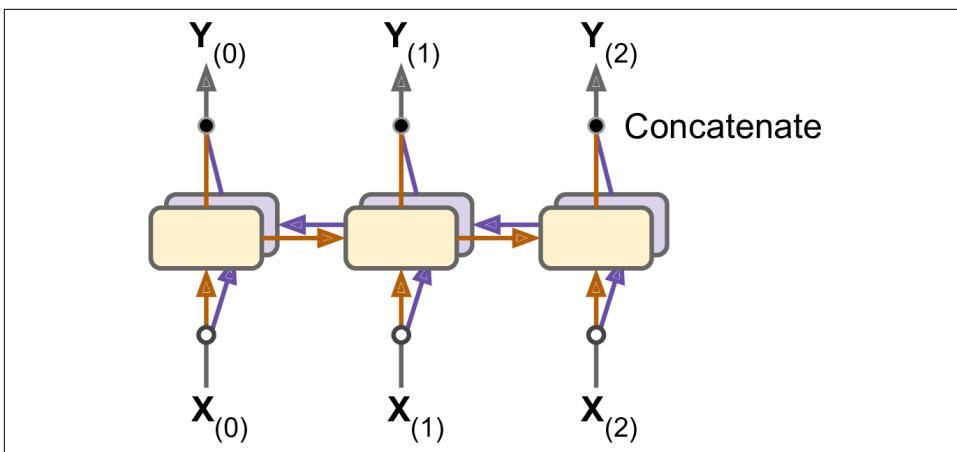


Figure 16-5. A bidirectional recurrent layer

Beam Search

Suppose you train an Encoder–Decoder model, and use it to translate the French sentence “Comment vas-tu?” to English. You are hoping that it will output the proper translation (“How are you?”), but unfortunately it outputs “How will you?” Looking at the training set, you notice many sentences such as “Comment vas-tu jouer?” which translates to “How will you play?” So it wasn’t absurd for the model to output “How will” after seeing “Comment vas.” Unfortunately, in this case it was a mistake, and the model could not go back and fix it, so it tried to complete the sentence as best it could. By greedily outputting the most likely word at every step, it ended up with a suboptimal translation. How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the k most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the k most likely sentences. The parameter k is called the *beam width*.

For example, suppose you use the model to translate the sentence “Comment vas-tu?” using beam search with a beam width of 3. At the first decoder step, the model will output an estimated probability for each possible word. Suppose the top three words are “How” (75% estimated probability), “What” (3%), and “You” (1%). That’s our short list so far. Next, we create three copies of our model and use them to find the next word for each sentence. Each model will output one estimated probability per word in the vocabulary. The first model will try to find the next word in the sentence “How,” and perhaps it will output a probability of 36% for the word “will,” 32% for the word “are,” 16% for the word “do,” and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “How.” The second model will try to complete the sentence “What”; it might output a conditional probability of 50% for

the word “are,” and so on. Assuming the vocabulary has 10,000 words, each model will output 10,000 probabilities.

Next, we compute the probabilities of each of the 30,000 two-word sentences that these models considered ($3 \times 10,000$). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “How” was 75%, while the estimated conditional probability of the word “will” (given that the first word is “How”) was 36%, so the estimated probability of the sentence “How will” is $75\% \times 36\% = 27\%$. After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. Perhaps they all start with the word “How”: “How will” (27%), “How are” (24%), and “How do” (12%). Right now, the sentence “How will” is winning, but “How are” has not been eliminated.

Then we repeat the same process: we use three models to predict the next word in each of these three sentences, and we compute the probabilities of all 30,000 three-word sentences we considered. Perhaps the top three are now “How are you” (10%), “How do you” (8%), and “How will you” (2%). At the next step we may get “How do you do” (7%), “How are you <eos>” (6%), and “How are you doing” (3%). Notice that “How will” was eliminated, and we now have three perfectly reasonable translations. We boosted our Encoder–Decoder model’s performance without any extra training, simply by using it more wisely.

You can implement beam search fairly easily using TensorFlow Addons:

```
beam_width = 10
decoder = tf.nn.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tf.nn.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

We first create a `BeamSearchDecoder`, which wraps all the decoder clones (in this case 10 clones). Then we create one copy of the encoder’s final state for each decoder clone, and we pass these states to the decoder, along with the start and end tokens.

With all this, you can get good translations for fairly short sentences (especially if you use pretrained word embeddings). Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word “milk” to its translation “lait” in [Figure 16-3](#): it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

This was the core idea in a groundbreaking [2014 paper](#)¹³ by Dzmitry Bahdanau et al. They introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “lait,” it will focus its attention on the word “milk.” This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words).¹⁴

[Figure 16-6](#) shows this model’s architecture (slightly simplified, as we will see). On the left, you have the encoder and the decoder. Instead of just sending the encoder’s final hidden state to the decoder (which is still done, although it is not shown in the figure), we now send all of its outputs to the decoder. At each time step, the decoder’s memory cell computes a weighted sum of all these encoder outputs: this determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to word number 2 (“milk”) than to the other two words, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

13 Dzmitry Bahdanau et al., “Neural Machine Translation by Jointly Learning to Align and Translate,” arXiv pre-print arXiv:1409.0473 (2014).

14 The most common metric used in NMT is the BiLingual Evaluation Understudy (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of n -grams (sequences of n words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced n -grams in the target translations.

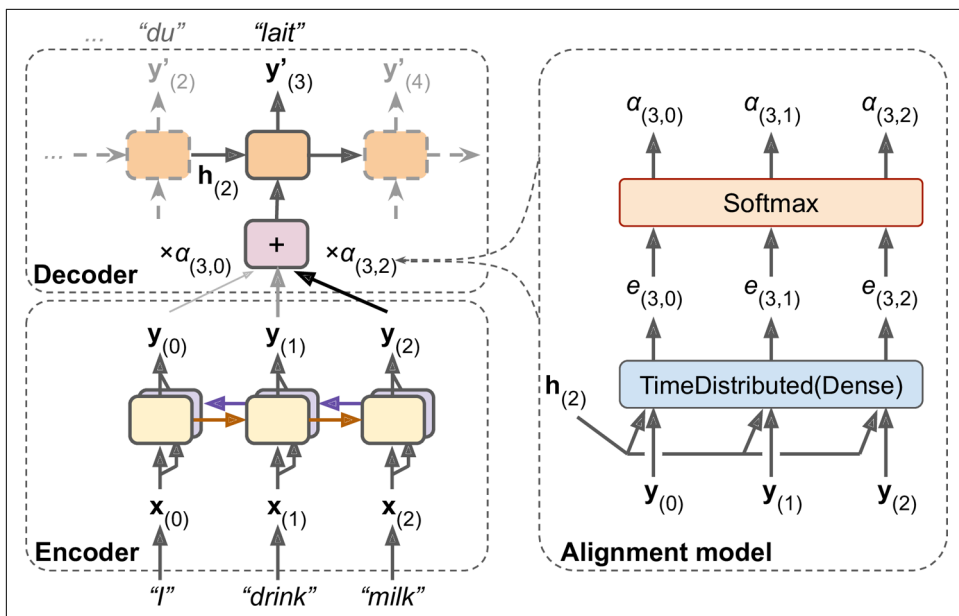


Figure 16-6. Neural machine translation using an Encoder–Decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? It's actually pretty simple: they are generated by a type of small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the Encoder–Decoder model. This alignment model is illustrated on the righthand side of [Figure 16-6](#). It starts with a time-distributed Dense layer¹⁵ with a single neuron, which receives as input all the encoder outputs, concatenated with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$): this score measures how well each output is aligned with the decoder's previous hidden state. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1 (since the softmax layer is not time-distributed). This particular attention mechanism is called *Bahdanau attention* (named after the paper's first author). Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

15 Recall that a time-distributed Dense layer is equivalent to a regular Dense layer that you apply independently at each time step (only much faster).



If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism was proposed shortly after, in a [2015 paper](#)¹⁶ by Minh-Thang Luong et al. Because the goal of the attention mechanism is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the *dot product* (see [Chapter 4](#)) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it much faster. For this to be possible, both vectors must have the same dimensionality. This is called *Luong attention* (again, after the paper's first author), or sometimes *multiplicative attention*. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification they proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$) rather than $\mathbf{h}_{(t-1)}$), then to use the output of the attention mechanism (noted $\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions (rather than using it to compute the decoder's current hidden state). They also proposed a variant of the dot product mechanism where the encoder outputs first go through a linear transformation (i.e., a time-distributed Dense layer without a bias term) before the dot products are computed. This is called the "general" dot product approach. They compared both dot product approaches to the concatenative attention mechanism (adding a rescaling parameter vector \mathbf{v}), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in [Equation 16-1](#).

¹⁶ Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

Equation 16-1. Attention mechanisms

$$\begin{aligned}\tilde{\mathbf{h}}_{(t)} &= \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)} \\ \text{with } \alpha_{(t,i)} &= \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})} \\ \text{and } e_{(t,i)} &= \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & \text{dot} \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & \text{general} \\ \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{concat} \end{cases}\end{aligned}$$

Here is how you can add Luong attention to an Encoder–Decoder model using TensorFlow Addons:

```
attention_mechanism = tf.nn.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tf.nn.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

We simply wrap the decoder cell in an `AttentionWrapper`, and we provide the desired attention mechanism (Luong attention in this example).

Visual Attention

Attention mechanisms are now used for a variety of purposes. One of their first applications beyond NMT was in generating image captions using **visual attention**.¹⁷ a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image. For example, in **Figure 16-7**, the model generated the caption “A woman is throwing a frisbee in a park,” and you can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”: clearly, most of its attention was focused on the frisbee.

¹⁷ Kelvin Xu et al., “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,” *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.



Figure 16-7. Visual attention: an input image (left) and the model's focus before producing the word “frisbee” (right)¹⁸

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake: for example, if an image of a dog walking in the snow is labeled as “a wolf walking in the snow,” then you can go back and check what the model focused on when it output the word “wolf.” You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there’s a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great [2016 paper](#)¹⁹ by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier’s prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement (think of a system deciding whether or not it should grant you a loan).

¹⁸ This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.

¹⁹ Marco Tulio Ribeiro et al., “Why Should I Trust You?: Explaining the Predictions of Any Classifier,” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

Attention mechanisms are so powerful that you can actually build state-of-the-art models using only attention mechanisms.

Attention Is All You Need: The Transformer Architecture

In a groundbreaking [2017 paper](#),²⁰ a team of Google researchers suggested that “Attention Is All You Need.” They managed to create an architecture called the *Transformer*, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers,²¹ just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). As an extra bonus, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

The Transformer architecture is represented in [Figure 16-8](#).

20 Ashish Vaswani et al., “Attention Is All You Need,” *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

21 Since the Transformer uses time-distributed Dense layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

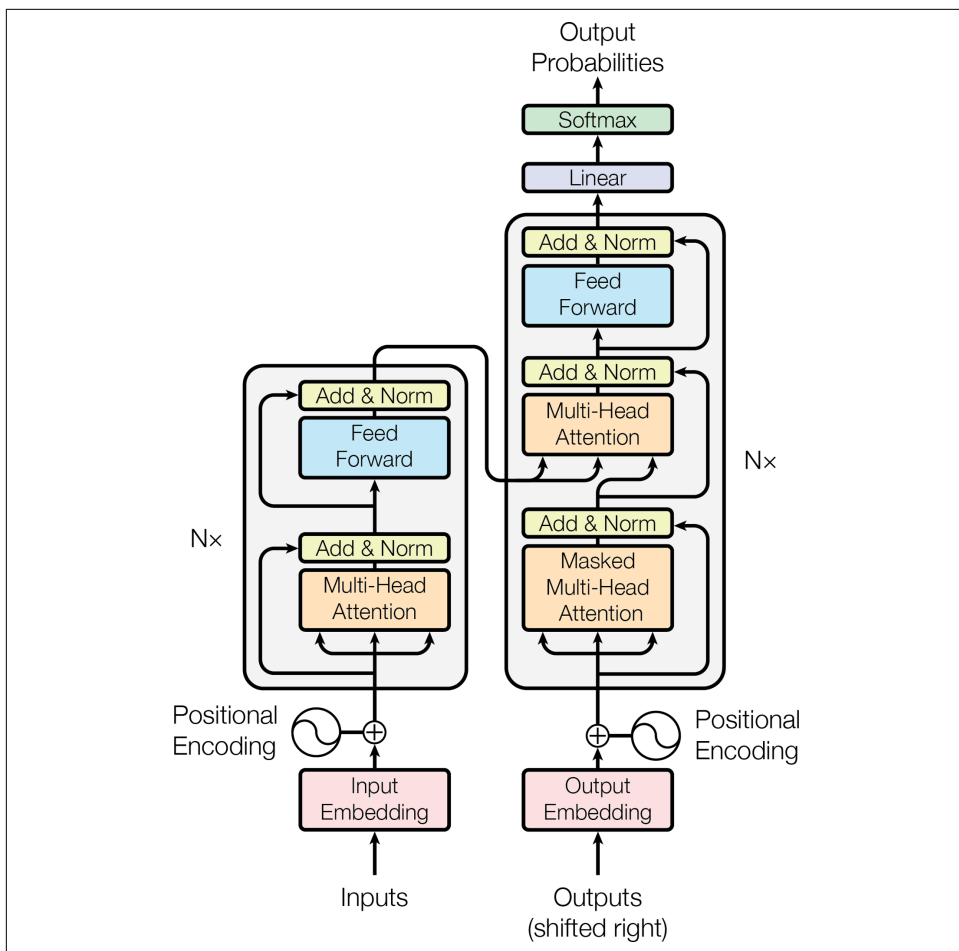


Figure 16-8. The Transformer architecture²²

Let's walk through this figure:

- The lefthand part is the encoder. Just like earlier, it takes as input a batch of sentences represented as sequences of word IDs (the input shape is $[batch\ size, max\ input\ sentence\ length]$), and it encodes each word into a 512-dimensional representation (so the encoder's output shape is $[batch\ size, max\ input\ sentence\ length, 512]$). Note that the top part of the encoder is stacked N times (in the paper, $N = 6$).

²² This is figure 1 from the paper, reproduced with the kind authorization of the authors.

- The righthand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning). It also receives the outputs of the encoder (i.e., the arrows coming from the left side). Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels. Just like earlier, the decoder outputs a probability for each possible next word, at each time step (its output shape is $[batch\ size, max\ output\ sentence\ length, vocabulary\ length]$).
- During inference, the decoder cannot be fed targets, so we feed it the previously output words (starting with a start-of-sequence token). So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).
- Looking more closely, you can see that you are already familiar with most components: there are two embedding layers, $5 \times N$ skip connections, each of them followed by a layer normalization layer, $2 \times N$ “Feed Forward” modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. All of these layers are time-distributed, so each word is treated independently of all the others. But how can we translate a sentence by only looking at one word at a time? Well, that's where the new components come in:
 - The encoder's *Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word “Queen” in the sentence “They welcomed the Queen of the United Kingdom” will depend on all the words in the sentence, but it will probably pay more attention to the words “United” and “Kingdom” than to the words “They” or “welcomed.” This attention mechanism is called *self-attention* (the sentence is paying attention to itself). We will discuss exactly how it works shortly. The decoder's *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it. Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word “Queen” in the input sentence when it is about to output this word's translation.
 - The *positional embeddings* are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence. The n^{th} positional embedding is added to the word embedding of the n^{th} word in each sentence. This gives the model access to each word's position, which is needed because the Multi-Head Attention layers do not consider the order or the position of the words; they only look at their relationships. Since all the other layers are

time-distributed, they have no way of knowing the position of each word (either relative or absolute). Obviously, the relative and absolute word positions are important, so we need to give this information to the Transformer somehow, and positional embeddings are a good way to do this.

Let's look a bit closer at both these novel components of the Transformer architecture, starting with the positional embeddings.

Positional embeddings

A positional embedding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional embedding is simply added to the word embedding of the i^{th} word in the sentence. These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the sine and cosine functions of different frequencies. The positional embedding matrix \mathbf{P} is defined in Equation 16-2 and represented at the bottom of Figure 16-9 (transposed), where $P_{p,i}$ is the i^{th} component of the embedding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional embeddings

$$P_{p,2i} = \sin(p/10000^{2i/d})$$

$$P_{p,2i+1} = \cos(p/10000^{2i/d})$$

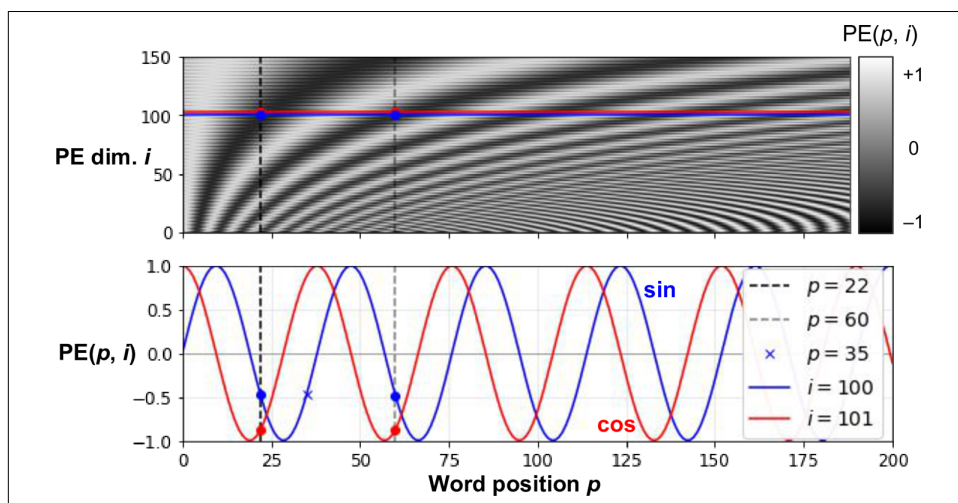


Figure 16-9. Sine/cosine positional embedding matrix (transposed, top) with a focus on two values of i (bottom)

This solution gives the same performance as learned positional embeddings do, but it can extend to arbitrarily long sentences, which is why it's favored. After the positional embeddings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional embedding for each position (e.g., the positional embedding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the bottom left of [Figure 16-9](#), and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional embedding values in the embedding dimensions $i = 100$ and $i = 101$, as you can see in [Figure 16-9](#). This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 25$ and $p = 35$ (marked by a cross).

There is no `PositionalEmbedding` layer in TensorFlow, but it is easy to create one. For efficiency reasons, we precompute the positional embedding matrix in the constructor (so we need to know the maximum sentence length, `max_steps`, and the number of dimensions for each word representation, `max_dims`). Then the `call()` method crops this embedding matrix to the size of the inputs, and it adds it to the inputs. Since we added an extra first dimension of size 1 when creating the positional embedding matrix, the rules of broadcasting will ensure that the matrix gets added to every sentence in the inputs:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Then we can create the first layers of the Transformer:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Now let's look deeper into the heart of the Transformer model: the Multi-Head Attention layer.

Multi-Head Attention

To understand how a Multi-Head Attention layer works, we must first understand the *Scaled Dot-Product Attention* layer, which it is based on. Let's suppose the encoder analyzed the input sentence "They played chess," and it managed to understand that the word "They" is the subject and the word "played" is the verb, so it encoded this information in the representations of these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it's as if the encoder created a dictionary {"subject": "They", "verb": "played", ...} and the decoder wanted to look up the value that corresponds to the key "verb." However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the *query*) will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1. Then the model can compute a weighted sum of the corresponding values, so if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played." In short, you can think of this whole process as a differentiable dictionary lookup. The similarity measure used by the Transformer is just the dot product, like in Luong attention. In fact, the equation is the same as for Luong attention, except for a scaling factor. The equation is shown in [Equation 16-3](#), in a vectorized form.

Equation 16-3. Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_{\text{keys}}}}\right)\mathbf{V}$$

In this equation:

- \mathbf{Q} is a matrix containing one row per query. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- \mathbf{K} is a matrix containing one row per key. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.

- \mathbf{V} is a matrix containing one row per value. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of each value.
- The shape of $\mathbf{Q} \mathbf{K}^T$ is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).
- The scaling factor scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the Masked Multi-Head Attention layer.

In the encoder, this equation is applied to every input sentence in the batch, with \mathbf{Q} , \mathbf{K} , and \mathbf{V} all equal to the list of words in the input sentence (so each word in the sentence will be compared to every word in the same sentence, including itself). Similarly, in the decoder's masked attention layer, the equation will be applied to every target sentence in the batch, with \mathbf{Q} , \mathbf{K} , and \mathbf{V} all equal to the list of words in the target sentence, but this time using a mask to prevent any word from comparing itself to words located after it (at inference time the decoder will only have access to the words it already output, not to future words, so during training we must mask out future output tokens). In the upper attention layer of the decoder, the keys \mathbf{K} and values \mathbf{V} are simply the list of word encodings produced by the encoder, and the queries \mathbf{Q} are the list of word encodings produced by the decoder.

The `keras.layers.Attention` layer implements Scaled Dot-Product Attention, efficiently applying Equation 16-3 to multiple sentences in a batch. Its inputs are just like \mathbf{Q} , \mathbf{K} , and \mathbf{V} , except with an extra batch dimension (the first dimension).



In TensorFlow, if A and B are tensors with more than two dimensions—say, of shape $[2, 3, 4, 5]$ and $[2, 3, 5, 6]$ respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in A will be multiplied by the matrix at the i^{th} row and j^{th} column in B . Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape $[2, 3, 4, 6]$.

If we ignore the skip connections, the layer normalization layers, the Feed Forward blocks, and the fact that this is Scaled Dot-Product Attention, not exactly Multi-Head Attention, then the rest of the Transformer model can be implemented like this:

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

The `use_scale=True` argument creates an additional parameter that lets the layer learn how to properly downscale the similarity scores. This is a bit different from the Transformer model, which always downscales the similarity scores by the same factor ($\sqrt{d_{\text{keys}}}$). The `causal=True` argument when creating the second attention layer ensures that each output token only attends to previous output tokens, not future ones.

Now it's time to look at the final piece of the puzzle: what is a Multi-Head Attention layer? Its architecture is shown in [Figure 16-10](#).

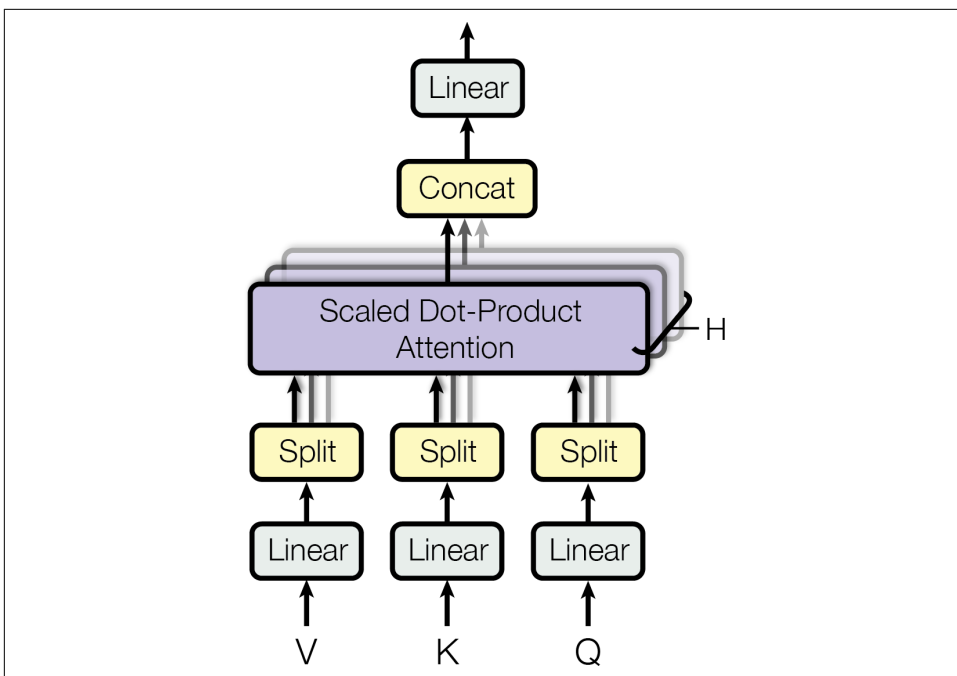


Figure 16-10. Multi-Head Attention layer architecture²³

As you can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed Dense layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed). But why? What is the intuition behind this architecture? Well, consider the word “played” we discussed earlier (in the sentence “They played chess”). The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the past tense. In short, the word representation encodes many different characteristics of the word. If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot. This is why the Multi-Head Attention layer applies multiple different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word’s characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a

²³ This is the right part of figure 2 from the paper, reproduced with the kind authorization of the authors.

verb, another linear layer will extract just the fact that it is past tense, and so on. Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

At the time of this writing, there is no Transformer class or MultiHeadAttention class available for TensorFlow 2. However, you can check out TensorFlow’s great [tutorial for building a Transformer model for language understanding](#). Moreover, the TF Hub team is currently porting several Transformer-based modules to TensorFlow 2, and they should be available soon. In the meantime, I hope I have demonstrated that it is not that hard to implement a Transformer yourself, and it is certainly a great exercise!

Recent Innovations in Language Models

The year 2018 has been called the “ImageNet moment for NLP”: progress was astounding, with larger and larger LSTM and Transformer-based architectures trained on immense datasets. I highly recommend you check out the following papers, all published in 2018:

- The [ELMo paper](#)²⁴ 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. For example, the word “queen” will not have the same embedding in “Queen of the United Kingdom” and in “queen bee.”
- The [ULMFiT paper](#)²⁵ by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model using self-supervised learning (i.e., generating the labels automatically from the data) on a huge text corpus, then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, they showed that by fine-tuning the pretrained model on just 100 labeled examples, they could achieve the same performance as a model trained from scratch on 10,000 examples.
- The [GPT paper](#)²⁶ by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining, but this time using a

24 Matthew Peters et al., “Deep Contextualized Word Representations,” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies 1* (2018): 2227–2237.

25 Jeremy Howard and Sebastian Ruder, “Universal Language Model Fine-Tuning for Text Classification,” *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics 1* (2018): 328–339.

26 Alec Radford et al., “Improving Language Understanding by Generative Pre-Training” (2018).

Transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A entails sentence B),²⁷ similarity (e.g., “Nice weather today” is very similar to “It is sunny”), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions). Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the **GPT-2 paper**,²⁸ which proposed a very similar architecture, but larger still (with over 1.5 billion parameters!) and they showed that it could achieve good performance on many tasks without any fine-tuning. This is called *zero-shot learning* (ZSL). A smaller version of the GPT-2 model (with “just” 117 million parameters) is available at <https://github.com/openai/gpt-2>, along with its pretrained weights.

- The **BERT paper**²⁹ by Jacob Devlin and other Google researchers also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer’s encoder). This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model’s strength:

Masked language model (MLM)

Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is “She had fun at the birthday party,” then the model may be given the sentence “She <mask> fun at the <mask> party” and it must predict the words “had” and “birthday” (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

27 For example, the sentence “Jane had a lot of fun at her friend’s birthday party” entails “Jane enjoyed the party,” but it is contradicted by “Everyone hated the party” and it is unrelated to “The Earth is flat.”

28 Alec Radford et al., “Language Models Are Unsupervised Multitask Learners” (2019).

29 Jacob Devlin et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies 1* (2019).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that “The dog sleeps” and “It snores loudly” are consecutive sentences, while “The dog sleeps” and “The Earth orbits the Sun” are not consecutive. This is a challenging task, and it significantly improves the performance of the model when it is fine-tuned on tasks such as question answering or entailment.

As you can see, the main innovations in 2018 and 2019 have been better subword tokenization, shifting from LSTMs to Transformers, and pretraining universal language models using self-supervised learning, then fine-tuning them with very few architectural changes (or none at all). Things are moving fast; no one can say what architectures will prevail next year. Today, it’s clearly Transformers, but tomorrow it might be CNNs (e.g., check out the [2018 paper](#)³⁰ by Maha Elbayad et al., where the researchers use masked 2D convolutional layers for sequence-to-sequence tasks). Or it might even be RNNs, if they make a surprise comeback (e.g., check out the [2018 paper](#)³¹ by Shuai Li et al. that shows that by making neurons independent of each other in a given RNN layer, it is possible to train much deeper RNNs capable of learning much longer sequences).

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks (GANs) to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use Encoder–Decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?

30 Maha Elbayad et al., “Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction,” arXiv preprint arXiv:1808.03867 (2018).

31 Shuai Li et al., “Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 5457–5466.

6. What is the most important layer in the Transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in [their paper](#) about LSTMs. They are artificial grammars that produce strings such as “BPBTSXXVPSEPE.” Check out Jenny Orr’s [nice introduction](#) to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr’s page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don’t.
9. Train an Encoder–Decoder model that can convert a date string from one format to another (e.g., from “April 22, 2019” to “2019-04-22”).
10. Go through TensorFlow’s [Neural Machine Translation with Attention tutorial](#).
11. Use one of the recent language models (e.g., BERT) to generate more convincing Shakespearean text.

Solutions to these exercises are available in [Appendix A](#).