Deep Learning

Lecture Topic:
Deep Learning for Text

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand how to process text data for machine learning applications

2. Understand bag-of-words approaches and sequence-modeling approaches for text processing

3. Understand the transformer architecture

4. Understand sequence-to-sequence learning

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Roadmap of this lecture (continued):

3. Sequence-to-Sequence Learning

4. Seq2Seq for Machine Translation

4.1 Try 1 for Machine Translation: RNN

4.2 Try 2 for Machine Translation: Transformer

Topics in NLP

That's what modern NLP is about: using machine learning and large datasets to give computers the ability not to *understand* language, which is a more lofty goal, but to ingest a piece of language as input and return something useful, like predicting the following:

- "What's the topic of this text?" (text classification)
- "Does this text contain abuse?" (content filtering)
- "Does this text sound positive or negative?" (sentiment analysis)
- "What should be the next word in this incomplete sentence?" (language modeling)
- "How would you say this in German?" (translation)
- "How would you summarize this article in one paragraph?" (summarization)
- etc.

Preparing Text Data

Deep learning models, being differentiable functions, can only process numeric tensors: they can't take raw text as input. *Vectorizing* text is the process of transforming text into numeric tensors. Text vectorization processes come in many shapes and forms, but they all follow the same template (see figure 11.1):

- First, you *standardize* the text to make it easier to process, such as by converting it to lowercase or removing punctuation.
- You split the text into units (called *tokens*), such as characters, words, or groups of words. This is called *tokenization*.
- You convert each such token into a numerical vector. This will usually involve first *indexing* all tokens present in the data.

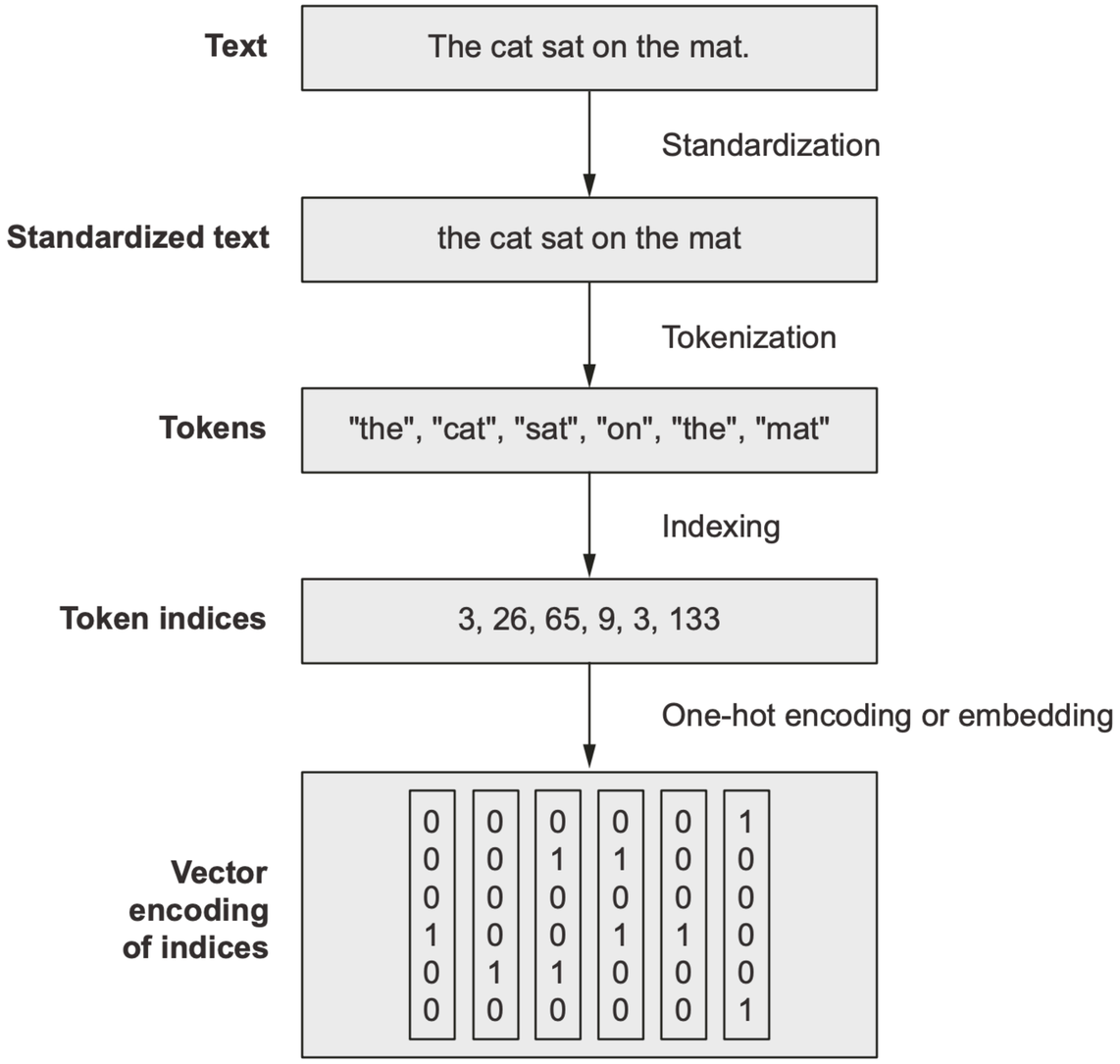


Figure 11.1 From raw text to vectors

Preparing Text Data: Text splitting (tokenization)

Once your text is standardized, you need to break it up into units to be vectorized (tokens), a step called *tokenization*. You could do this in three different ways:

- *Word-level tokenization*—Where tokens are space-separated (or punctuation-separated) substrings. A variant of this is to further split words into subwords when applicable—for instance, treating "staring" as "star+ing" or "called" as "call+ed."
- *N-gram tokenization*—Where tokens are groups of *N* consecutive words. For instance, "the cat" or "he was" would be 2-gram tokens (also called bigrams).
- *Character-level tokenization*—Where each character is its own token. In practice, this scheme is rarely used, and you only really see it in specialized contexts, like text generation or speech recognition.

Preparing Text Data: Text splitting (tokenization)

In general, you'll always use either word-level or N-gram tokenization. There are two kinds of text-processing models: those that care about word order, called *sequence models*, and those that treat input words as a set, discarding their original order, called *bag-of-words models*. If you're building a sequence model, you'll use word-level tokenization, and if you're building a bag-of-words model, you'll use N-gram tokenization. N-grams are a way to artificially inject a small amount of local word order information into the model. Throughout this chapter, you'll learn more about each type of model

Understanding N-grams and bag-of-words

Word N-grams are groups of *N* (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence "the cat sat on the mat." It may be decomposed into the following set of 2-grams:

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

It may also be decomposed into the following set of 3-grams:

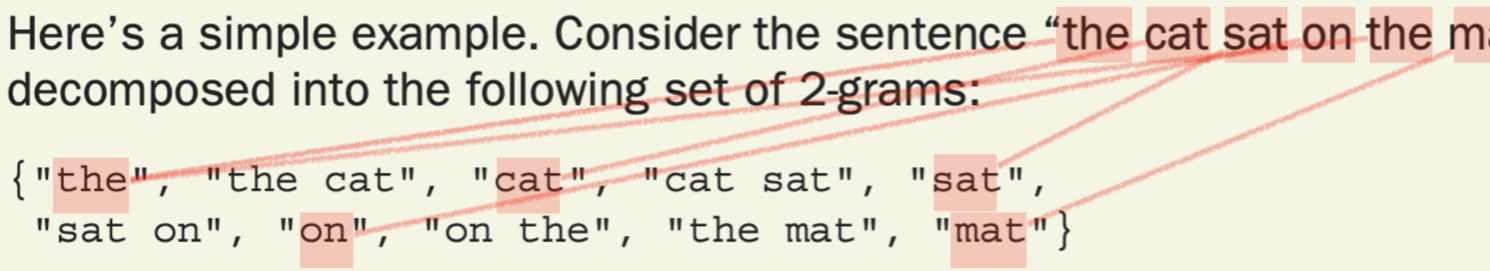
```
{"the", "the cat", "cat", "cat sat", "the cat sat",
 "sat", "sat on", "on", "cat sat on", "on the",
 "sat on the", "the mat", "mat", "on the mat"}
```

Such a set is called a *bag-of-2-grams* or *bag-of-3-grams*, respectively. The term "bag" here refers to the fact that you're dealing with a set of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization methods is called *bag-of-words* (or *bag-of-N-grams*).

Understanding N-grams and bag-of-words

Word N-grams are groups of *N* (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence "the cat sat on the mat." It may be decomposed into the following set of 2-grams:



```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

It may also be decomposed into the following set of 3-grams:

```
{"the", "the cat", "cat", "cat sat", "the cat sat",
 "sat", "sat on", "on", "cat sat on", "on the",
 "sat on the", "the mat", "mat", "on the mat"}
```

Such a set is called a *bag-of-2-grams* or *bag-of-3-grams*, respectively. The term "bag" here refers to the fact that you're dealing with a set of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization methods is called *bag-of-words* (or *bag-of-N-grams*).

Understanding N-grams and bag-of-words

Word N-grams are groups of *N* (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence "the cat sat on the mat." It may be decomposed into the following set of 2-grams:

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

It may also be decomposed into the following set of 3-grams:

```
{"the", "the cat", "cat", "cat sat", "the cat sat",
 "sat", "sat on", "on", "cat sat on", "on the",
 "sat on the", "the mat", "mat", "on the mat"}
```

Such a set is called a *bag-of-2-grams* or *bag-of-3-grams*, respectively. The term "bag" here refers to the fact that you're dealing with a set of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization methods is called *bag-of-words* (or *bag-of-N-grams*).

Preparing Text Data: Vocabulary indexing

Build an index of all terms found in the training data (the "vocabulary"), and assign a unique integer to each entry in the vocabulary.

Something like this:

```
vocabulary = {}
for text in dataset:
    text = standardize(text)
    tokens = tokenize(text)
    for token in tokens:
        if token not in vocabulary:
            vocabulary[token] = len(vocabulary)
```

Preparing Text Data: Vocabulary indexing (example)

Text: The cat sat on the mat.

Vocabulary (bag of 2-grams): the, cat, sat, on, mat, the cat, cat sat, sat on, on the, the mat.

Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Preparing Text Data: Vocabulary indexing

You can then convert that integer into a vector encoding that can be processed by a neural network, like a one-hot vector:

```
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),))
    token_index = vocabulary[token]
    vector[token_index] = 1
    return vector
```

Note that at this step it's common to restrict the vocabulary to only the top 20,000 or 30,000 most common words found in the training data. Any text dataset tends to feature an extremely large number of unique terms, most of which only show up once or twice—indexing those rare terms would result in an excessively large feature space, where most features would have almost no information content.

Example of one-hot encoding

Text: The cat sat on the mat.

Vocabulary (bag of 2-grams): the, cat, sat, on, mat, the cat, cat sat, sat on, on the, the mat.

Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

One-hot encoding:

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

Preparing Text Data: Vocabulary indexing

Now, there's an important detail here that we shouldn't overlook: when we look up a new token in our vocabulary index, it may not necessarily exist. Your training data may not have contained any instance of the word "cherimoya" (or maybe you excluded it from your index because it was too rare), so doing `token_index = vocabulary["cherimoya"]` may result in a `KeyError`. To handle this, you should use an "out of vocabulary" index (abbreviated as *OOV index*)—a catch-all for any token that wasn't in the index. It's usually index 1: you're actually doing `token_index = vocabulary.get(token, 1)`. When decoding a sequence of integers back into words, you'll replace 1 with something like "[UNK]" (which you'd call an "OOV token").

Preparing Text Data: Vocabulary indexing

"Why use 1 and not 0?" you may ask. That's because 0 is already taken. There are two special tokens that you will commonly use: the OOV token (index 1), and the *mask token* (index 0). While the OOV token means "here was a word we did not recognize," the mask token tells us "ignore me, I'm not a word." You'd use it in particular to pad sequence data: because data batches need to be contiguous, all sequences in a batch of sequence data must have the same length, so shorter sequences should be padded to the length of the longest sequence. If you want to make a batch of data with the sequences `[5, 7, 124, 4, 89]` and `[8, 34, 21]`, it would have to look like this:

```
[[5,  7, 124, 4, 89]
 [8, 34,  21, 0,  0]]
```

Use the TextVectorization layer

This is what the `TextVectorization` layer looks like:

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization(
    output_mode="int",
)
```

Configures the layer to return sequences of words encoded as integer indices. There are several other output modes available, which you will see in action in a bit.

By default, the `TextVectorization` layer will use the setting "convert to lowercase and remove punctuation" for text standardization, and "split on whitespace" for tokenization. But importantly, you can provide custom functions for standardization and tokenization, which means the layer is flexible enough to handle any use case.

Use the TextVectorization layer

To index the vocabulary of a text corpus, just call the `adapt()` method of the layer with a `Dataset` object that yields strings, or just with a list of Python strings:

```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Note that you can retrieve the computed vocabulary via `get_vocabulary()`—this can be useful if you need to convert text encoded as integer sequences back into words. The first two entries in the vocabulary are the mask token (index 0) and the OOV token (index 1). Entries in the vocabulary list are sorted by frequency, so with a real-world dataset, very common words like "the" or "a" would come first.

```
>>> text_vectorization.get_vocabulary()
["", "[UNK]", "erase", "write", ...]
```

Use the TextVectorization layer

For a demonstration, let's try to encode and then decode an example sentence:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

Use the TextVectorization layer

For a demonstration, let's try to encode and then decode an example sentence:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

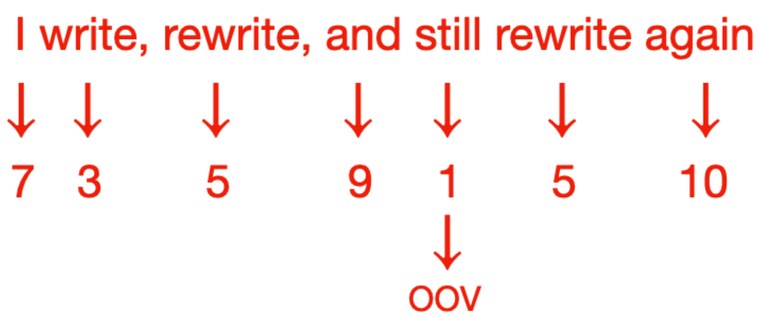
Recall that the vocabulary has already been built using the dataset:

```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Use the TextVectorization layer

For a demonstration, let's try to encode and then decode an example sentence:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

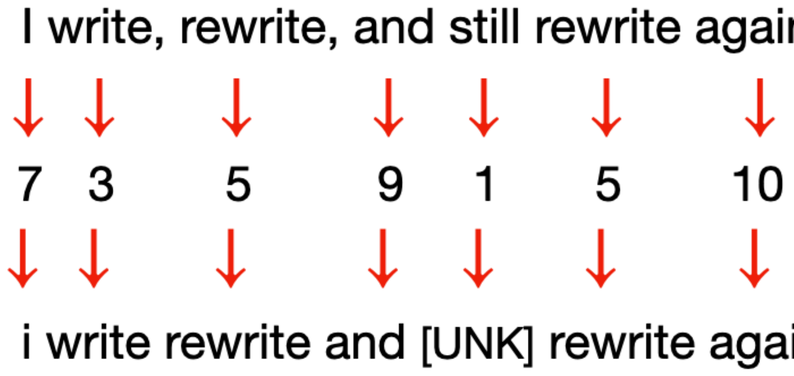



```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Use the TextVectorization layer

For a demonstration, let's try to encode and then decode an example sentence:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```




```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Use the TextVectorization layer

Using the `TextVectorization` layer in a `tf.data` pipeline or as part of a model

Importantly, because `TextVectorization` is mostly a dictionary lookup operation, it can't be executed on a GPU (or TPU)—only on a CPU. So if you're training your model on a GPU, your `TextVectorization` layer will run on the CPU before sending its output to the GPU. This has important performance implications.

There are two ways we could use our `TextVectorization` layer. The first option is to put it in the `tf.data` pipeline, like this:

```
int_sequence_dataset = string_dataset.map(
    text_vectorization,
    num_parallel_calls=4)
```

string_dataset would be a dataset that yields string tensors.

The num_parallel_calls argument is used to parallelize the map() call across multiple CPU cores.

Use the TextVectorization layer

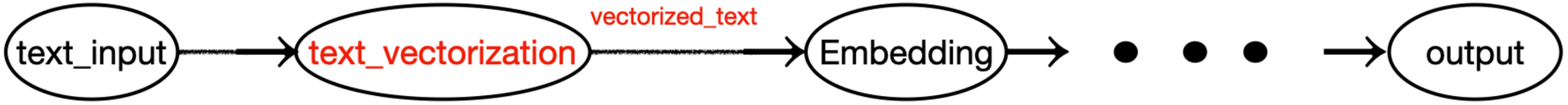
The second option is to make it part of the model (after all, it's a Keras layer), like this:

```
text_input = keras.Input(shape=(), dtype="string")
vectorized_text = text_vectorization(text_input)
embedded_input = keras.layers.Embedding(...)(vectorized_text)
output = ...
model = keras.Model(text_input, output)
```

Create a symbolic input that expects strings.

Apply the text vectorization layer to it.

You can keep chaining new layers on top—just your regular Functional API model.



Use the TextVectorization layer

So if you're training the model on GPU or TPU, you'll probably want to go with the first option to get the best performance. This is what we will do in all practical examples throughout this chapter. When training on a CPU, though, synchronous processing is fine: you will get 100% utilization of your cores regardless of which option you go with.

Now, if you were to export our model to a production environment, you would want to ship a model that accepts raw strings as input, like in the code snippet for the second option above—otherwise you would have to reimplement text standardization and tokenization in your production environment (maybe in JavaScript?), and you would face the risk of introducing small preprocessing discrepancies that would hurt the model's accuracy. Thankfully, the `TextVectorization` layer enables you to include text preprocessing right into your model, making it easier to deploy—even if you were originally using the layer as part of a `tf.data` pipeline. In the sidebar "Exporting a model that processes raw strings," you'll learn how to export an inference-only trained model that incorporates text preprocessing.

Quiz questions:

1. What is text tokenization?

2. What is text vectorization?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
   - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
   - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
   - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
   - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
   - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
   - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
   - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
   - 2.8 Attention and Transformer
   - 2.9 Try 8: Transformer encoder
   - 2.10 Try 9: Transformer encoder with positional embedding

Text classification: process IMDB dataset

Let's start by downloading the dataset from the Stanford page of Andrew Maas and uncompressing it:

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

You're left with a directory named aclImdb, with the following structure:

```
aclImdb/
...train/      → Training data
......pos/     → Positive samples in training data → 12,500 text files
......neg/     → Negative samples in training data → 12,500 text files
...test/       → Testing data
......pos/     → Positive samples in testing data → 12,500 text files
......neg/     → Negative samples in testing data → 12,500 text files
```

Text classification: process IMDB dataset

There's also a train/unsup subdirectory in there, which we don't need. Let's delete it:

```
!rm -r aclImdb/train/unsup
```

Take a look at the content of a few of these text files. Whether you're working with text data or image data, remember to always inspect what your data looks like before you dive into modeling it. It will ground your intuition about what your model is actually doing:

```
!cat aclImdb/train/pos/4077_10.txt
```

Text classification: process IMDB dataset

Next, let's prepare a validation set by setting apart 20% of the training text files in a new directory, aclImdb/val:

```
import os, pathlib, shutil, random

base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)
```

Shuffle the list of training files using a seed, to ensure we get the same validation set every time we run the code.

Take 20% of the training files to use for validation.

Move the files to aclImdb/val/neg and aclImdb/val/pos.

Text classification: process IMDB dataset

Remember how, in chapter 8, we used the `image_dataset_from_directory` utility to create a batched `Dataset` of images and their labels for a directory structure? You can do the exact same thing for text files using the `text_dataset_from_directory` utility. Let's create three `Dataset` objects for training, validation, and testing:

```
from tensorflow import keras
batch_size = 32

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
```

Running this line should output "Found 20000 files belonging to 2 classes"; if you see "Found 70000 files belonging to 3 classes," it means you forgot to delete the aclImdb/train/unsup directory.

Text classification: process IMDB dataset

These datasets yield inputs that are TensorFlow `tf.string` tensors and targets that are `int32` tensors encoding the value "0" or "1."

Listing 11.2 Displaying the shapes and dtypes of the first batch

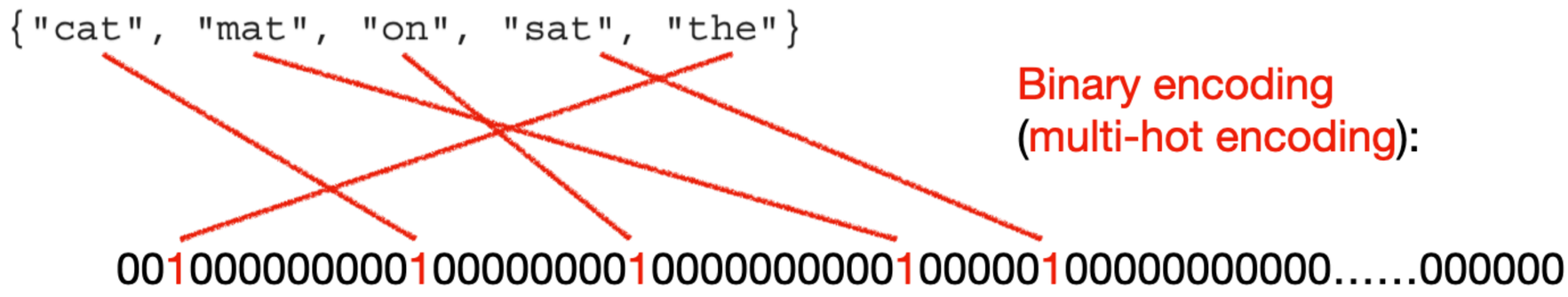
```
>>> for inputs, targets in train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32,)
inputs.dtype: <dtype: "string">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor(b"This string contains the movie review.", shape=(),
    dtype=string)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
   - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
   - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
   - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
   - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
   - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
   - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
   - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
   - 2.8 Attention and Transformer
   - 2.9 Try 8: Transformer encoder
   - 2.10 Try 9: Transformer encoder with positional embedding

Try 1: Bag-of-words approach, unigram with binary encoding

If you use a bag of single words, the sentence “the cat sat on the mat” becomes



First, let's process our raw text datasets with a `TextVectorization` layer so that they yield multi-hot encoded binary word vectors. Our layer will only look at single words (that is to say, *unigrams*).

Listing 11.3 Preprocessing our datasets with a `TextVectorization` layer

```
text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
)
text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)

binary_1gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

Limit the vocabulary to the 20,000 most frequent words. Otherwise we'd be indexing every word in the training data—potentially tens of thousands of terms that only occur once or twice and thus aren't informative. In general, 20,000 is the right vocabulary size for text classification.

Encode the output tokens as multi-hot binary vectors.

Prepare a dataset that only yields raw text inputs (no labels).

Use that dataset to index the dataset vocabulary via the adapt() method.

Prepare processed versions of our training, validation, and test dataset. Make sure to specify num_parallel_calls to leverage multiple CPU cores.

Try 1: Bag-of-words approach, unigram with binary encoding

You can try to inspect the output of one of these datasets.

Listing 11.4 Inspecting the output of our binary unigram dataset

```
>>> for inputs, targets in binary_1gram_train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32, 20000)
inputs.dtype: <dtype: "float32">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor([1. 1. 1. ... 0. 0. 0.], shape=(20000,), dtype=float32)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

Inputs are batches of 20,000-dimensional vectors.

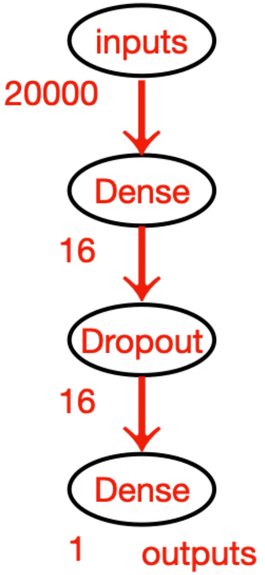
These vectors consist entirely of ones and zeros.

Try 1: Bag-of-words approach, unigram with binary encoding

Next, let's write a reusable model-building function that we'll use in all of our experiments in this section.

Listing 11.5 Our model-building utility

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(max_tokens=20000, hidden_dim=16):
    inputs = keras.Input(shape=(max_tokens,))
    x = layers.Dense(hidden_dim, activation="relu")(inputs)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model
```



Try 1: Bag-of-words approach, unigram with binary encoding

Finally, let's train and test our model.

Listing 11.6 Training and testing the binary unigram model

```
model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_1gram.keras",
                                    save_best_only=True)
]
model.fit(binary_1gram_train_ds.cache(),
          validation_data=binary_1gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_1gram.keras")
print(f"Test acc: {model.evaluate(binary_1gram_test_ds)[1]:.3f}")
```

We call cache() on the datasets to cache them in memory: this way, we will only do the preprocessing once, during the first epoch, and we'll reuse the preprocessed texts for the following epochs. This can only be done if the data is small enough to fit in memory.

Test accuracy: 89.2%

Quiz question:

1. What are the pros and cons of the approach in try 1?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Try 2: Bag-of-words approach, bigram with binary encoding

With bigrams, our sentence becomes

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

The `TextVectorization` layer can be configured to return arbitrary N-grams: bigrams, trigrams, etc. Just pass an `ngrams=N` argument as in the following listing.

Listing 11.7 Configuring the `TextVectorization` layer to return bigrams

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="multi_hot",
)
```

Try 2: Bag-of-words approach, bigram with binary encoding

Listing 11.8 Training and testing the binary bigram model

```
text_vectorization.adapt(text_only_train_ds)
binary_2gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_2gram.keras",
                                    save_best_only=True)
]
```

Try 2: Bag-of-words approach, bigram with binary encoding

```
model.fit(binary_2gram_train_ds.cache(),
          validation_data=binary_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_2gram.keras")
print(f"Test acc: {model.evaluate(binary_2gram_test_ds)[1]:.3f}")
```

Test accuracy = 90.4%

Better than “Bag-of-words, unigram with binary encoding”:
Test accuracy = 89.2%

Quiz question:

1. What are the pros and cons of the approach in try 2?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Try 3: Bag-of-words approach, bigram with TF-IDF encoding

You can also add a bit more information to this representation by counting how many times each word or N-gram occurs, that is to say, by taking the histogram of the words over the text:

```
{"the": 2, "the cat": 1, "cat": 1, "cat sat": 1, "sat": 1,
 "sat on": 1, "on": 1, "on the": 1, "the mat: 1", "mat": 1}
```

Here's how you'd count bigram occurrences with the `TextVectorization` layer.

Listing 11.9 Configuring the `TextVectorization` layer to return token counts

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="count"
)
```

Try 3: Bag-of-words approach, bigram with TF-IDF encoding

TF-IDF is so common that it's built into the `TextVectorization` layer. All you need to do to start using it is to switch the `output_mode` argument to `"tf_idf"`.

Listing 11.10 Configuring `TextVectorization` to return TF-IDF-weighted outputs

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="tf_idf",
)
```

Try 3: Bag-of-words approach, bigram with TF-IDF encoding

Understanding TF-IDF normalization

The more a given term appears in a document, the more important that term is for understanding what the document is about. At the same time, the frequency at which the term appears across all documents in your dataset matters too: terms that appear in almost every document (like “the” or “a”) aren’t particularly informative, while terms that appear only in a small subset of all texts (like “Herzog”) are very distinctive, and thus important. TF-IDF is a metric that fuses these two ideas. It weights a given term by taking “term frequency,” how many times the term appears in the current document, and dividing it by a measure of “document frequency,” which estimates how often the term comes up across the dataset. You’d compute it as follows:

```
def tfidf(term, document, dataset):
    term_freq = document.count(term)
    doc_freq = math.log(sum(doc.count(term) for doc in dataset) + 1)
    return term_freq / doc_freq
```

Try 3: Bag-of-words approach, bigram with TF-IDF encoding

Listing 11.11 Training and testing the TF-IDF bigram model

```
text_vectorization.adapt(text_only_train_ds)

tfidf_2gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
tfidf_2gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
tfidf_2gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("tfidf_2gram.keras",
                                    save_best_only=True)
]
model.fit(tfidf_2gram_train_ds.cache(),
          validation_data=tfidf_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("tfidf_2gram.keras")
print(f"Test acc: {model.evaluate(tfidf_2gram_test_ds)[1]:.3f}")
```

The adapt() call will learn the TF-IDF weights in addition to the vocabulary.

Test accuracy =89.8%

Better than “Bag-of-words, unigram with binary encoding”: Test accuracy = 89.2%

Worse than “Bag-of-words, bigram with binary encoding”: Test accuracy =90.4%

Exporting a model that processes raw strings

In the preceding examples, we did our text standardization, splitting, and indexing as part of the `tf.data` pipeline. But if we want to export a standalone model independent of this pipeline, we should make sure that it incorporates its own text preprocessing (otherwise, you'd have to reimplement in the production environment, which can be challenging or can lead to subtle discrepancies between the training data and the production data). Thankfully, this is easy.

Just create a new model that reuses your `TextVectorization` layer and adds to it the model you just trained:

```
inputs = keras.Input(shape=(1,), dtype="string")
processed_inputs = text_vectorization(inputs)
outputs = model(processed_inputs)
inference_model = keras.Model(inputs, outputs)
```

One input sample would be one string.

Apply text preprocessing.

Apply the previously trained model.

Instantiate the end-to-end model.

The resulting model can process batches of raw strings:

```
import tensorflow as tf
raw_text_data = tf.convert_to_tensor([
    ["That was an excellent movie, I loved it."],
])
predictions = inference_model(raw_text_data)
print(f"{float(predictions[0] * 100):.2f} percent positive")
```

Quiz question:

1. What are the pros and cons of the approach in try 3?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

The sequence model approach

Let's try out a first sequence model in practice. First, let's prepare datasets that return integer sequences.

Listing 11.12 Preparing integer sequence datasets

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y)),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

In order to keep a manageable input size, we'll truncate the inputs after the first 600 words. This is a reasonable choice, since the average review length is 233 words, and only 5% of reviews are longer than 600 words.

Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM

Listing 11.13 A sequence model built on one-hot encoded vector sequences

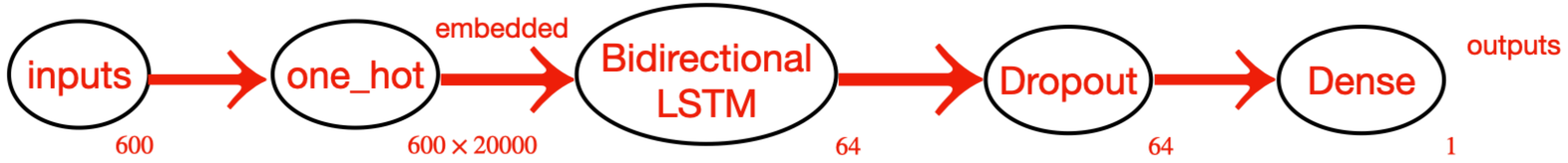
```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

One input is a sequence of integers.

Encode the integers into binary 20,000-dimensional vectors.

Add a bidirectional LSTM.

Finally, add a classification layer.



Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM

Now, let's train our model.

Test accuracy =87%

Listing 11.14 Training a first basic sequence model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Worse than "Bag-of-words, unigram with binary encoding": Test accuracy = 89.2%

Worse than "Bag-of-words, bigram with binary encoding": Test accuracy =90.4%

Worse than "Bag-of-words, bigram with TF-IDF encoding": Test accuracy =89.8%

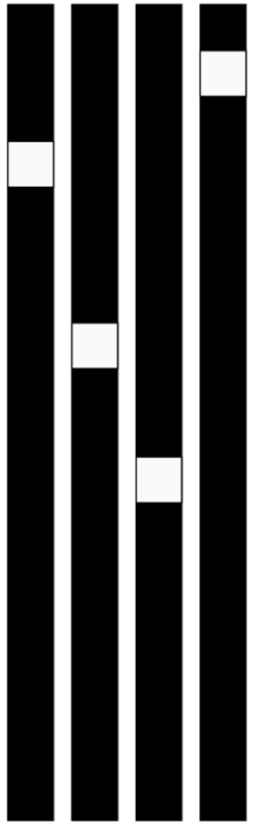
Quiz question:

1. What are the pros and cons of the approach in try 4?

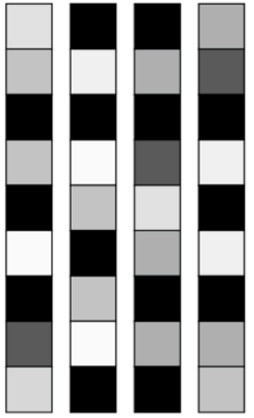
Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Use word embedding

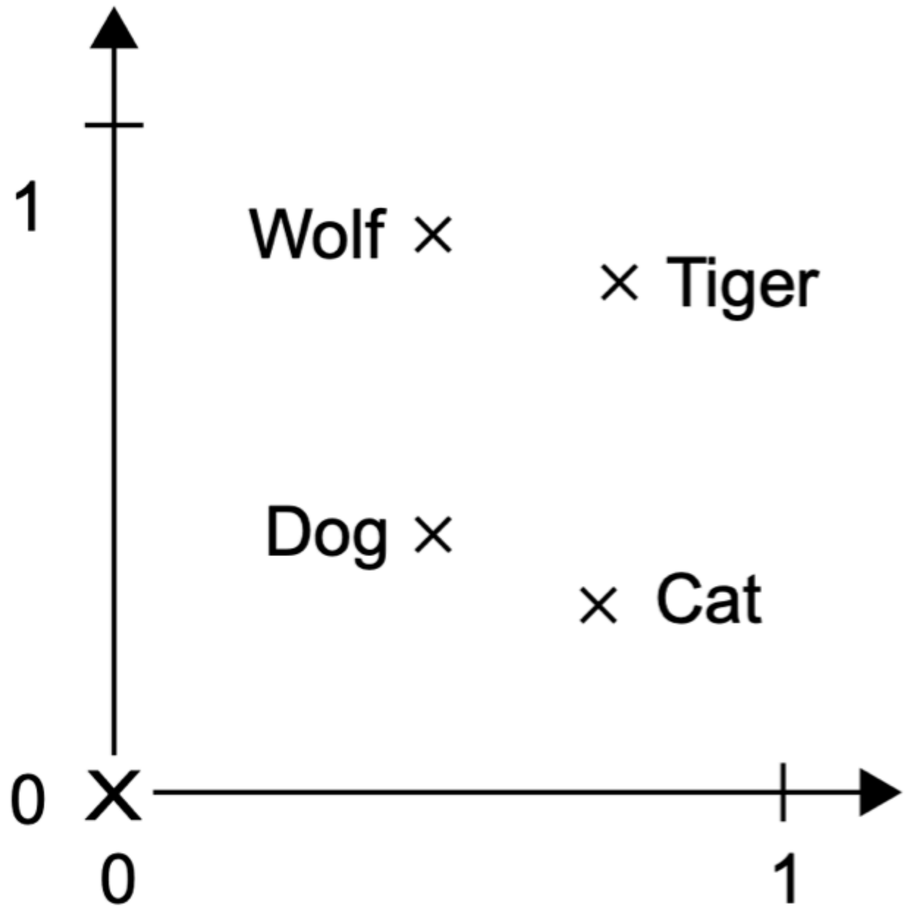


One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded



Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Structure revealed by word embedding



How to get word embedding

Let's look at how to use such an embedding space in practice. There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine learning task than the one you're trying to solve. These are called *pretrained word embeddings*.

Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM

It's thus reasonable to *learn* a new embedding space with every new task. Fortunately, backpropagation makes this easy, and Keras makes it even easier. It's about learning the weights of a layer: the `Embedding` layer.

Listing 11.15 Instantiating an `Embedding` layer

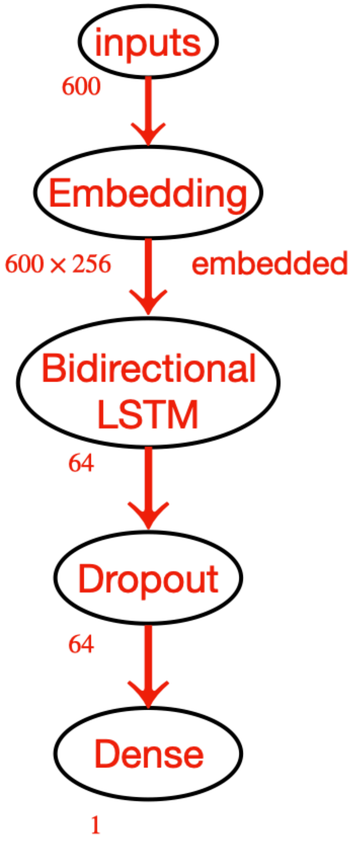
```
embedding_layer = layers.Embedding(input_dim=max_tokens, output_dim=256)
```

The Embedding layer takes at least two arguments: the number of possible tokens and the dimensionality of the embeddings (here, 256).

The `Embedding` layer takes as input a rank-2 tensor of integers, of shape `(batch_size, sequence_length)`, where each entry is a sequence of integers. The layer then returns a 3D floating-point tensor of shape `(batch_size, sequence_length, embedding_dimensionality)`.

Listing 11.16 Model that uses an `Embedding` layer trained from scratch

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```



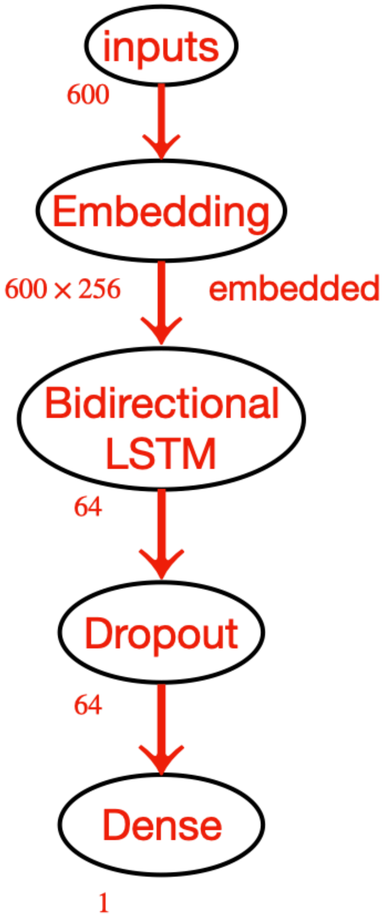
Test accuracy =87% (but it trains much faster than Try 4 due to smaller embedding space)

Worse than “Bag-of-words, unigram with binary encoding”:
Test accuracy = 89.2%

Worse than “Bag-of-words, bigram with binary encoding”:
Test accuracy =90.4%

Worse than “Bag-of-words, bigram with TF-IDF encoding”:
Test accuracy =89.8%

Comparable to “Sequence approach, one-hot encoding, Bi-directional LSTM”:
Test accuracy: 87%



Quiz question:

1. What are the pros and cons of the approach in try 5?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Understanding Padding and Masking

One thing that's slightly hurting model performance here is that our input sequences are full of zeros. This comes from our use of the `output_sequence_length=max_length` option in `TextVectorization` (with `max_length` equal to 600): sentences longer than 600 tokens are truncated to a length of 600 tokens, and sentences shorter than 600 tokens are padded with zeros at the end so that they can be concatenated together with other sequences to form contiguous batches.

We're using a bidirectional RNN: two RNN layers running in parallel, with one processing the tokens in their natural order, and the other processing the same tokens in reverse. The RNN that looks at the tokens in their natural order will spend its last iterations seeing only vectors that encode padding—possibly for several hundreds of iterations if the original sentence was short. The information stored in the internal state of the RNN will gradually fade out as it gets exposed to these meaningless inputs.

Understanding Padding and Masking

We need some way to tell the RNN that it should skip these iterations. There's an API for that: *masking*.

The `Embedding` layer is capable of generating a "mask" that corresponds to its input data. This mask is a tensor of ones and zeros (or True/False booleans), of shape `(batch_size, sequence_length)`, where the entry `mask[i, t]` indicates where time-step `t` of sample `i` should be skipped or not (the timestep will be skipped if `mask[i, t]` is 0 or False, and processed otherwise).

By default, this option isn't active—you can turn it on by passing `mask_zero=True` to your `Embedding` layer. You can retrieve the mask with the `compute_mask()` method:

```
>>> embedding_layer = Embedding(input_dim=10, output_dim=256, mask_zero=True)
>>>  some_input = [
...   [4, 3, 2, 1, 0, 0, 0],
...   [5, 4, 3, 2, 1, 0, 0],
...   [2, 1, 0, 0, 0, 0, 0]]
>>> mask = embedding_layer.compute_mask(some_input)
<tf.Tensor: shape=(3, 7), dtype=bool, numpy=
array([[ True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True, False, False],
       [ True,  True, False, False, False, False, False]])>
```

Understanding Padding and Masking

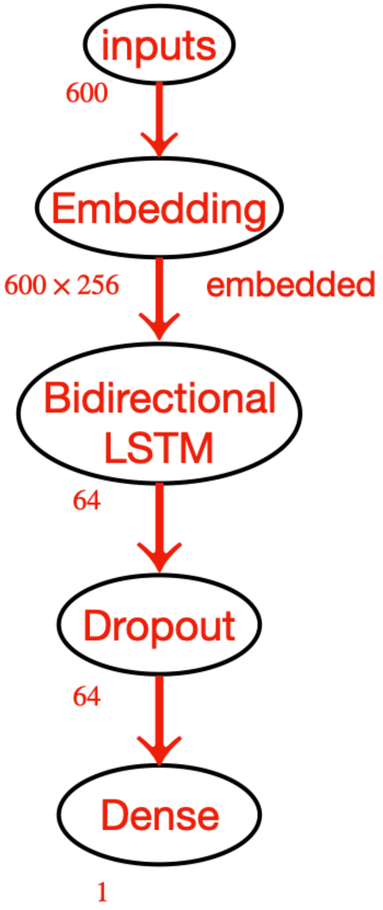
In practice, you will almost never have to manage masks by hand. Instead, Keras will automatically pass on the mask to every layer that is able to process it (as a piece of metadata attached to the sequence it represents). This mask will be used by RNN layers to skip masked steps. If your model returns an entire sequence, the mask will also be used by the loss function to skip masked steps in the output sequence.

Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM

Let's try retraining our model with masking enabled.

Listing 11.17 Using an `Embedding` layer with masking enabled

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(
    input_dim=max_tokens, output_dim=256, mask_zero=True)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```



Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM

Test accuracy =88%

```
callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru_with_masking.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru_with_masking.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Worse than “Try 1: Bag-of-words, unigram with binary encoding”: Test accuracy = 89.2%

Worse than “Try 2: Bag-of-words, bigram with binary encoding”: Test accuracy =90.4%

Worse than “Try 3: Bag-of-words, bigram with TF-IDF encoding”: Test accuracy =89.8%

Better than “Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM”: Test accuracy: 87%

Better than “Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM”: Test accuracy =87%

Quiz question:

1. What are the pros and cons of the approach in try 6?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
   - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
   - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
   - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
   - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
   - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
   - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
   - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
   - 2.8 Attention and Transformer
   - 2.9 Try 8: Transformer encoder
   - 2.10 Try 9: Transformer encoder with positional embedding

Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM

Popular word embedding:

1) Word2Vec: https://code.google.com/archive/p/word2vec

2) GloVe: https://nlp.stanford.edu/projects/glove

Let's look at how you can get started using GloVe embeddings in a Keras model. The same method is valid for Word2Vec embeddings or any other word-embedding database. We'll start by downloading the GloVe files and parse them. We'll then load the word vectors into a Keras `Embedding` layer, which we'll use to build a new model.

Get GloVe

First, let's download the GloVe word embeddings precomputed on the 2014 English Wikipedia dataset. It's an 822 MB zip file containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens).

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation.

Listing 11.18 Parsing the GloVe word-embeddings file

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print(f"Found {len(embeddings_index)} word vectors.")
```

Get GloVe ready for the Embedding layer

Next, let's build an embedding matrix that you can load into an `Embedding` layer. It must be a matrix of shape `(max_words, embedding_dim)`, where each entry *i* contains the `embedding_dim`-dimensional vector for the word of index *i* in the reference word index (built during tokenization).

Listing 11.19 Preparing the GloVe word-embeddings matrix

```
embedding_dim = 100

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Retrieve the vocabulary indexed by our previous TextVectorization layer.

Use it to create a mapping from words to their index in the vocabulary.

Prepare a matrix that we'll fill with the GloVe vectors.

Fill entry *i* in the matrix with the word vector for index *i*. Words not found in the embedding index will be all zeros.

Load GloVe into the Embedding layer

Finally, we use a `Constant` initializer to load the pretrained embeddings in an `Embedding` layer. So as not to disrupt the pretrained representations during training, we freeze the layer via `trainable=False`:

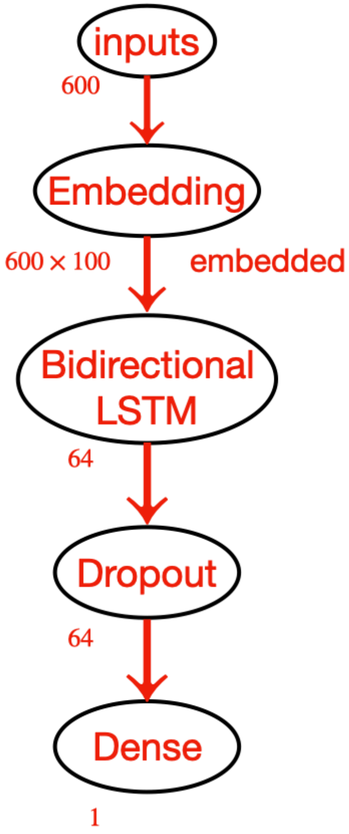
```
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)
```

We're now ready to train a new model—identical to our previous model, but leveraging the 100-dimensional pretrained GloVe embeddings instead of 128-dimensional learned embeddings.

Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM

Listing 11.20 Model that uses a pretrained Embedding layer

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("glove_embeddings_sequence_model.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("glove_embeddings_sequence_model.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```



Test accuracy: no better than Try 6 (but still worth trying it).

Quiz question:

1. What are the pros and cons of the approach in try 7?

Roadmap of this lecture:

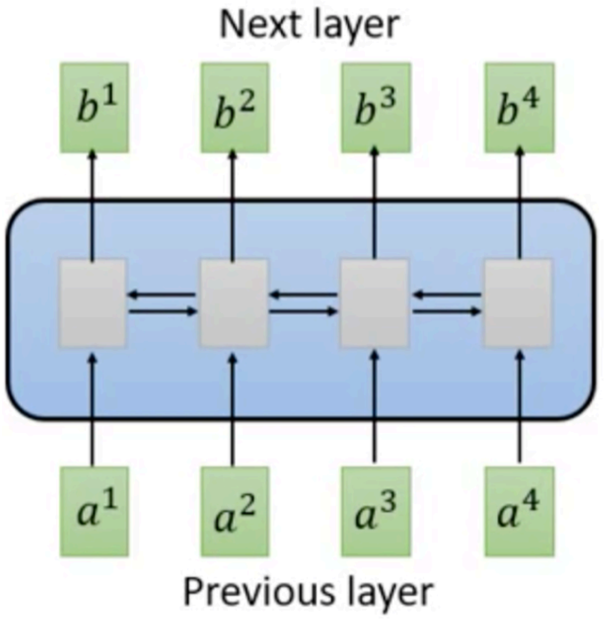
1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Attention and Transformer

Transformer: sequence-to-sequence model with “self-attention”

Adapted from lecture by Prof. Hugn-yi Lee “Transformer”

Sequence



When we need to process a sequence,
a common solution is to us RNN.

Benefit of RNN: every output uses information
from all the previous inputs (for uni-directional RNN)
or information from the whole sequence of inputs
(for bi-directional RNN)

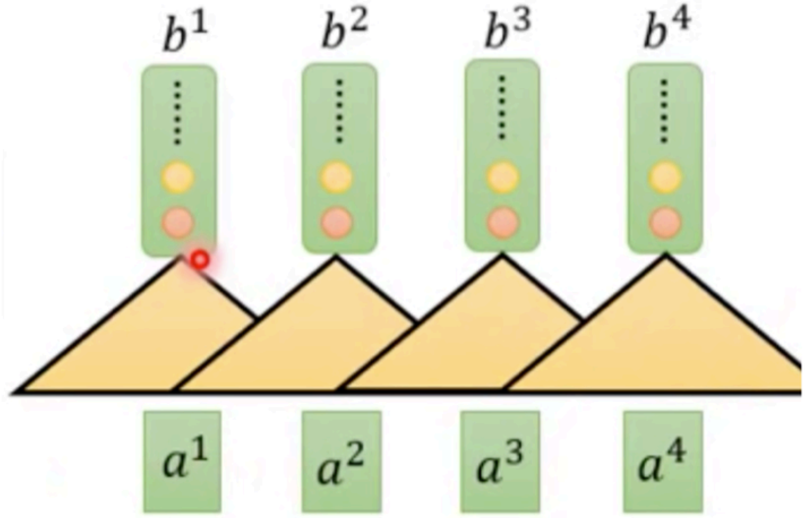
Drawback of RNN: hard to use parallel computing, slow

Sequence

CNN can also do sequence-to-sequence learning.

Benefit: parallel computing, fast.

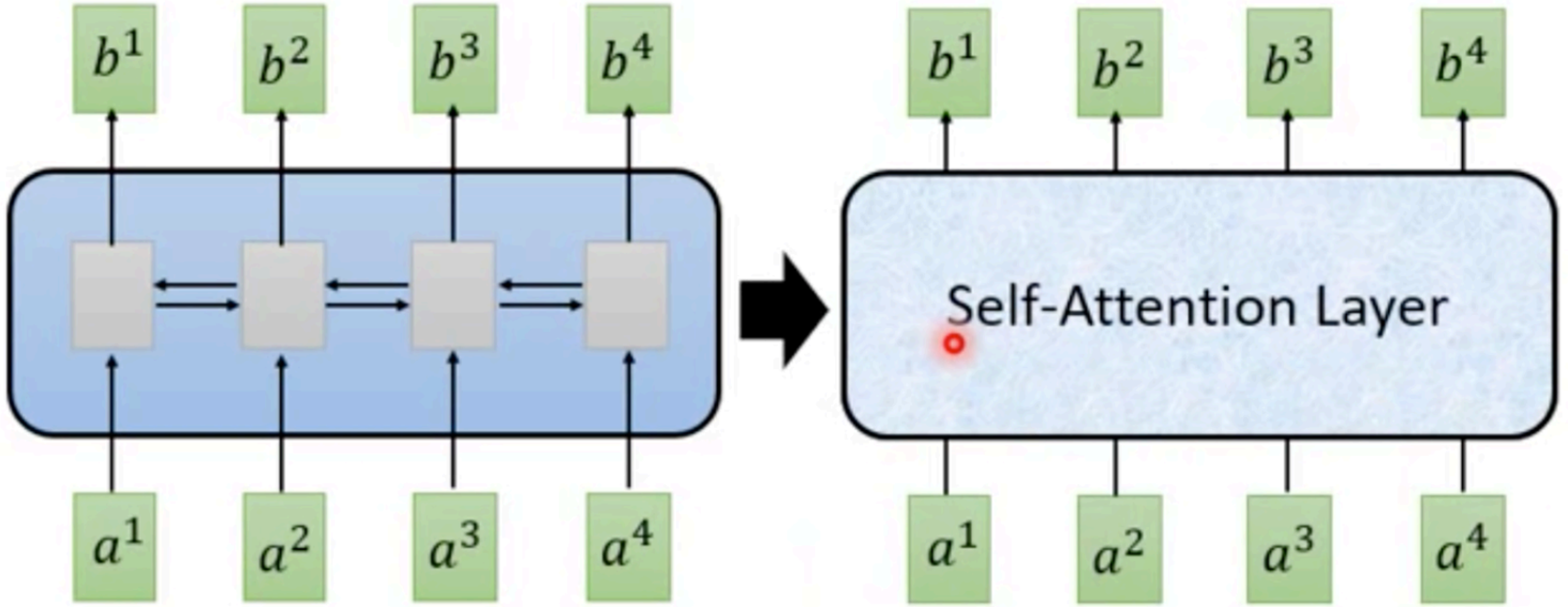
Drawback: every filter considers only nearby information, which is limited.



To consider global information, multiple layers are needed.

Self-Attention

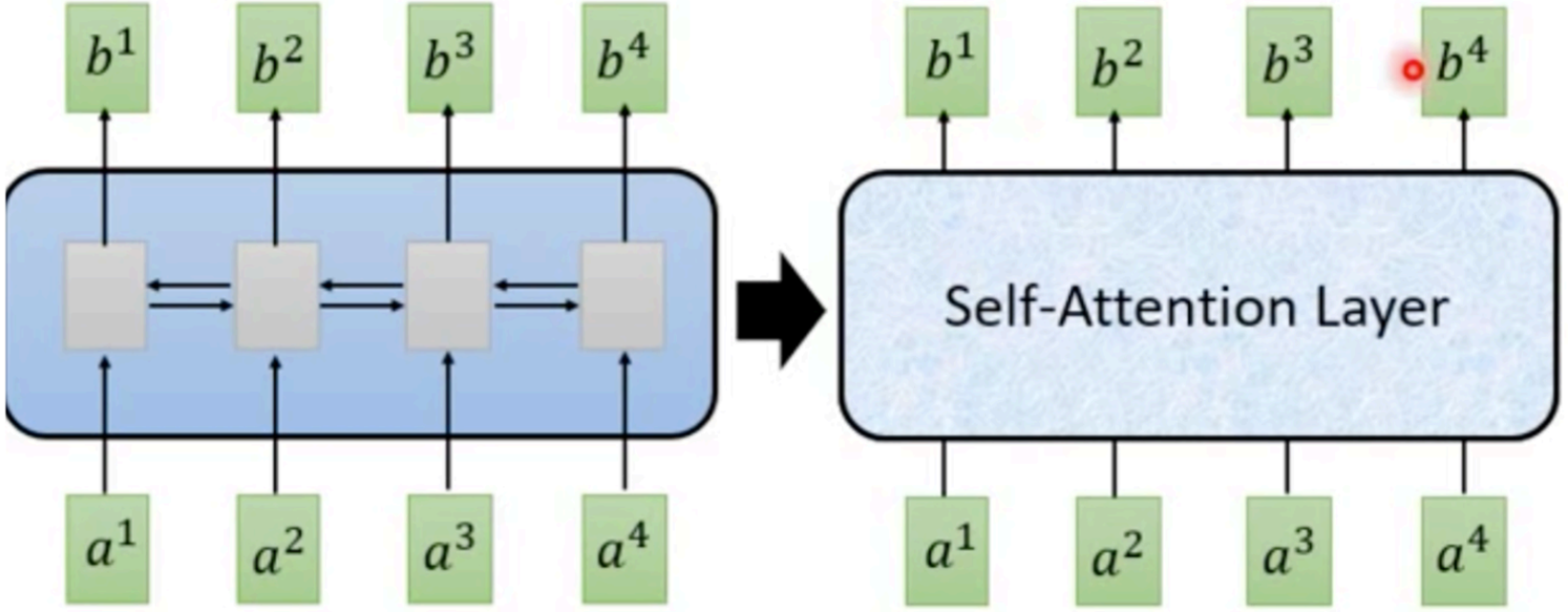
Use a “self-attention layer”
to do sequence-to-sequence learning.



Self-Attention

b^i is obtained based on the whole input sequence.

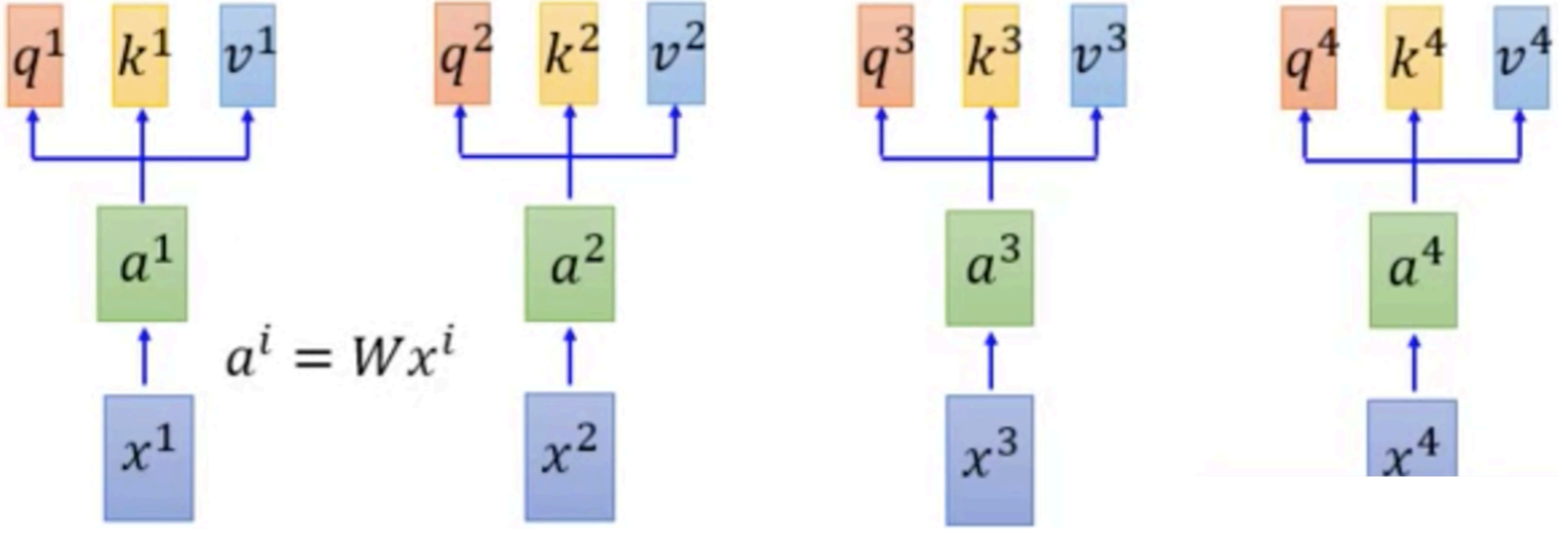
b^1, b^2, b^3, b^4 can be computed in parallel.



Self-attention

https://arxiv.org/abs/1706.03762

q: query
k: key
v: value



query-key-value model

This terminology comes from search engines and recommender systems.

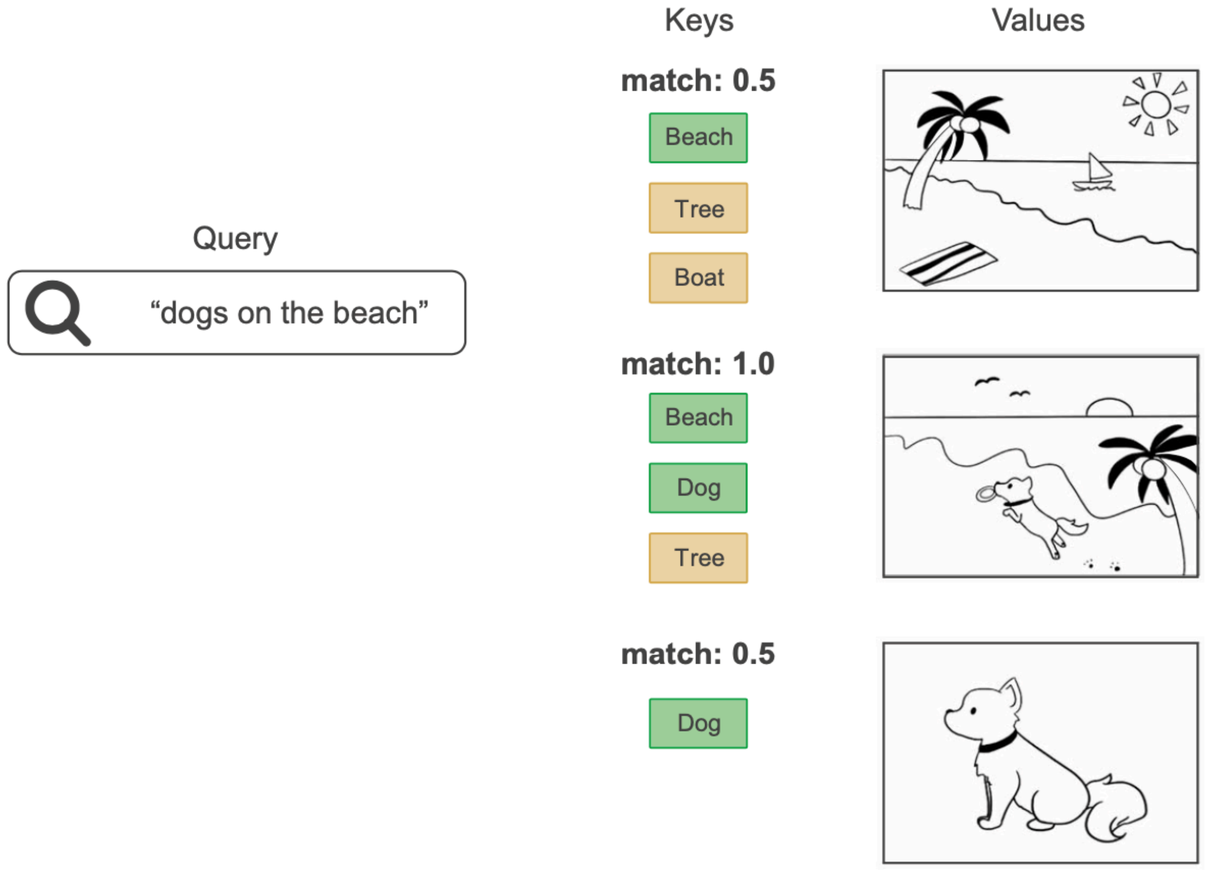


Figure 11.7 Retrieving images from a database: the “query” is compared to a set of “keys,” and the match scores are used to rank “values” (images).

Self-attention

https://arxiv.org/abs/1706.03762

q: query (to match others)

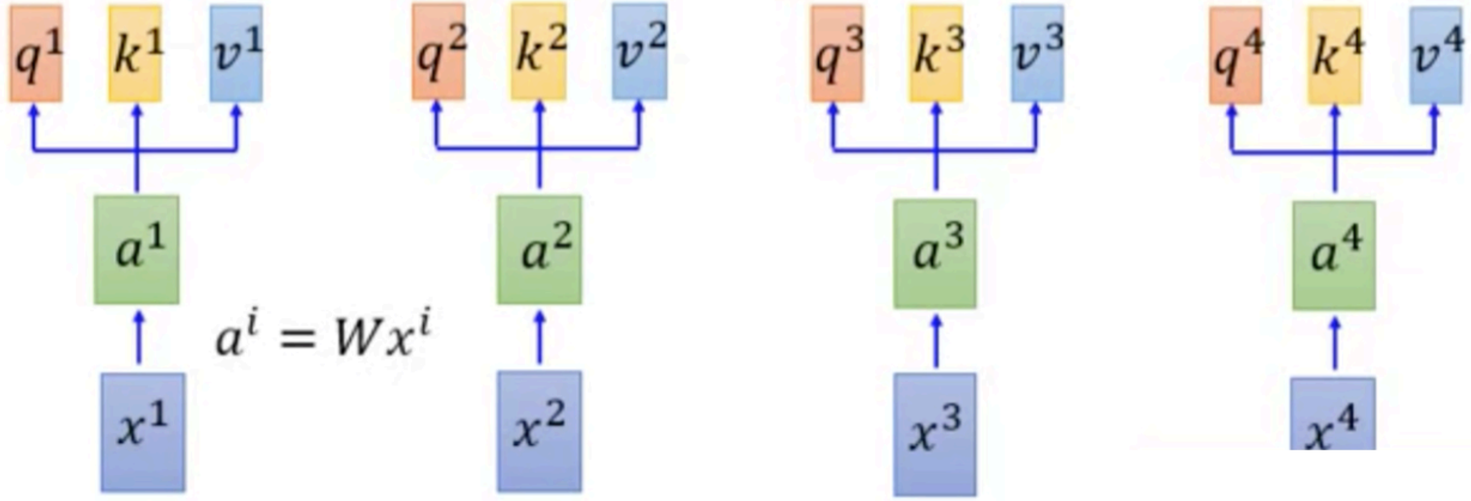
$$q^i = W^q a^i$$

k: key (to be matched)

$$k^i = W^k a^i$$

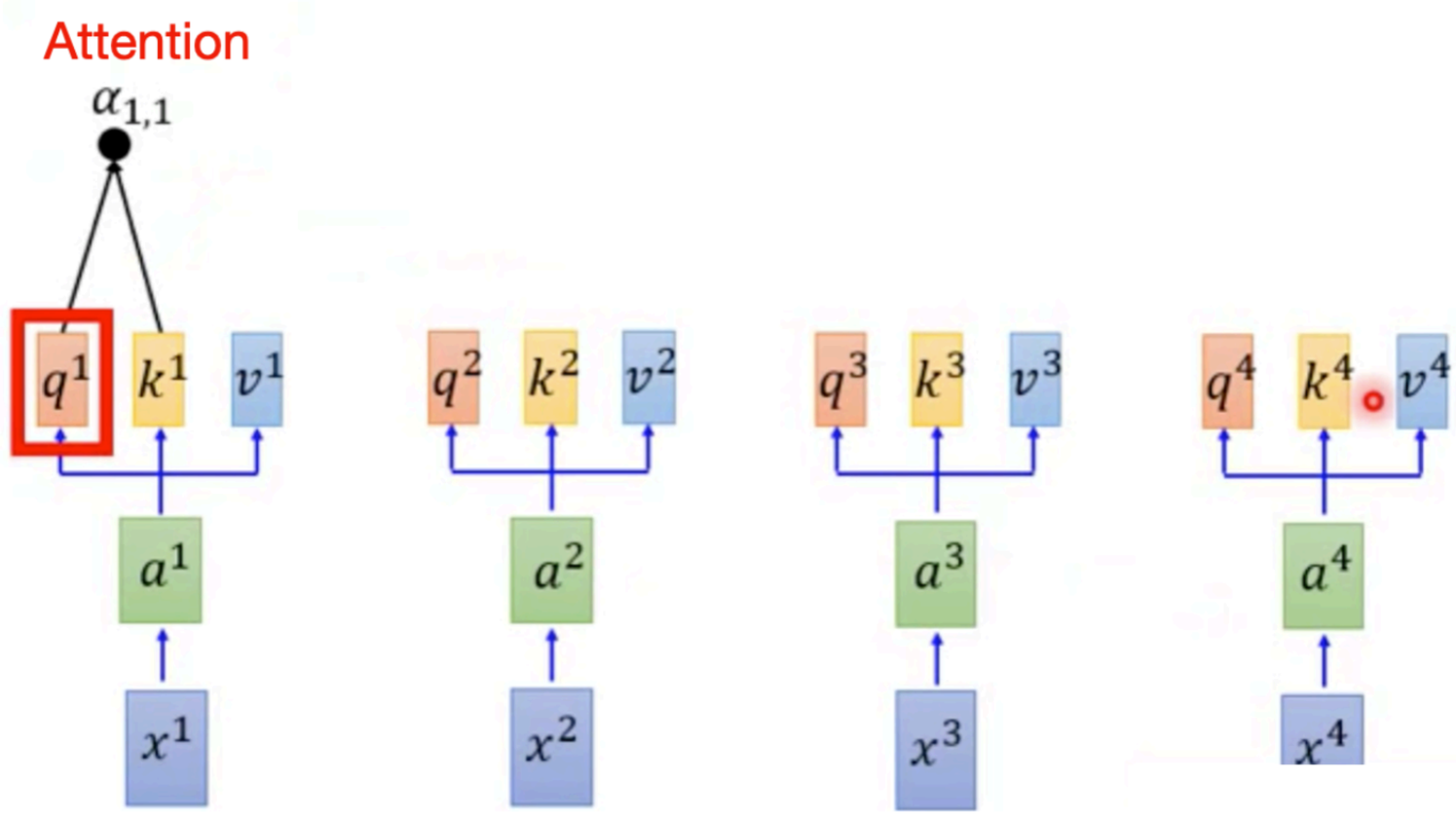
v: information to be extracted

$$v^i = W^v a^i$$

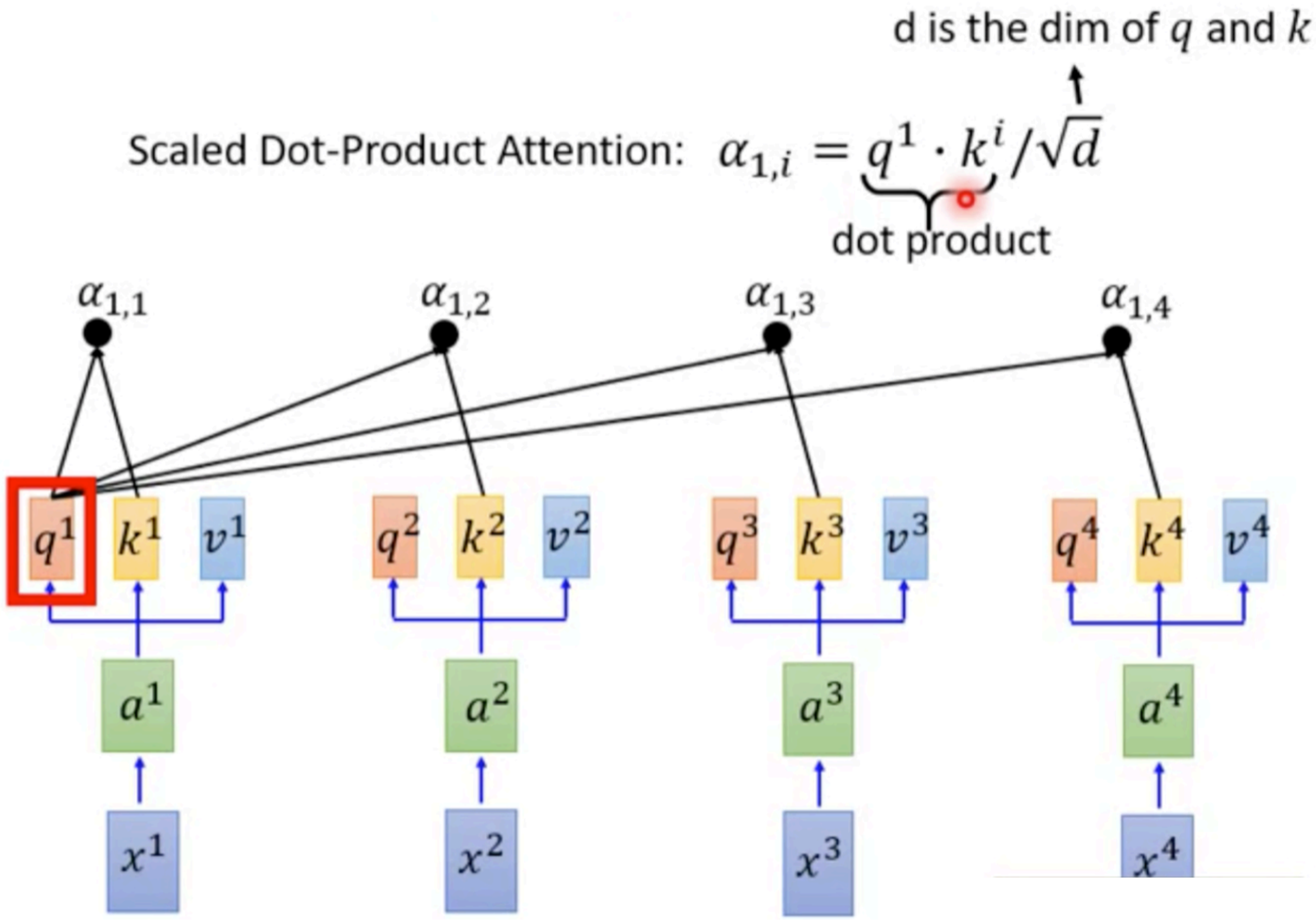


Match every query to every key, to find the corresponding “attention”.

Self-attention

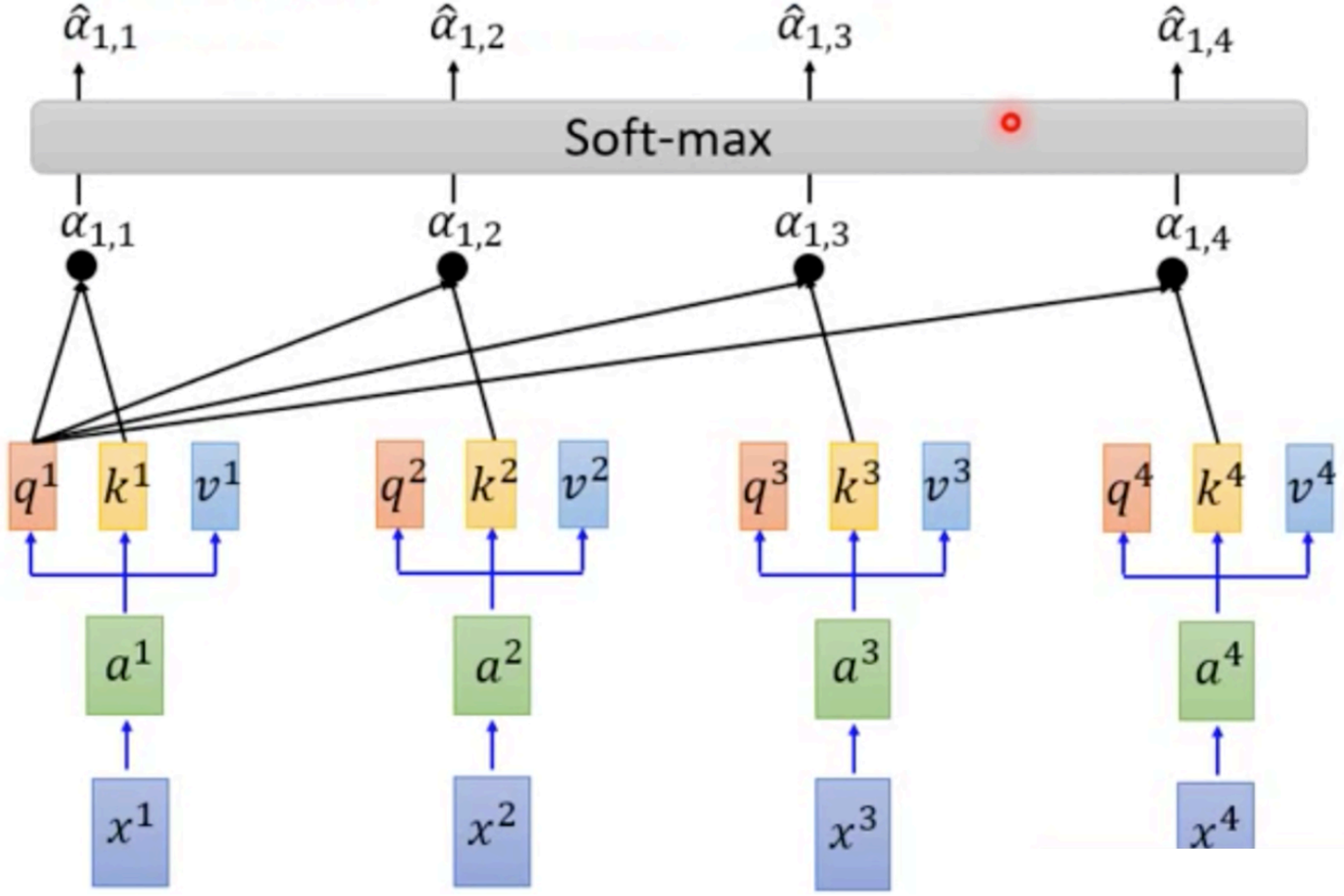


Self-attention



Self-attention

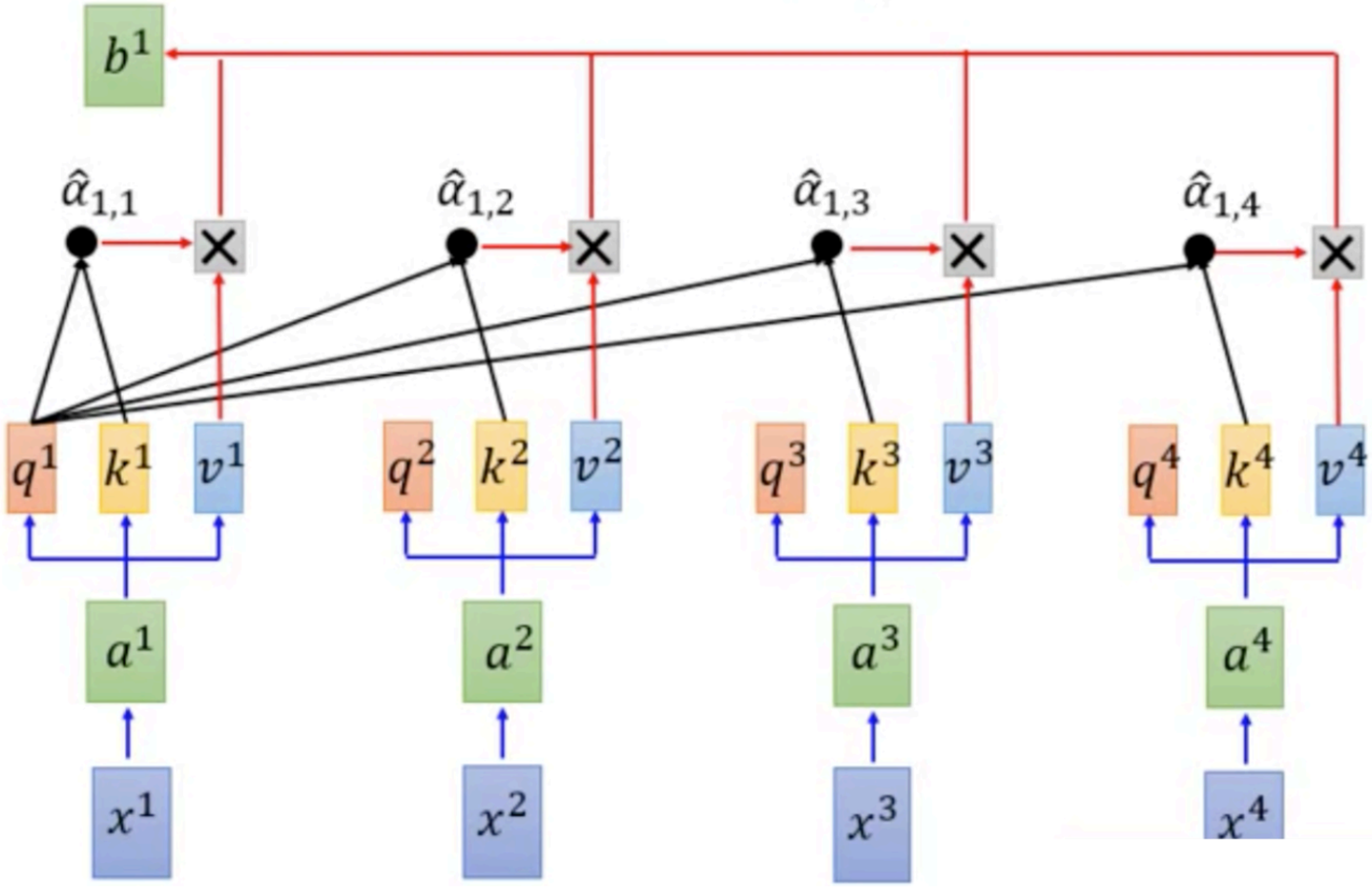
$$\hat{\alpha}_{1,i} = exp(\alpha_{1,i}) / \sum_j exp(\alpha_{1,j})$$



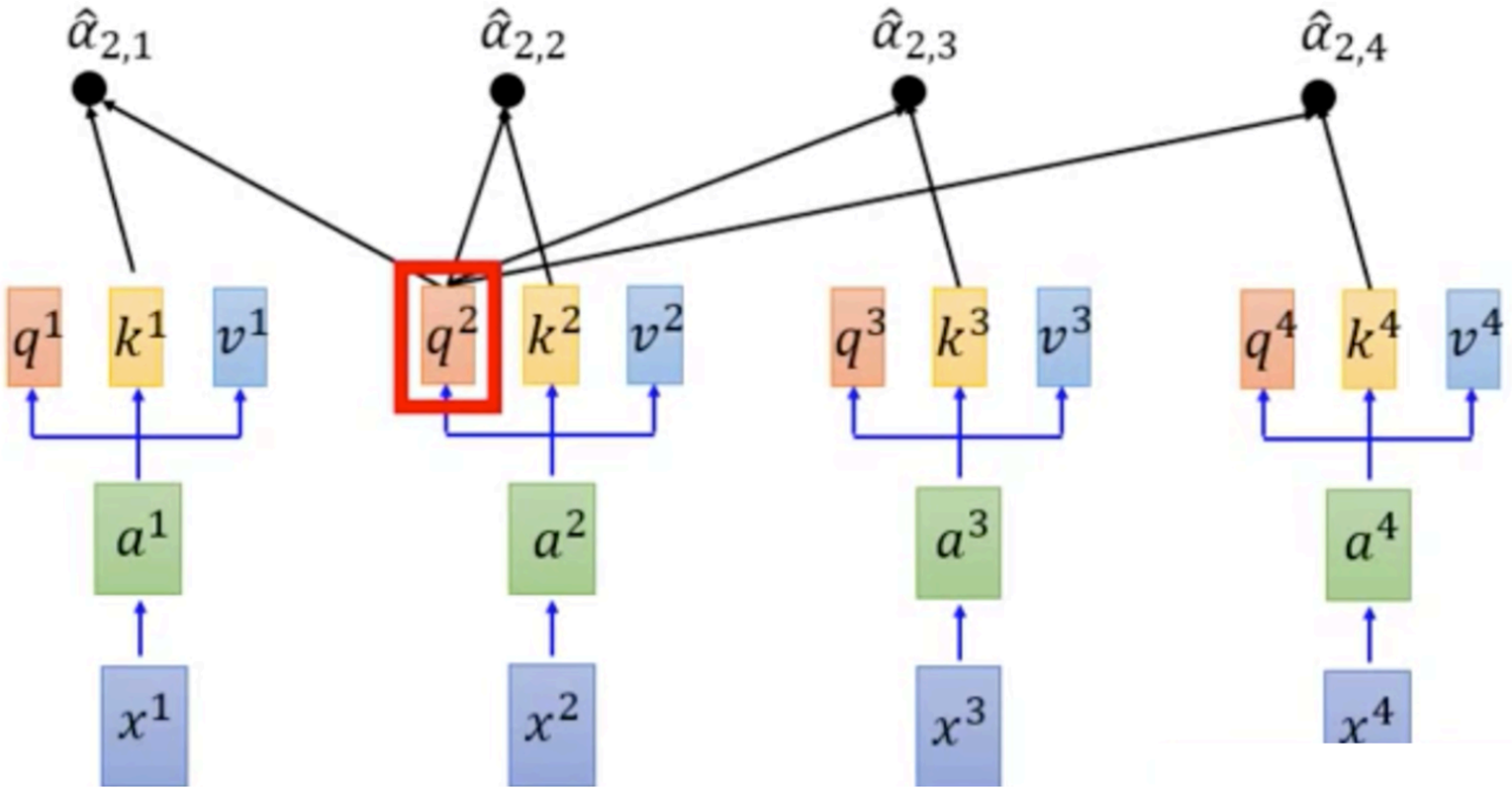
Self-attention

b^1 considers the whole sequence.

$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$

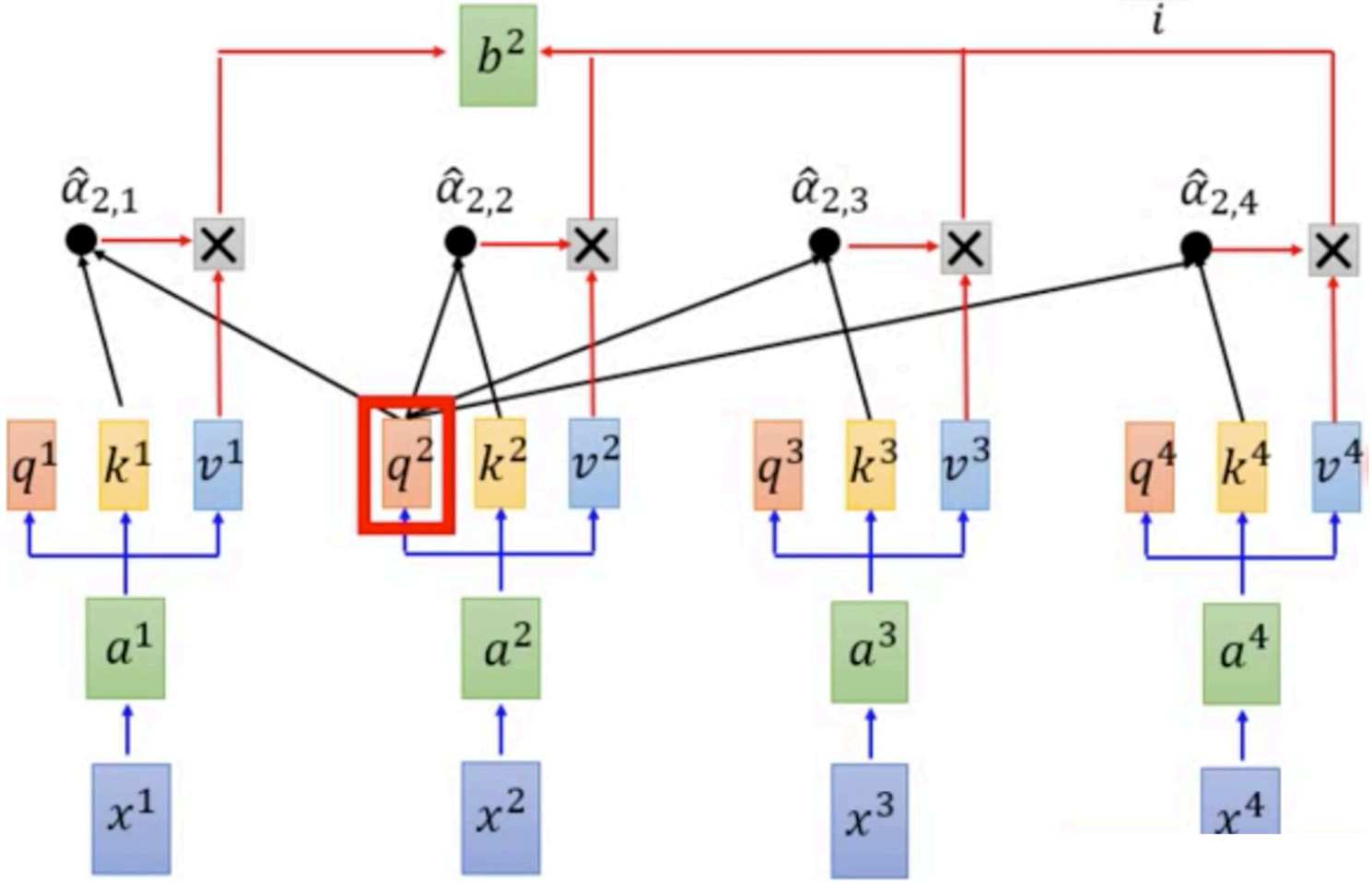


Self-attention

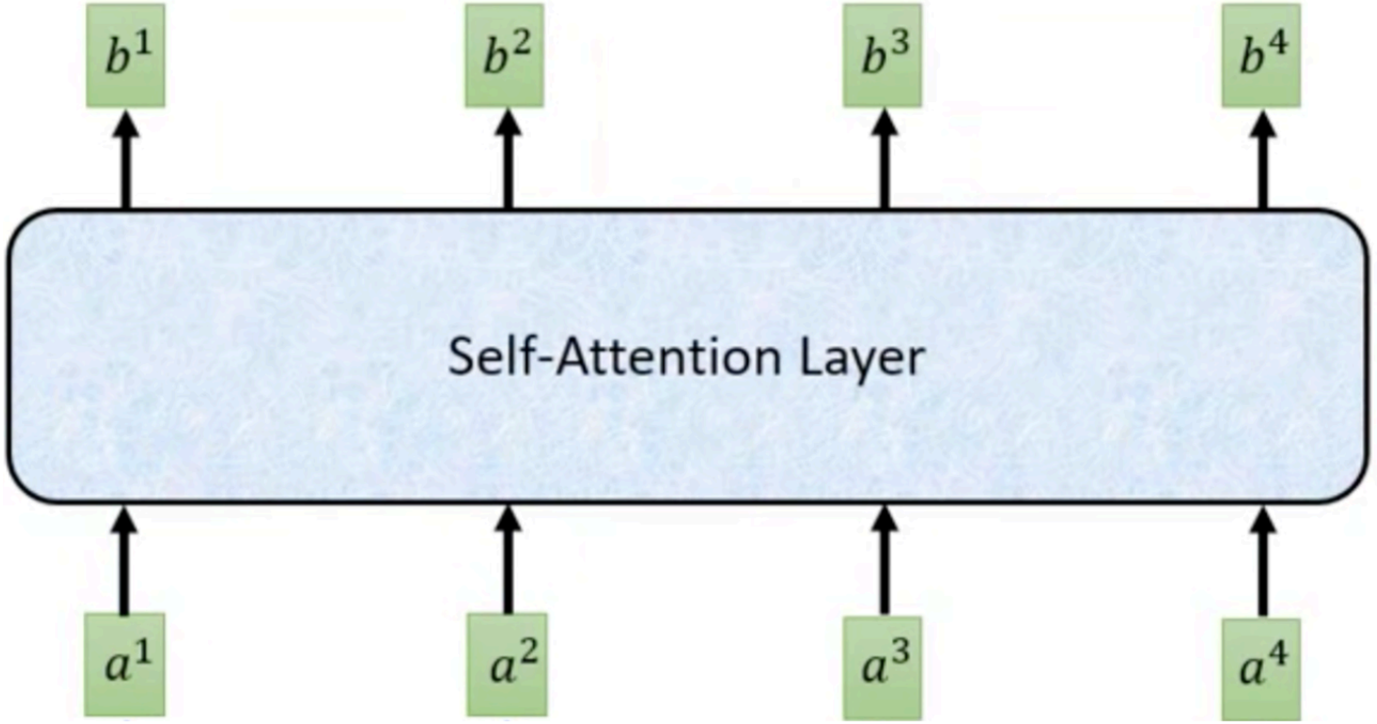


Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



Self-attention



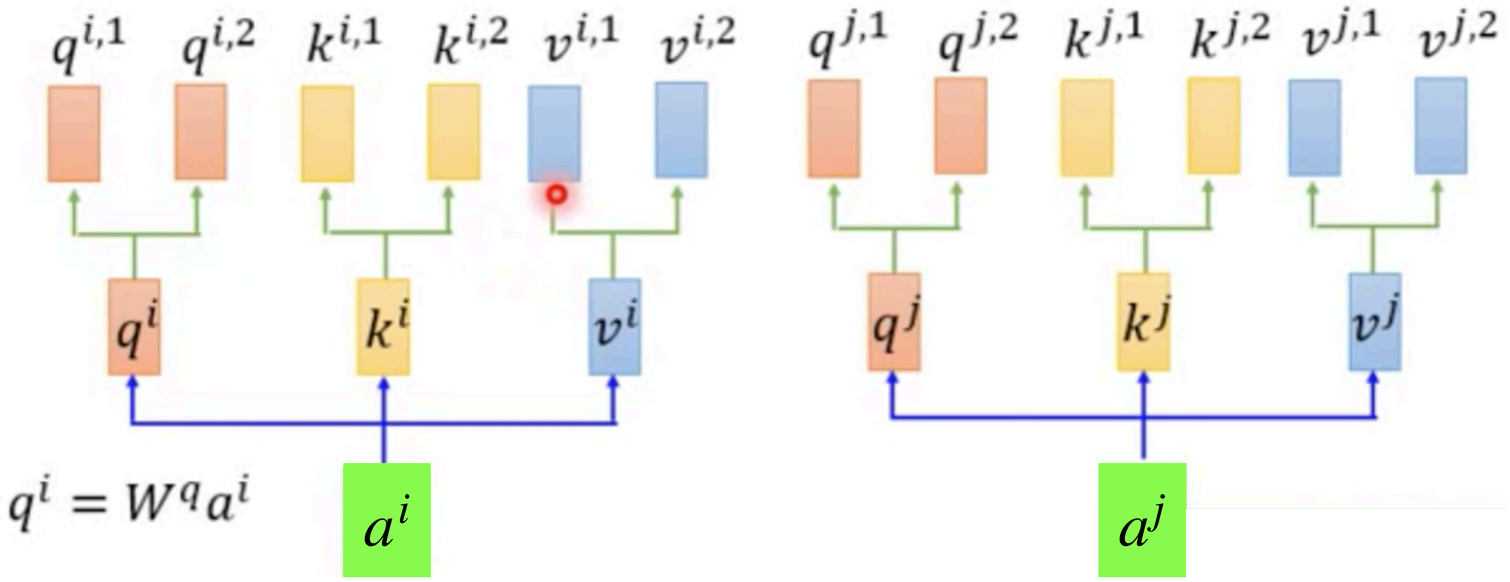
Similar to fully connected layer: consider whole sequence for each output.

Similar to CNN: shared weights, parallel computing.

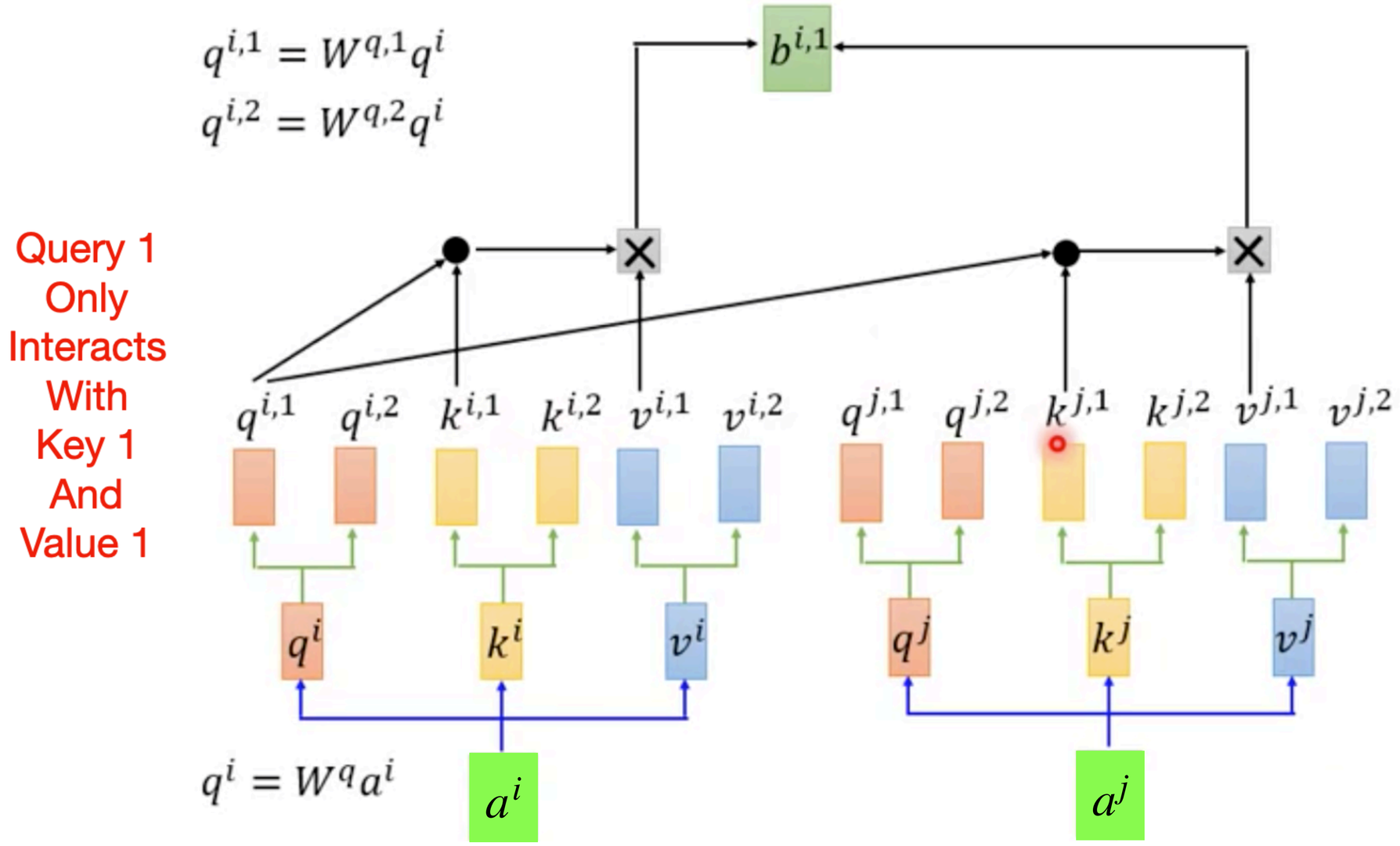
Similar to RNN: useful for sequence learning.

Multi-head Self-attention (2 heads as example)

$$q^{i,1} = W^{q,1} q^i$$
$$q^{i,2} = W^{q,2} q^i$$



Multi-head Self-attention (2 heads as example)



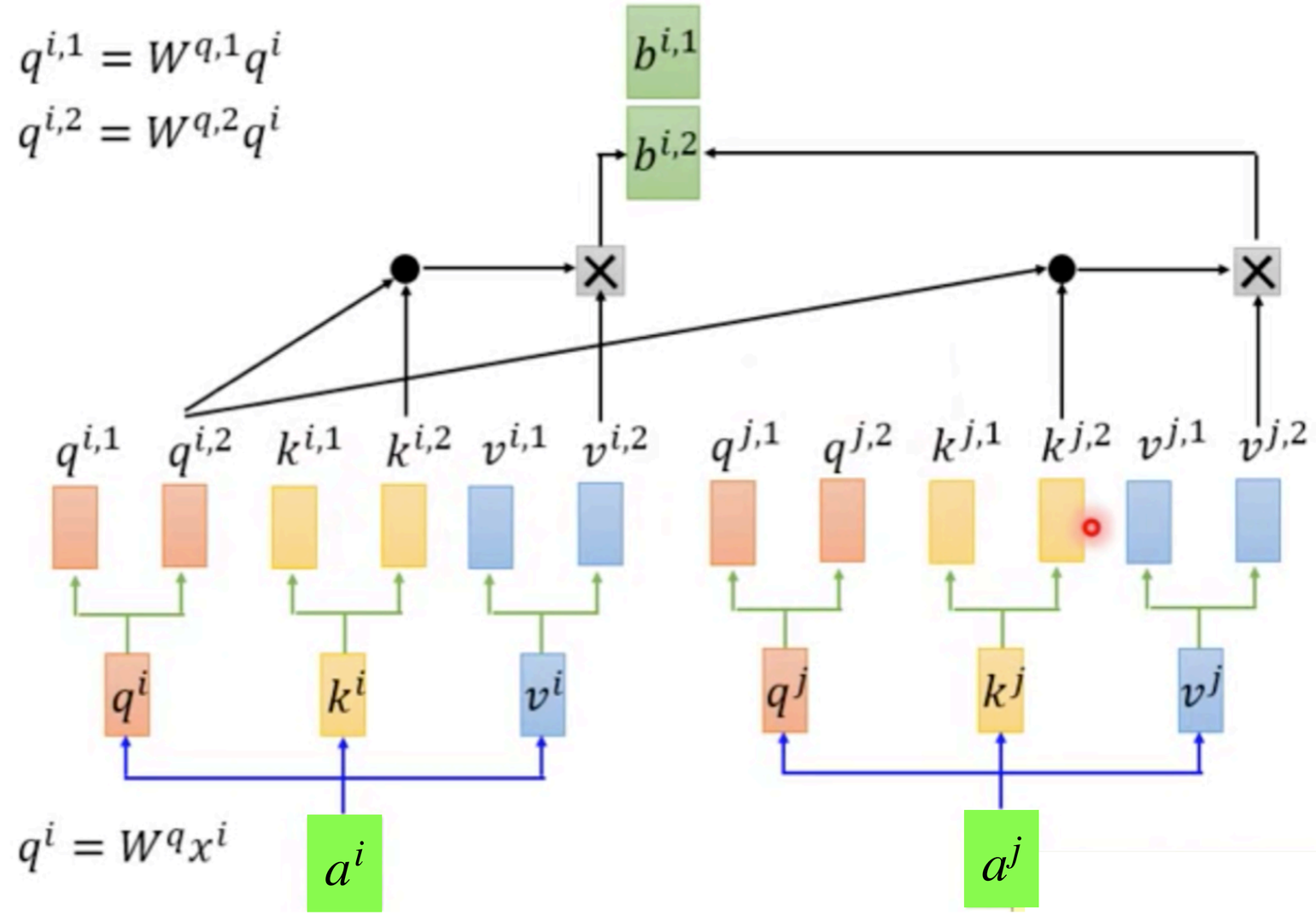
Query 1 Only Interacts With Key 1 And Value 1

Multi-head Self-attention (2 heads as example)

$q^{i,1} = W^{q,1} q^i$

$q^{i,2} = W^{q,2} q^i$

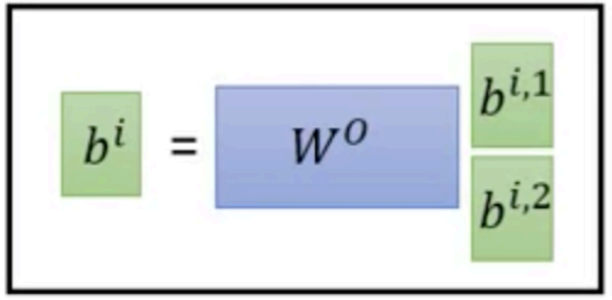
Query 2 Only Interacts With Key 2 And Value 2

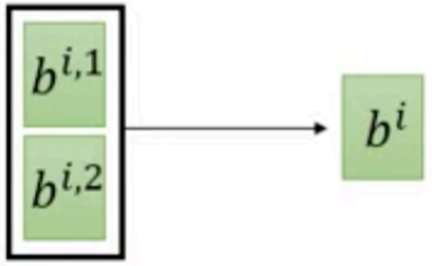


$q^i = W^q x^i$

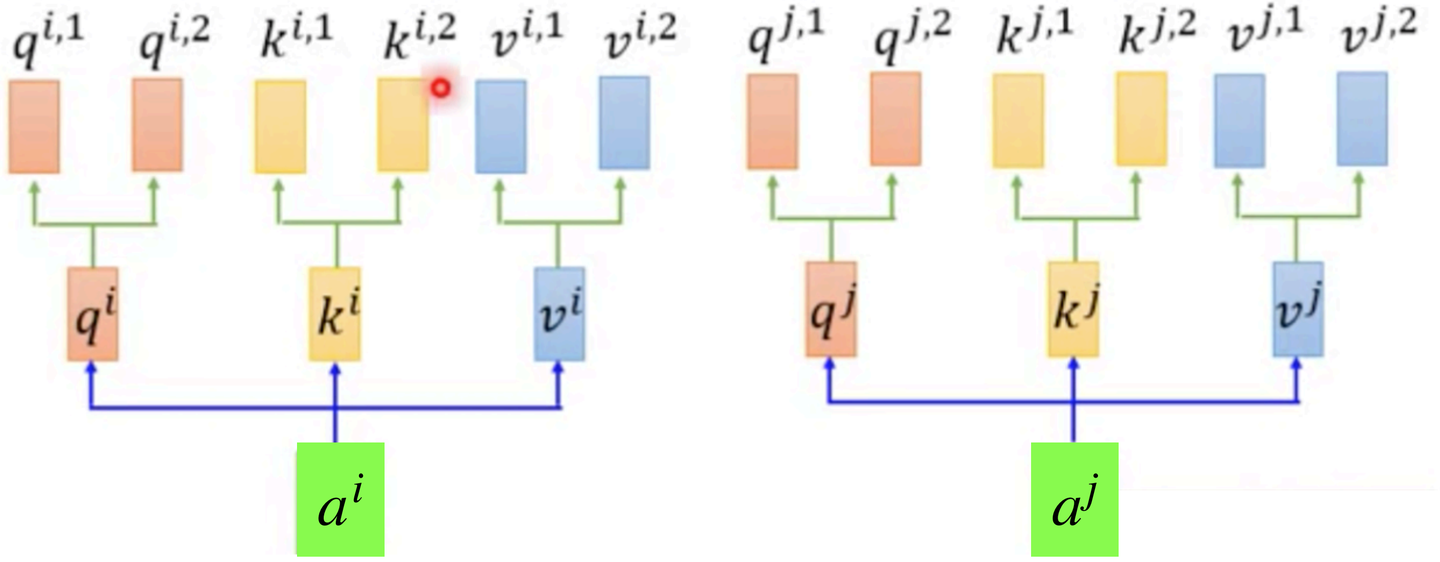
Multi-head Self-attention (2 heads as example)

Adjust Dimension of output b^i

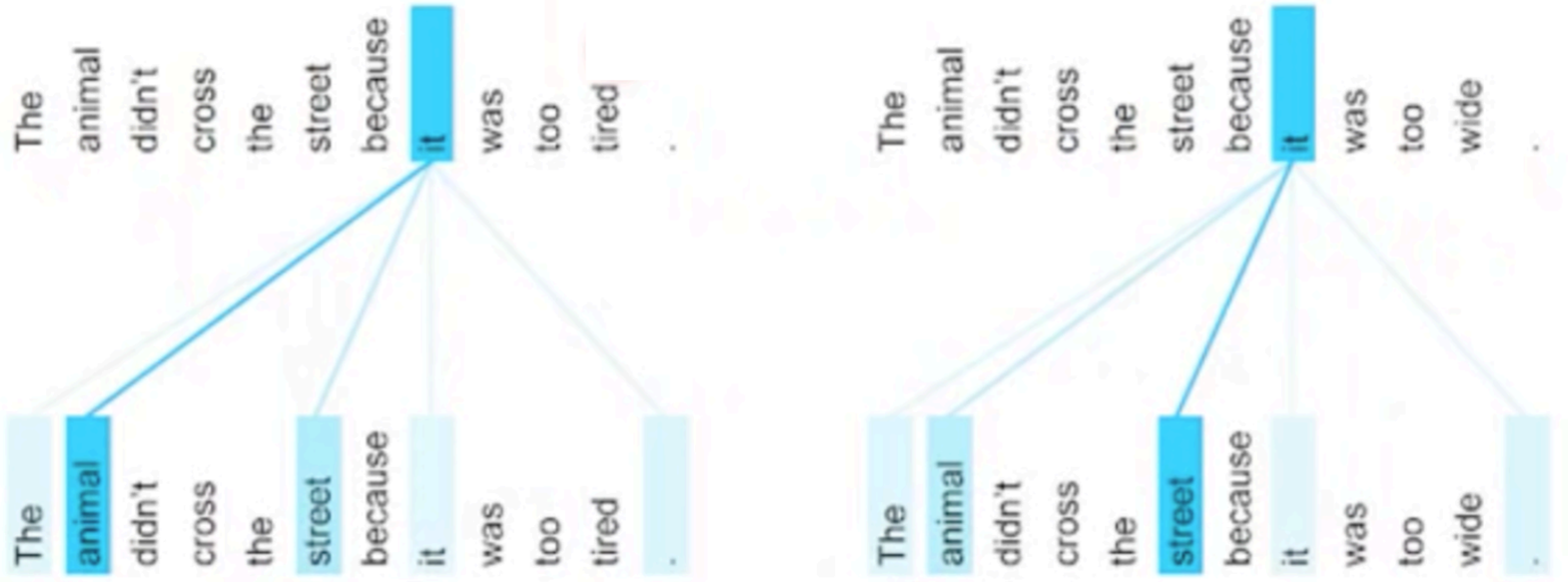




Benefit of multi-head: Different heads Use different Kinds of Attention (e.g., local attention, global attention, etc.)



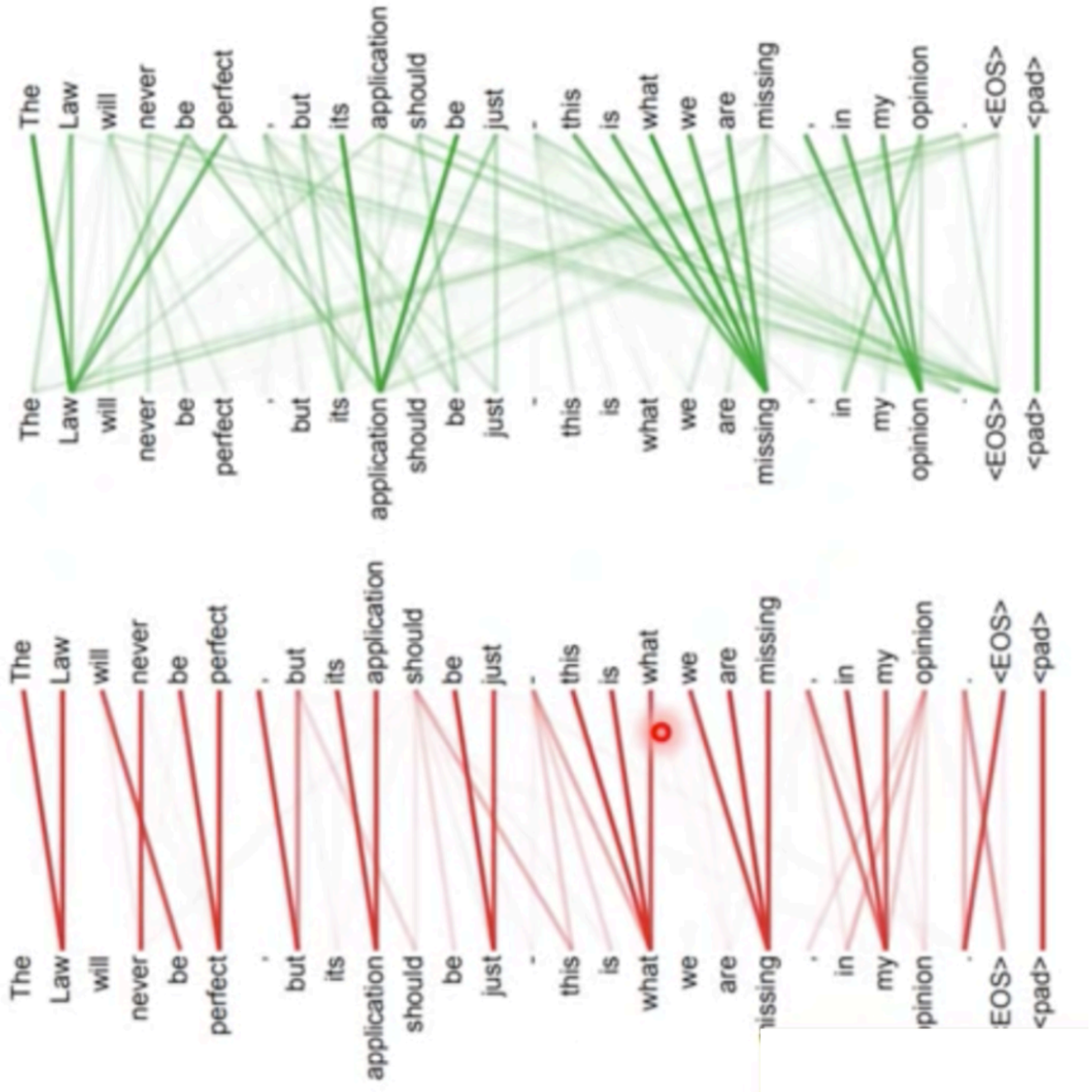
Attention Visualization



The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

https://ai.googleblog.com/2017/08/transformer-novel-neural-r

Visualization of Multi-Head Attention



Understand Self-Attention

Clearly, a smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That's where *self-attention* comes in. The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. This produces context-aware token representations. Consider an example sentence: "The train left the station on time." Now, consider one word in the sentence: station. What kind of station are we talking about? Could it be a radio station? Maybe the International Space Station? Let's figure it out algorithmically via self-attention (see figure 11.6).

Understand Self-Attention

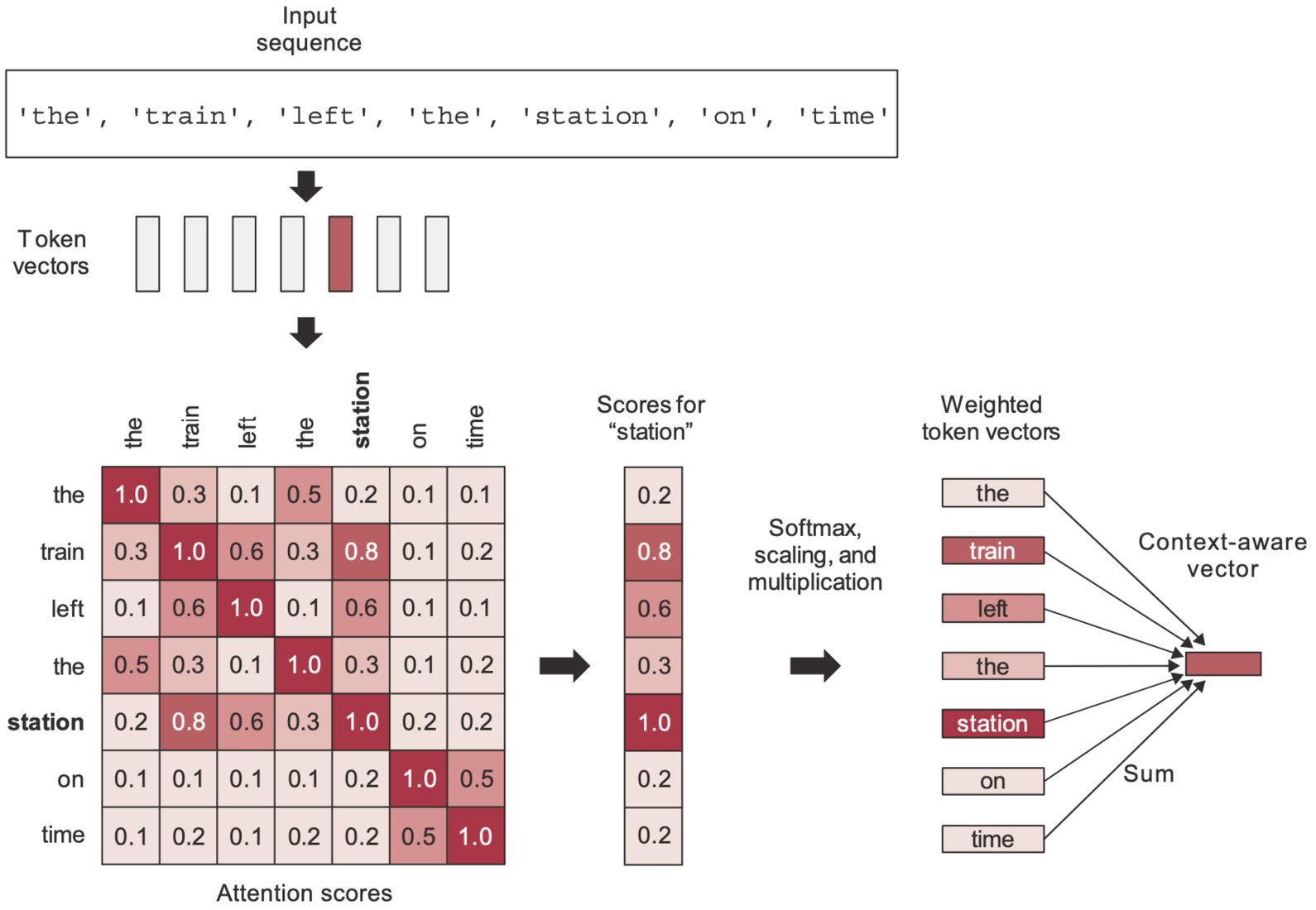


Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new "station" vector.

Understand Self-Attention

Of course, in practice you'd use a vectorized implementation. Keras has a built-in layer to handle it: the `MultiHeadAttention` layer. Here's how you would use it:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

Reading this, you're probably wondering

- Why are we passing the inputs to the layer *three* times? That seems redundant.
- What are these "multiple heads" we're referring to? That sounds intimidating—do they also grow back if you cut them?

Both of these questions have simple answers. Let's take a look.

Generalized Self-Attention: the query-key-value model

So far, we have only considered one input sequence. However, the Transformer architecture was originally developed for machine translation, where you have to deal with two input sequences: the source sequence you're currently translating (such as "How's the weather today?"), and the target sequence you're converting it to (such as "¿Qué tiempo hace hoy?"). A Transformer is a *sequence-to-sequence* model: it was designed to convert one sequence into another. You'll learn about sequence-to-sequence models in depth later in this chapter.

Generalized Self-Attention: the query-key-value model

Now let's take a step back. The self-attention mechanism as we've introduced it performs the following, schematically:

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
```

This means "for each token in `inputs` (A), compute how much the token is related to every token in `inputs` (B), and use these scores to weight a sum of tokens from `inputs` (C)." Crucially, there's nothing that requires A, B, and C to refer to the same input sequence. In the general case, you could be doing this with three different sequences. We'll call them "query," "keys," and "values." The operation becomes "for each element in the query, compute how much the element is related to every key, and use these scores to weight a sum of values":

```
outputs = sum(values * pairwise_scores(query, keys))
```

Multi-head attention

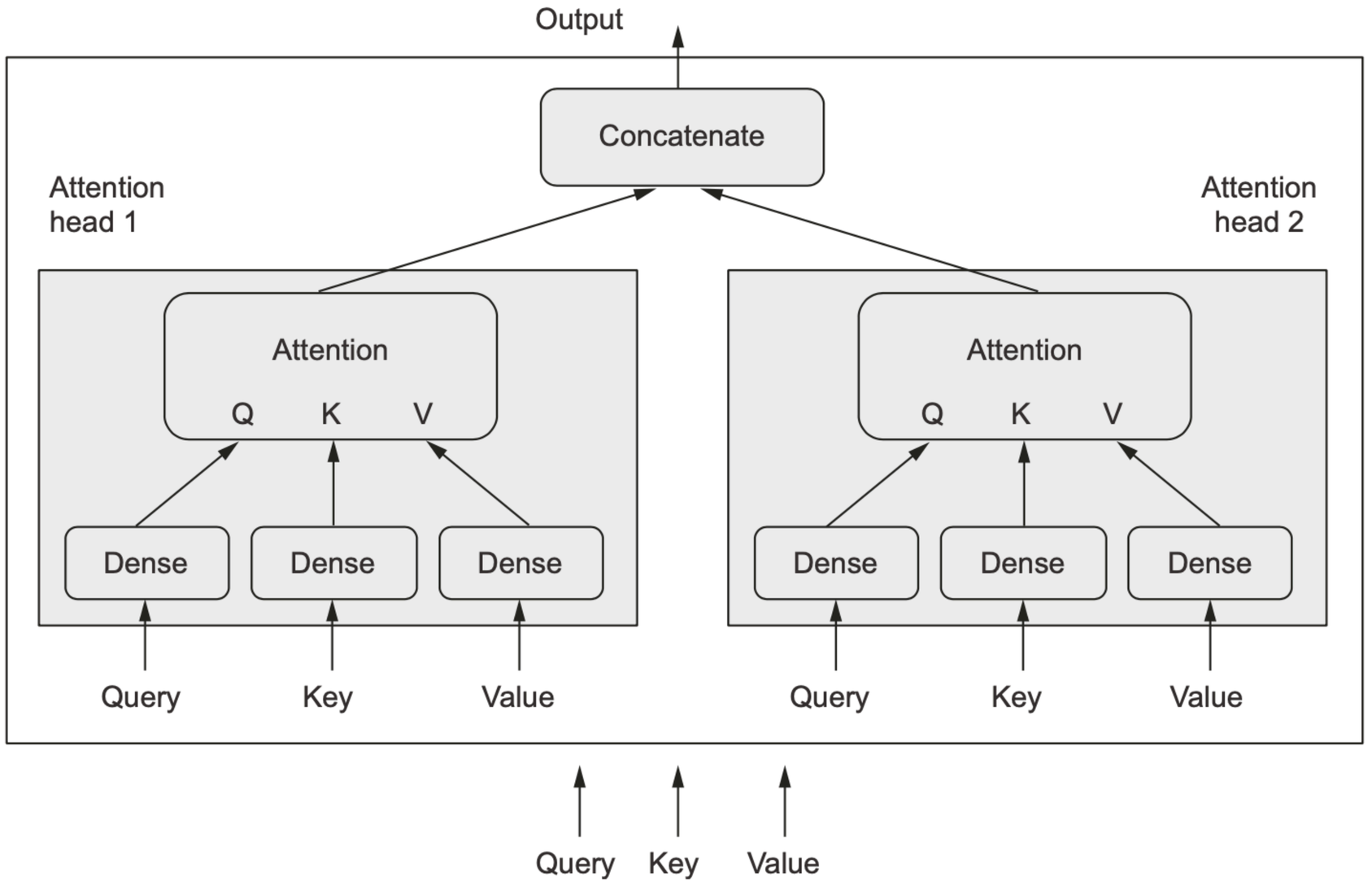


Figure 11.8 The `MultiHeadAttention` layer

The Transformer Encoder

If adding extra dense projections is so useful, why don't we also apply one or two to the output of the attention mechanism? Actually, that's a great idea—let's do that. And our model is starting to do a lot, so we might want to add residual connections to make sure we don't destroy any valuable information along the way—you learned in chapter 9 that they're a must for any sufficiently deep architecture. And there's another thing you learned in chapter 9: normalization layers are supposed to help gradients flow better during backpropagation. Let's add those too.

The Transformer Encoder

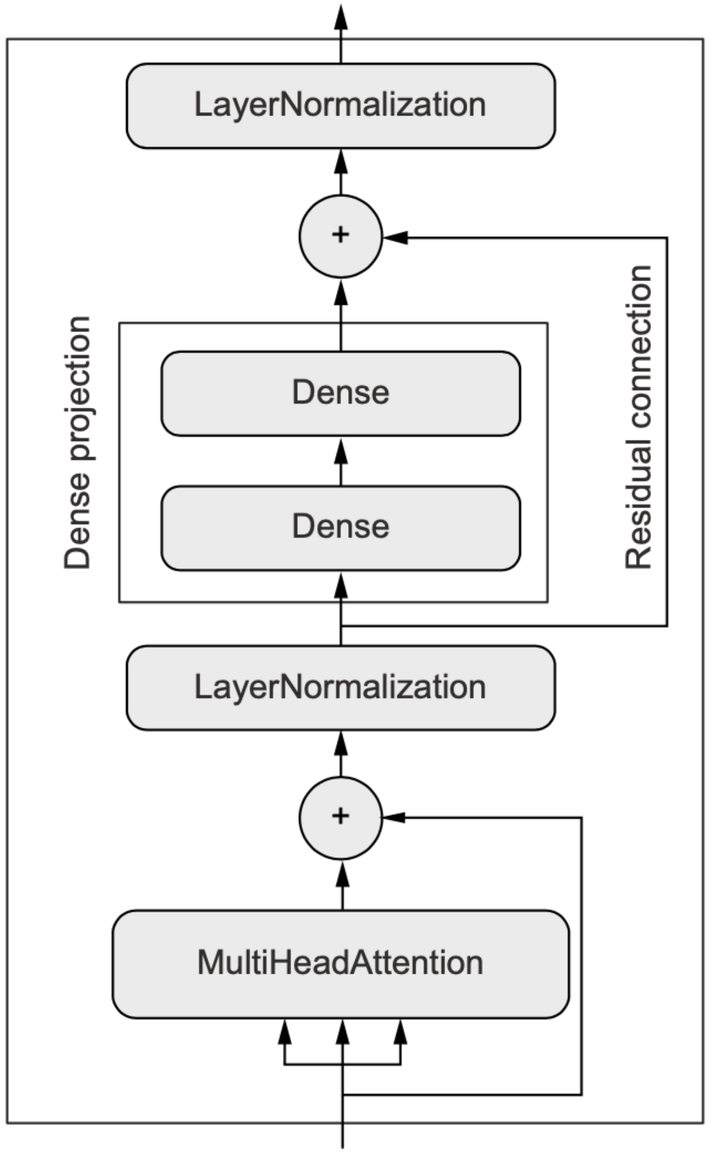


Figure 11.9 The `TransformerEncoder` chains a `MultiHeadAttention` layer with a dense projection and adds normalization as well as residual connections.

The Transformer Encoder

Crucially, the encoder part can be used for text classification—it's a very generic module that ingests a sequence and learns to turn it into a more useful representation. Let's implement a Transformer encoder and try it on the movie review sentiment classification task.

Quiz questions:

1. What is self-attention?

2. What is the architecture of the Transformer encoder?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Try 8: Transformer encoder

Listing 11.21 Transformer encoder implemented as a subclassed `Layer`

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
```

Size of the input token vectors

Size of the inner dense layer

Number of attention heads

Try 8: Transformer encoder

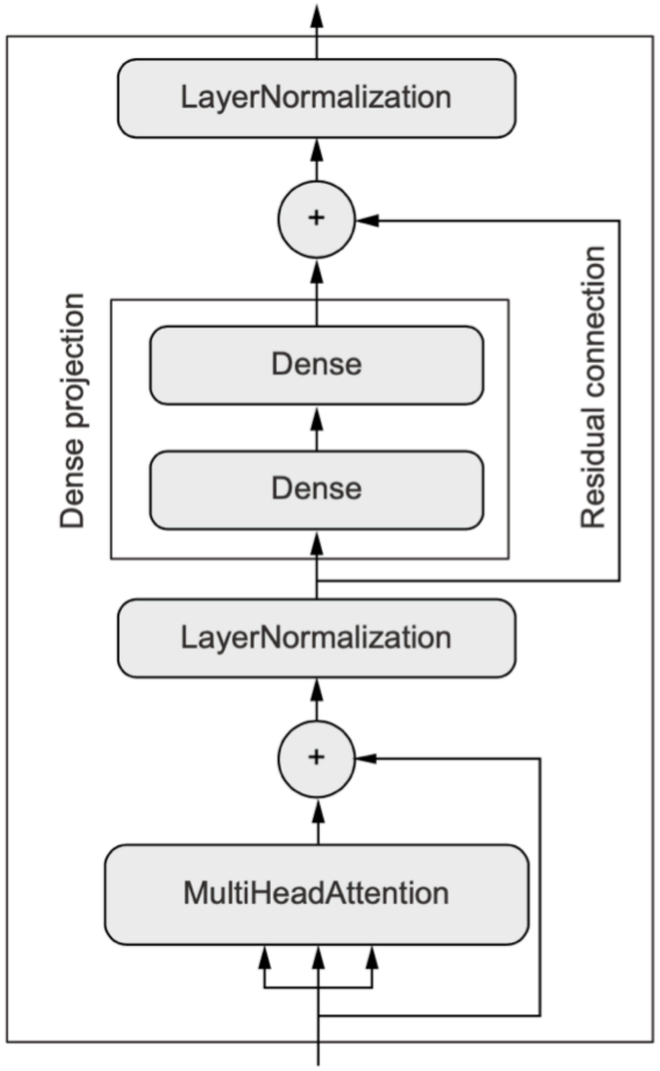
```
    self.layernorm_1 = layers.LayerNormalization()
    self.layernorm_2 = layers.LayerNormalization()

def call(self, inputs, mask=None):
    if mask is not None:
        mask = mask[:, tf.newaxis, :]
    attention_output = self.attention(
        inputs, inputs, attention_mask=mask)
    proj_input = self.layernorm_1(inputs + attention_output)
    proj_output = self.dense_proj(proj_input)
    return self.layernorm_2(proj_input + proj_output)

def get_config(self):
    config = super().get_config()
    config.update({
        "embed_dim": self.embed_dim,
        "num_heads": self.num_heads,
        "dense_dim": self.dense_dim,
    })
    return config
```

Computation goes in call().

The mask that will be generated by the Embedding layer will be 2D, but the attention layer expects to be 3D or 4D, so we expand its rank.

Implement serialization so we can save the model.

Try 8: Transformer encoder



Saving custom layers

When you write custom layers, make sure to implement the `get_config` method: this enables the layer to be reinstantiated from its config dict, which is useful during model saving and loading. The method should return a Python dict that contains the values of the constructor arguments used to create the layer.

All Keras layers can be serialized and deserialized as follows:

```
config = layer.get_config()
new_layer = layer.__class__.from_config(config)
```

The config does not contain weight values, so all weights in the layer get initialized from scratch.

For instance:

```
layer = PositionalEmbedding(sequence_length, input_dim, output_dim)
config = layer.get_config()
new_layer = PositionalEmbedding.from_config(config)
```

When saving a model that contains custom layers, the savefile will contain these config dicts. When loading the model from the file, you should provide the custom layer classes to the loading process, so that it can make sense of the config objects:

```
model = keras.models.load_model(
    filename, custom_objects={"PositionalEmbedding": PositionalEmbedding})
```

Try 8: Transformer encoder

You'll note that the normalization layers we're using here aren't `BatchNormalization` layers like those we've used before in image models. That's because `BatchNormalization` doesn't work well for sequence data. Instead, we're using the `LayerNormalization` layer, which normalizes each sequence independently from other sequences in the batch. Like this, in NumPy-like pseudocode:

Try 8: Transformer encoder

```
def layer_normalization(batch_of_sequences):
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1)
    return (batch_of_sequences - mean) / variance
```

Input shape: (batch_size, sequence_length, embedding_dim)

To compute mean and variance, we only pool data over the last axis (axis -1).

Compare to `BatchNormalization` (during training):

```
def batch_normalization(batch_of_images):
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

Input shape: (batch_size, height, width, channels)

Pool data over the batch axis (axis 0), which creates interactions between samples in a batch.

While `BatchNormalization` collects information from many samples to obtain accurate statistics for the feature means and variances, `LayerNormalization` pools data within each sequence separately, which is more appropriate for sequence data.

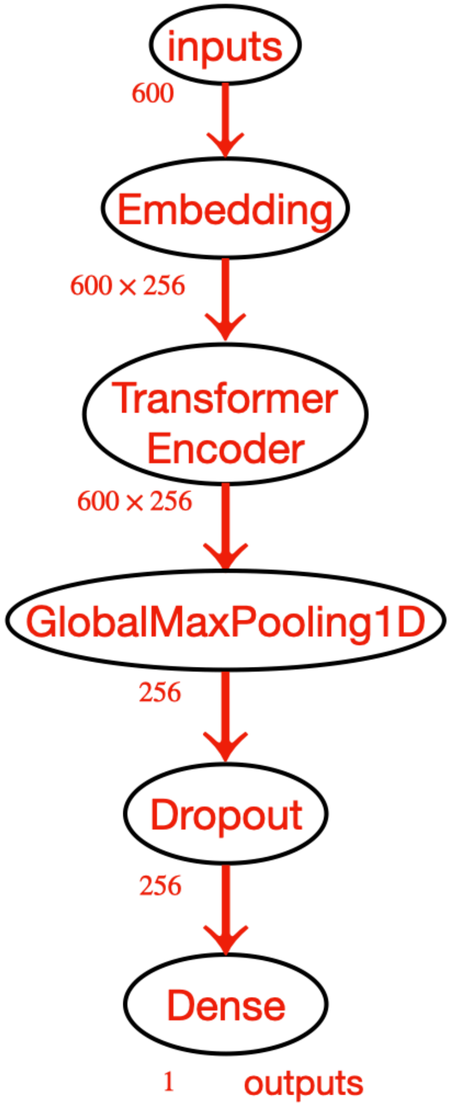
Try 8: Transformer encoder

Now that we've implemented our `TransformerEncoder`, we can use it to assemble a text-classification model similar to the GRU-based one you've seen previously.

Listing 11.22 Using the Transformer encoder for text classification

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Since TransformerEncoder returns full sequences, we need to reduce each sequence to a single vector for classification via a global pooling layer.



Try 8: Transformer encoder

Listing 11.23 Training and evaluating the Transformer encoder based model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
          callbacks=callbacks)

model = keras.models.load_model(
    "transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Provide the custom TransformerEncoder class to the model-loading process.

Test accuracy = 87.5%

Worse than "Try 1: Bag-of-words, unigram with binary encoding": Test accuracy = 89.2%

Worse than "Try 2: Bag-of-words, bigram with binary encoding": Test accuracy =90.4%

Worse than "Try 3: Bag-of-words, bigram with TF-IDF encoding": Test accuracy =89.8%

Better than "Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM": Test accuracy: 87%

Better than "Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM": Test accuracy =87%

Worse than "Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-directional LSTM": Test accuracy =88%

Comparable to "Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM": Test accuracy <= 88%

Issue with Try 8: Transformer encoder

Something wrong? What about position information?
(After all, it is a “sequence model”.)

	Word order awareness	Context awareness (cross-words interactions)
Bag-of-unigrams	No	No
Bag-of-bigrams	Very limited	No
RNN	Yes	No
Self-attention	No	Yes
Transformer	Yes	Yes

Figure 11.10 Features of different types of NLP models

Quiz question:

1. What are the pros and cons of the approach in try 8?

Roadmap of this lecture:

1. How to prepare text data
2. Text classification for IMDB
    - 2.1 Try 1: Bag-of-words approach, unigram with binary encoding
    - 2.2 Try 2: Bag-of-words approach, bigram with binary encoding
    - 2.3 Try 3: Bag-of-words approach, bigram with TF-IDF encoding
    - 2.4 Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM
    - 2.5 Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM
    - 2.6 Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-direc LSTM
    - 2.7 Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM
    - 2.8 Attention and Transformer
    - 2.9 Try 8: Transformer encoder
    - 2.10 Try 9: Transformer encoder with positional embedding

Using Positional Encoding to re-inject order information

The idea behind positional encoding is very simple: to give the model access to word-order information, we're going to add the word's position in the sentence to each word embedding. Our input word embeddings will have two components: the usual word vector, which represents the word independently of any specific context, and a position vector, which represents the position of the word in the current sentence. Hopefully, the model will then figure out how to best leverage this additional information.

The simplest scheme you could come up with would be to concatenate the word's position to its embedding vector. You'd add a "position" axis to the vector and fill it with 0 for the first word in the sequence, 1 for the second, and so on.

That may not be ideal, however, because the positions can potentially be very large integers, which will disrupt the range of values in the embedding vector. As you know, neural networks don't like very large input values, or discrete input distributions.

Using Positional Encoding to re-inject order information

The original "Attention is all you need" paper used an interesting trick to encode word positions: it added to the word embeddings a vector containing values in the range `[-1, 1]` that varied cyclically depending on the position (it used cosine functions to achieve this). This trick offers a way to uniquely characterize any integer in a large range via a vector of small values. It's clever, but it's not what we're going to use in our case. We'll do something simpler and more effective: we'll learn position-embedding vectors the same way we learn to embed word indices. We'll then proceed to add our position embeddings to the corresponding word embeddings, to obtain a position-aware word embedding. This technique is called "positional embedding." Let's implement it.

Using Positional Encoding to re-inject order information

Listing 11.24 Implementing positional embedding as a subclassed layer

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

A downside of position embeddings is that the sequence length needs to be known in advance.

Prepare an Embedding layer for the token indices.

And another one for the token positions

Add both embedding vectors together.

Like the Embedding layer, this layer should be able to generate a mask so we can ignore padding 0s in the inputs. The compute_mask method will called automatically by the framework, and the mask will get propagated to the next layer.

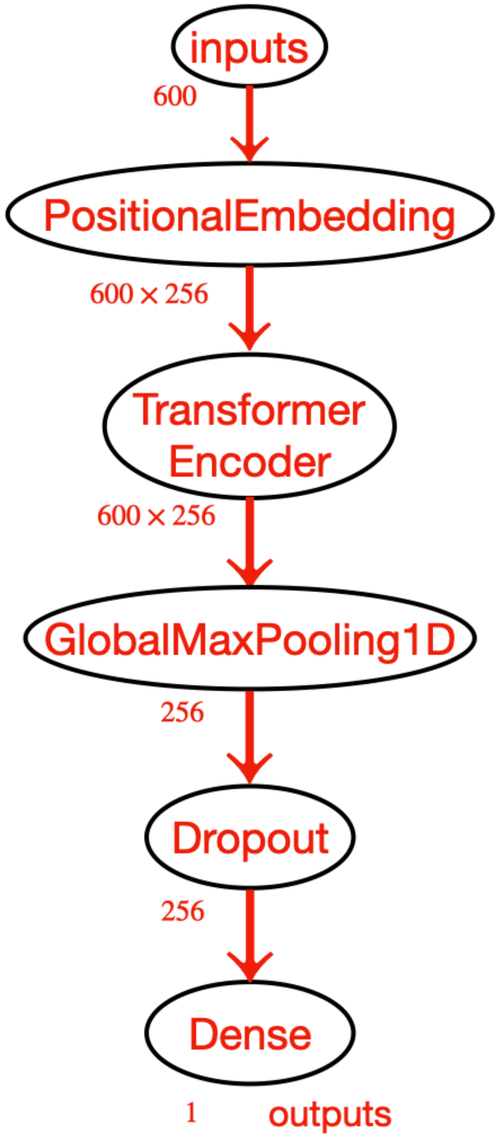
Implement serialization so we can save the model.

Try 9: Transformer encoder with positional embedding

Listing 11.25 Combining the Transformer encoder with positional embedding

```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
     callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                    "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Look here!



Try 9: Transformer encoder with positional embedding

Test accuracy = 88.3%

Worse than "Try 1: Bag-of-words, unigram with binary encoding": Test accuracy = 89.2%

Worse than "Try 2: Bag-of-words, bigram with binary encoding": Test accuracy =90.4%

Worse than "Try 3: Bag-of-words, bigram with TF-IDF encoding": Test accuracy =89.8%

Better than "Try 4: Sequence approach, one-hot encoding, Bi-directional LSTM": Test accuracy: 87%

Better than "Try 5: Sequence approach, learn word embedding with the Embedding layer, Bi-directional LSTM": Test accuracy =87%

Better than "Try 6: Sequence approach, learn word embedding with Embedding layer (with masking enabled), Bi-directional LSTM": Test accuracy =88%

Better than "Try 7: Sequence approach, pre-trained word embedding, Bi-direc LSTM": Test accuracy <= 88%

Better than "Try 8: Transformer encoder": Test accuracy = 87.5%

When to use sequence models over bag-of-words models

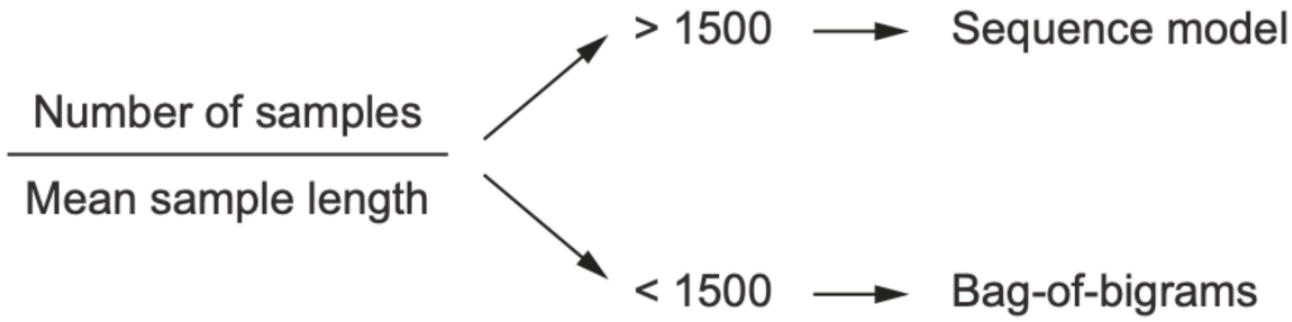


Figure 11.11 A simple heuristic for selecting a text-classification model: the ratio between the number of training samples and the mean number of words per sample

Quiz question:

1. What are the pros and cons of the approach in try 9?

Roadmap of this lecture (continued):

3. Sequence-to-Sequence Learning

4. Seq2Seq for Machine Translation

4.1 Try 1 for Machine Translation: RNN

4.2 Try 2 for Machine Translation: Transformer

Sequence-to-Sequence Learning

A sequence-to-sequence model takes a sequence as input (often a sentence or paragraph) and translates it into a different sequence. This is the task at the heart of many of the most successful applications of NLP:

- *Machine translation*—Convert a paragraph in a source language to its equivalent in a target language.
- *Text summarization*—Convert a long document to a shorter version that retains the most important information.
- *Question answering*—Convert an input question into its answer.
- *Chatbots*—Convert a dialogue prompt into a reply to this prompt, or convert the history of a conversation into the next reply in the conversation.
- *Text generation*—Convert a text prompt into a paragraph that completes the prompt.
- Etc.

Sequence-to-Sequence Learning

The general template behind sequence-to-sequence models is described in figure 11.12. During training,

- An *encoder* model turns the source sequence into an intermediate representation.
- A *decoder* is trained to predict the next token `i` in the target sequence by looking at both previous tokens (`0` to `i - 1`) and the encoded source sequence.

During inference, we don't have access to the target sequence—we're trying to predict it from scratch. We'll have to generate it one token at a time:

1. We obtain the encoded source sequence from the encoder.
2. The decoder starts by looking at the encoded source sequence as well as an initial "seed" token (such as the string `"[start]"`), and uses them to predict the first real token in the sequence.
3. The predicted sequence so far is fed back into the decoder, which generates the next token, and so on, until it generates a stop token (such as the string `"[end]"`).

Sequence-to-Sequence Learning

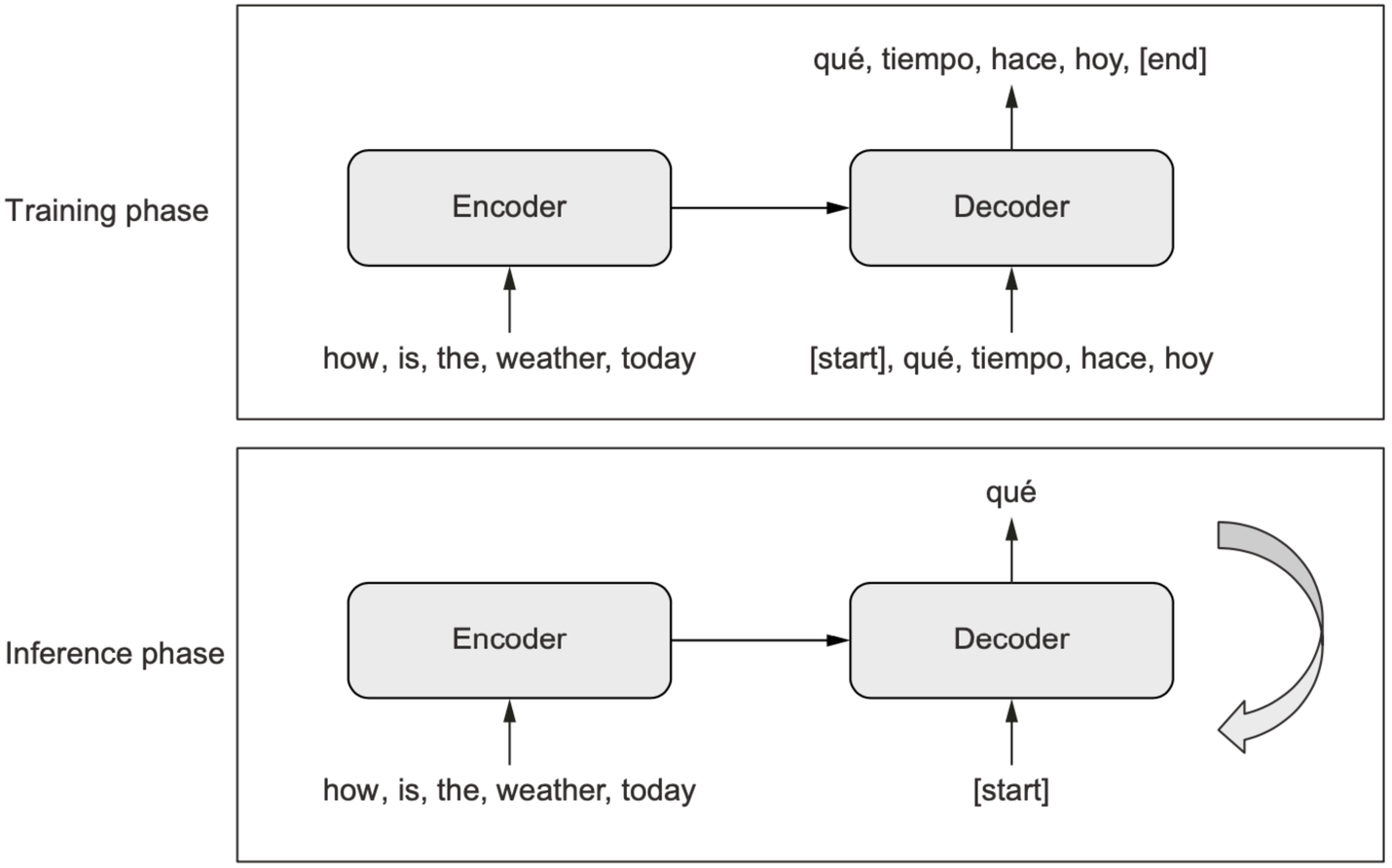


Figure 11.12 Sequence-to-sequence learning: the source sequence is processed by the encoder and is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.

Quiz questions:

1. What is sequence-to-sequence (seq2seq) learning?
2. What are some important applications of seq2seq learning?

Roadmap of this lecture (continued):

3. Sequence-to-Sequence Learning

4. Seq2Seq for Machine Translation

4.1 Try 1 for Machine Translation: RNN

4.2 Try 2 for Machine Translation: Transformer

A Machine Translation Example

Machine translation is precisely what Transformer was developed for! We'll start with a recurrent sequence model, and we'll follow up with the full Transformer architecture.

We'll be working with an English-to-Spanish translation dataset available at www.manythings.org/anki/. Let's download it:

```
!wget http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip
!unzip -q spa-eng.zip
```

A Machine Translation Example

The text file contains one example per line: an English sentence, followed by a tab character, followed by the corresponding Spanish sentence. Let's parse this file.

```
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []

for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

Iterate over the lines in the file.

Each line contains an English phrase and its Spanish translation, tab-separated.

We prepend "[start]" and append "[end]" to the Spanish sentence, to match the template from figure 11.12.

Our `text_pairs` look like this:

```
>>> import random
>>> print(random.choice(text_pairs))
("Soccer is more popular than tennis.",
 "[start] El fútbol es más popular que el tenis. [end]")
```

A Machine Translation Example

Let's shuffle them and split them into the usual training, validation, and test sets:

```
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

Training set: 70%
Validation set: 15%
Test set: 15%

A Machine Translation Example

Next, let's prepare two separate `TextVectorization` layers: one for English and one for Spanish. We're going to need to customize the way strings are preprocessed:

- We need to preserve the `"[start]"` and `"[end]"` tokens that we've inserted. By default, the characters `[` and `]` would be stripped, but we want to keep them around so we can tell apart the word "start" and the start token `"[start]"`.
- Punctuation is different from language to language! In the Spanish `TextVectorization` layer, if we're going to strip punctuation characters, we need to also strip the character `¿`.

Note that for a non-toy translation model, we would treat punctuation characters as separate tokens rather than stripping them, since we would want to be able to generate correctly punctuated sentences. In our case, for simplicity, we'll get rid of all punctuation.

Prepare TextVectorization layers for English and for Spanish

Listing 11.26 Vectorizing the English and Spanish text pairs

```
import tensorflow as tf
import string
import re

strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")

vocab_size = 15000
sequence_length = 20

source_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
target_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

Prepare a custom string standardization function for the Spanish TextVectorization layer: it preserves [and] but strips ¿ (as well as all other characters from strings.punctuation).

To keep things simple, we'll only look at the top 15,000 words in each language, and we'll restrict sentences to 20 words.

The English layer

The Spanish layer

Generate Spanish sentences that have one extra token, since we'll need to offset the sentence by one step during training.

Learn the vocabulary of each language.

A Machine Translation Example

Finally, we can turn our data into a `tf.data` pipeline. We want it to return a tuple `(inputs, target)` where `inputs` is a dict with two keys, "encoder_inputs" (the English sentence) and "decoder_inputs" (the Spanish sentence), and `target` is the Spanish sentence offset by one step ahead.

A Machine Translation Example

Listing 11.27 Preparing datasets for the translation task

```
batch_size = 64

def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({
        "english": eng,
        "spanish": spa[:, :-1],
    }, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

The input Spanish sentence doesn't include the last token to keep inputs and targets at the same length.

The target Spanish sentence is one step ahead. Both are still the same length (20 words).

Use in-memory caching to speed up preprocessing.

A Machine Translation Example

Here's what our dataset outputs look like:

```
>>> for inputs, targets in train_ds.take(1):
>>>     print(f"inputs['english'].shape: {inputs['english'].shape}")
>>>     print(f"inputs['spanish'].shape: {inputs['spanish'].shape}")
>>>     print(f"targets.shape: {targets.shape}")
inputs["encoder_inputs"].shape: (64, 20)
inputs["decoder_inputs"].shape: (64, 20)
targets.shape: (64, 20)
```

Roadmap of this lecture (continued):

3. Sequence-to-Sequence Learning

4. Seq2Seq for Machine Translation

4.1 Try 1 for Machine Translation: RNN

4.2 Try 2 for Machine Translation: Transformer

Try 1 for Machine Translation: RNN

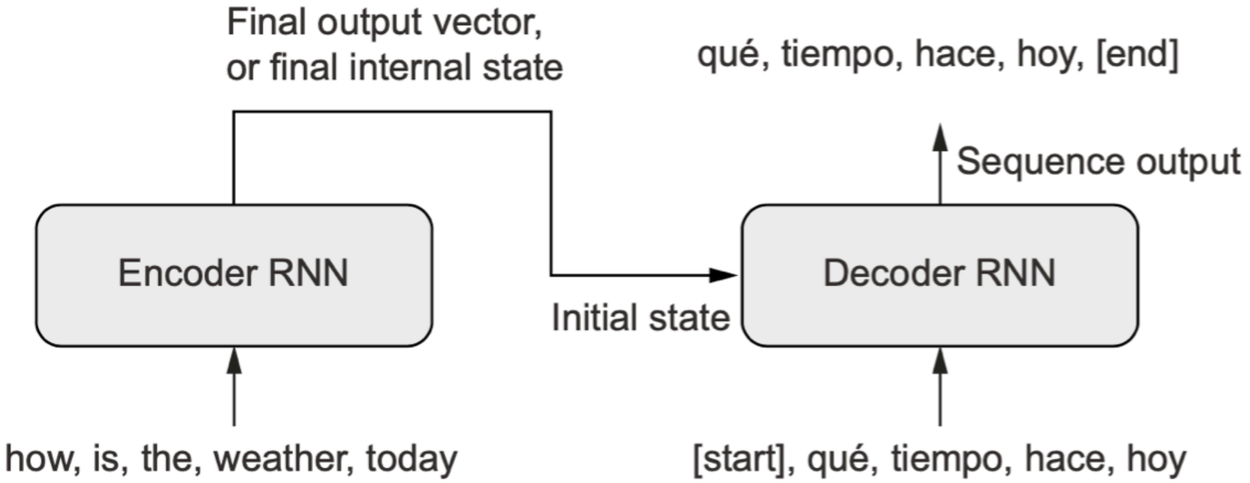


Figure 11.13 A sequence-to-sequence RNN: an RNN encoder is used to produce a vector that encodes the entire source sequence, which is used as the initial state for an RNN decoder.

Try 1 for Machine Translation: RNN

Listing 11.28 GRU-based encoder

```
from tensorflow import keras
from tensorflow.keras import layers

embed_dim = 256
latent_dim = 1024

source = keras.Input(shape=(None,), dtype="int64", name="english")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(
    layers.GRU(latent_dim), merge_mode="sum")(x)
```

Don't forget masking: it's critical in this setup.

The English source sentence goes here. Specifying the name of the input enables us to fit() the model with a dict of inputs.

Our encoded source sentence is the last output of a bidirectional GRU.

Next, let's add the decoder—a simple GRU layer that takes as its initial state the encoded source sentence. On top of it, we add a `Dense` layer that produces for each output step a probability distribution over the Spanish vocabulary.

Listing 11.29 GRU-based decoder and the end-to-end model

The Spanish target sentence goes here.

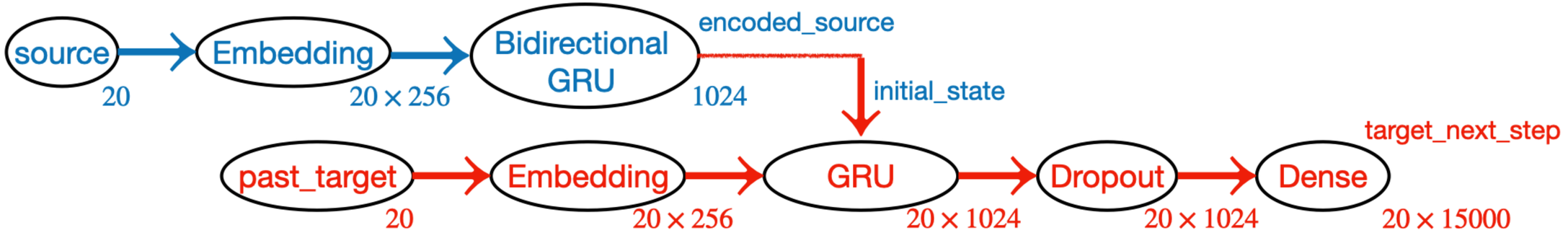
Don't forget masking.

```
past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
x = decoder_gru(x, initial_state=encoded_source)
x = layers.Dropout(0.5)(x)
target_next_step = layers.Dense(vocab_size, activation="softmax")(x)
seq2seq_rnn = keras.Model([source, past_target], target_next_step)
```

Predicts the next token

The encoded source sentence serves as the initial state of the decoder GRU.

End-to-end model: maps the source sentence and the target sentence to the target sentence one step in the future



Try 1 for Machine Translation: RNN

During training, the decoder takes as input the entire target sequence, but thanks to the step-by-step nature of RNNs, it only looks at tokens 0…N in the input to predict token N in the output (which corresponds to the next token in the sequence, since the output is intended to be offset by one step). This means we only use information from the past to predict the future, as we should; otherwise we'd be cheating, and our model would not work at inference time.

Let's start training.

Listing 11.30 Training our recurrent sequence-to-sequence model

```
seq2seq_rnn.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds)
```

Try 1 for Machine Translation: RNN

We picked accuracy as a crude way to monitor validation-set performance during training. We get to 64% accuracy: on average, the model predicts the next word in the Spanish sentence correctly 64% of the time. However, in practice, next-token accuracy isn't a great metric for machine translation models, in particular because it makes the assumption that the correct target tokens from 0 to *N* are already known when predicting token *N*+1. In reality, during inference, you're generating the target sentence from scratch, and you can't rely on previously generated tokens being 100% correct. If you work on a real-world machine translation system, you will likely use "BLEU scores" to evaluate your models—a metric that looks at entire generated sequences and that seems to correlate well with human perception of translation quality.

Try 1 for Machine Translation: RNN

At last, let's use our model for inference. We'll pick a few sentences in the test set and check how our model translates them. We'll start from the seed token, `"[start]"`, and feed it into the decoder model, together with the encoded English source sentence. We'll retrieve a next-token prediction, and we'll re-inject it into the decoder repeatedly, sampling one new target token at each iteration, until we get to `"[end]"` or reach the maximum sentence length.

Try 1 for Machine Translation: RNN

Listing 11.31 Translating new sentences with our RNN encoder and decoder

Prepare a dict to convert token index predictions to string tokens.

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

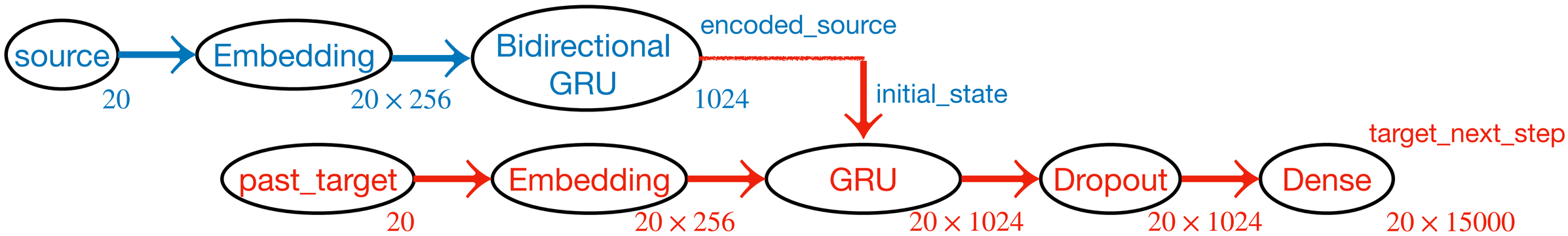
Seed token

Sample the next token.

Exit condition: either hit max length or sample a stop character

Convert the next token prediction to a string and append it to the generated sentence.

seq2seq_rnn model:



Try 1 for Machine Translation: RNN

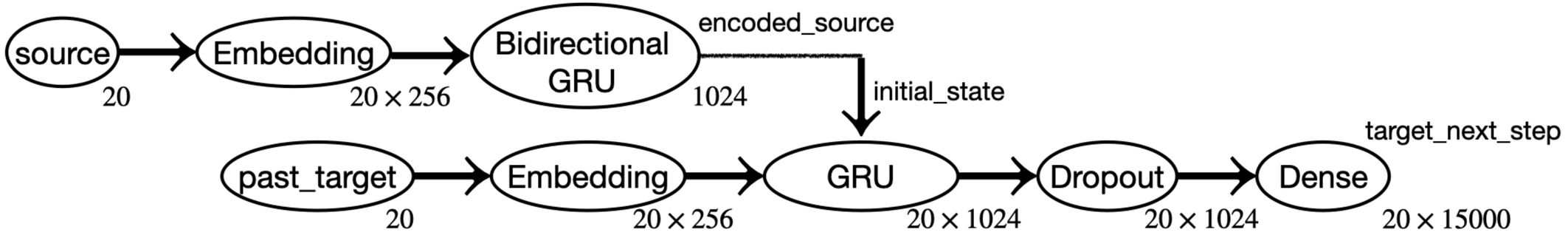
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

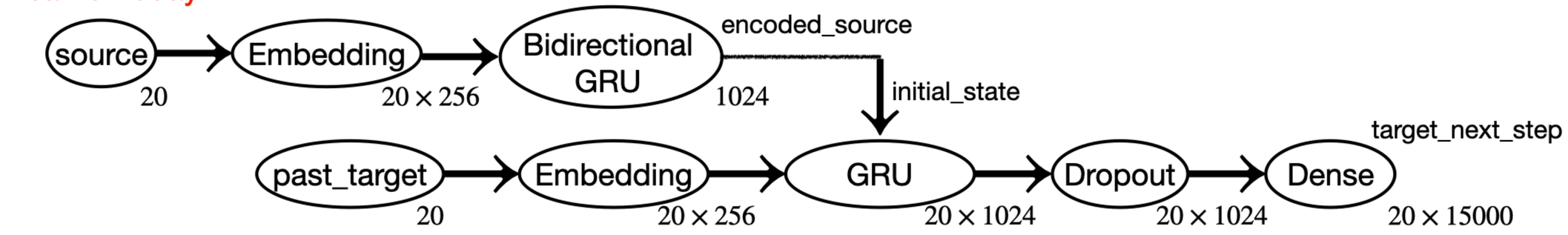
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

how is the weather today



Try 1 for Machine Translation: RNN

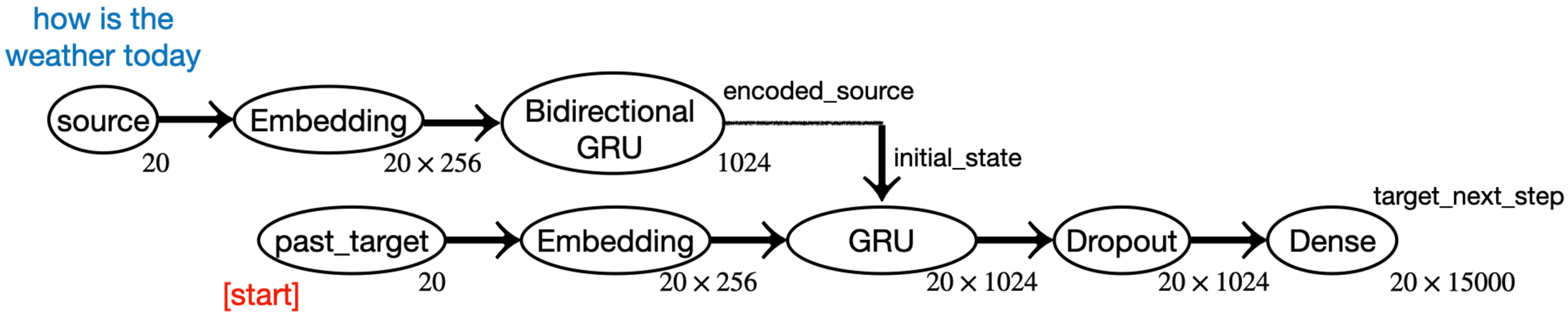
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

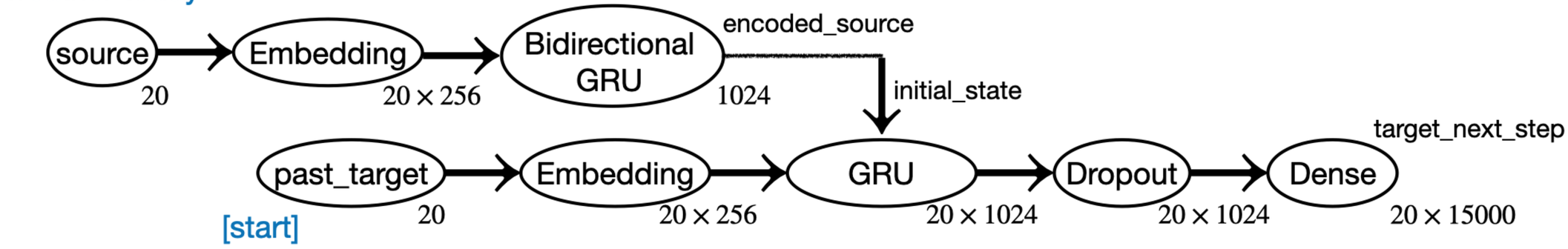
Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 0$

how is the weather today



Try 1 for Machine Translation: RNN

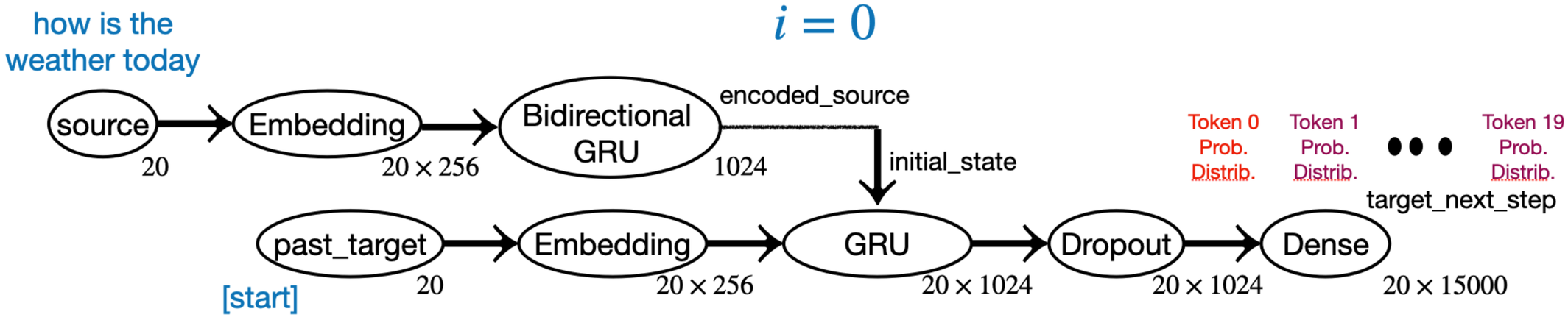
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

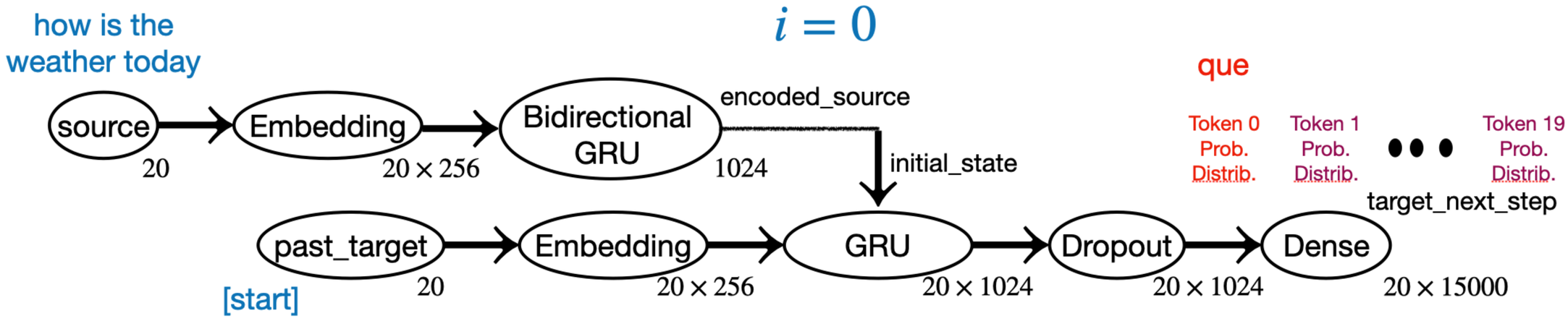
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

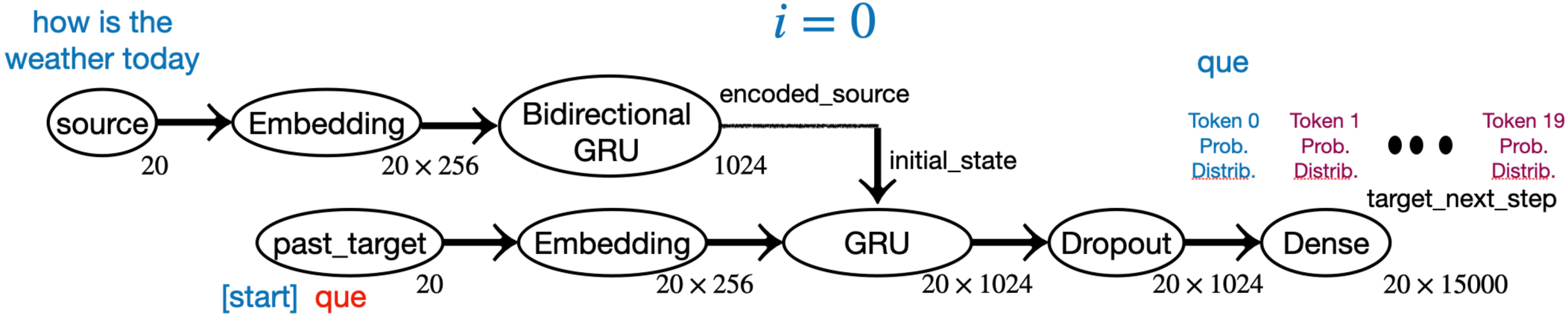
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

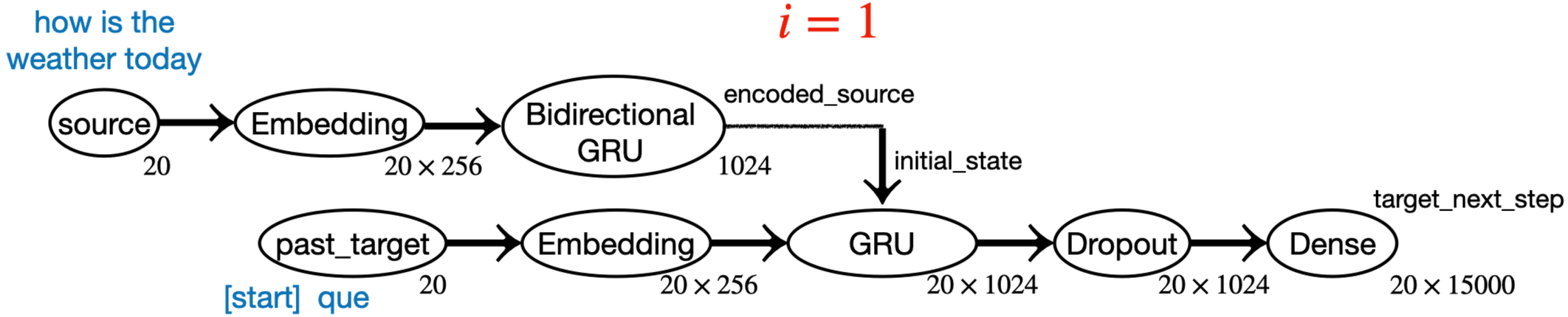
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 1$



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

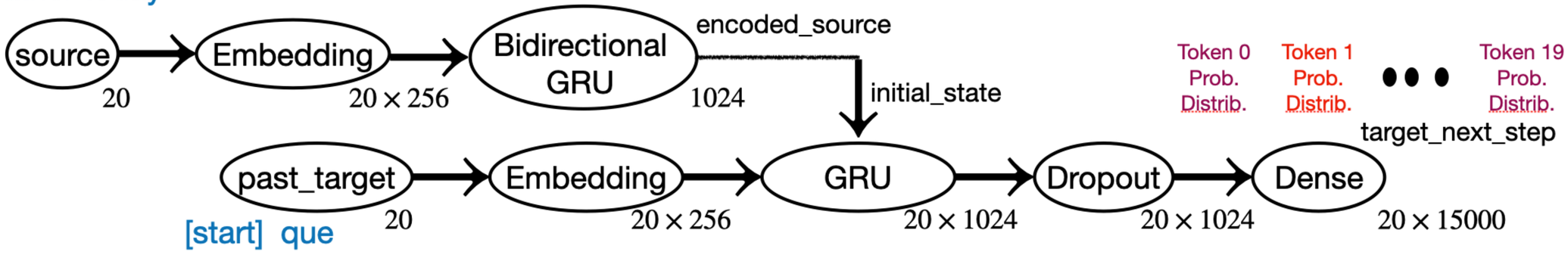
Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 1$

how is the weather today



Try 1 for Machine Translation: RNN

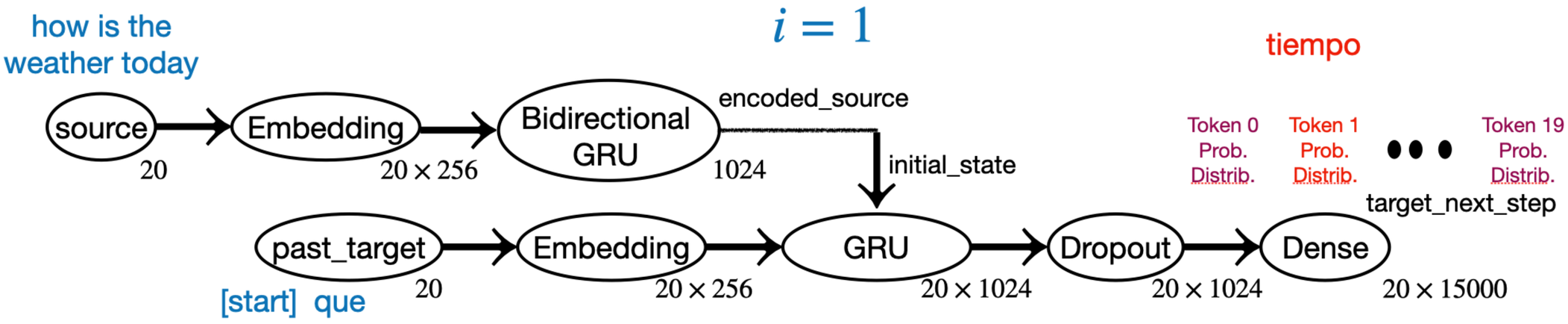
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

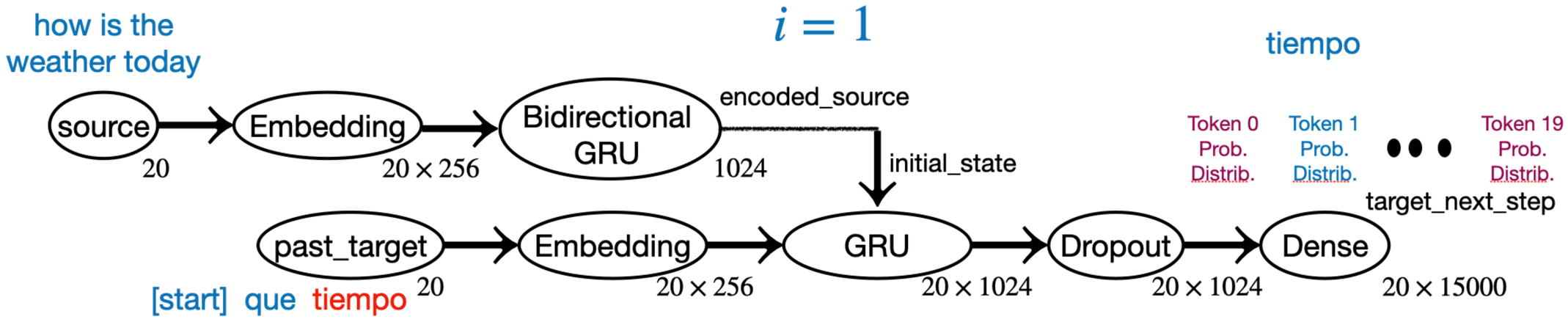
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

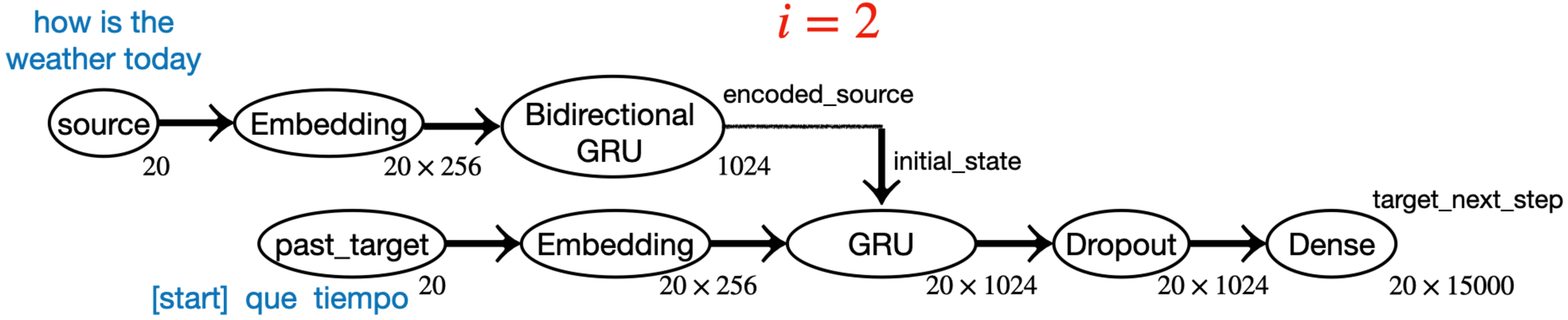
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 2$



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

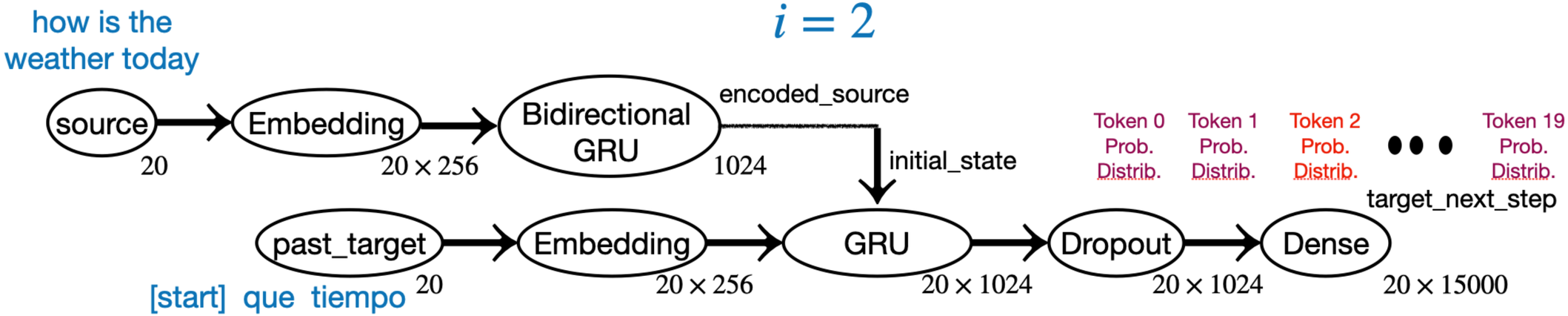
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 2$



Try 1 for Machine Translation: RNN

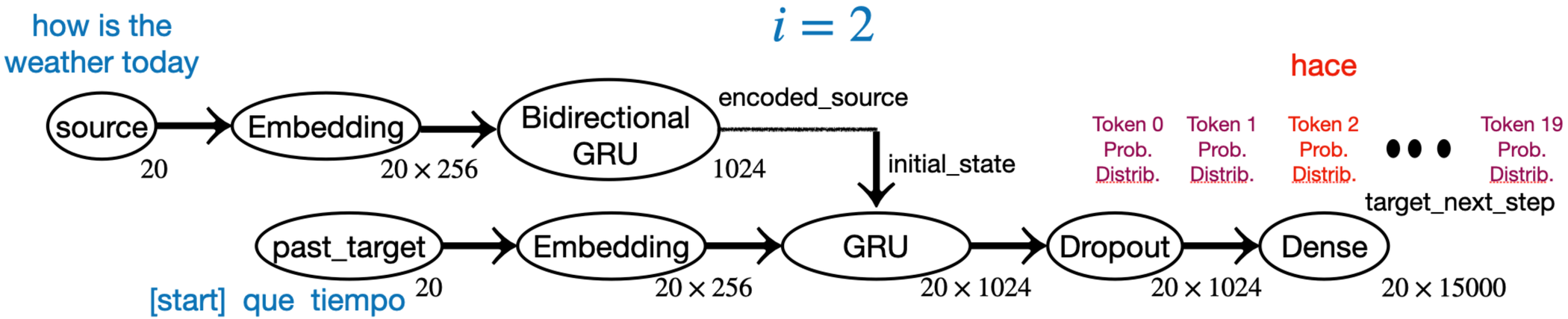
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

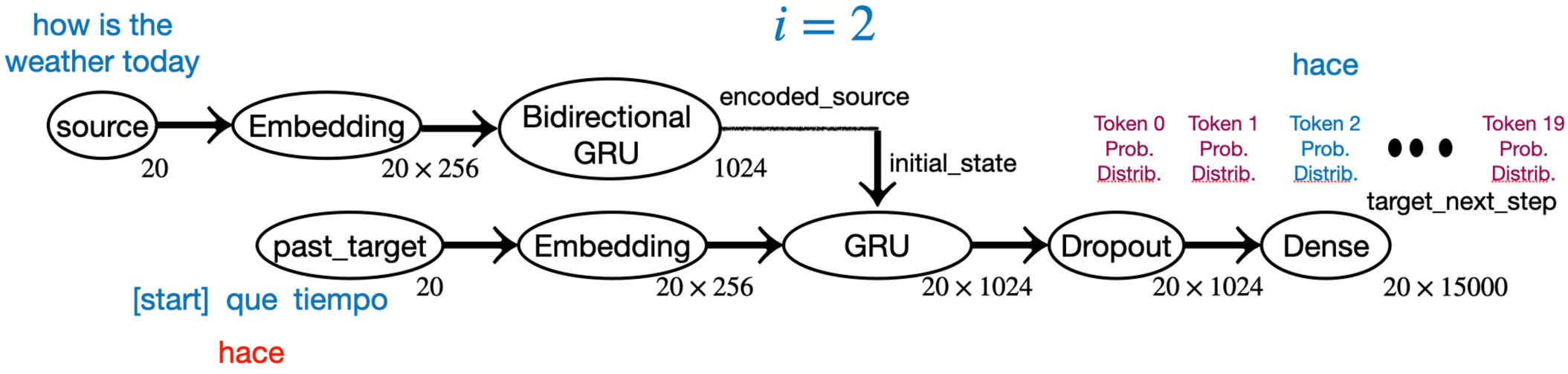
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

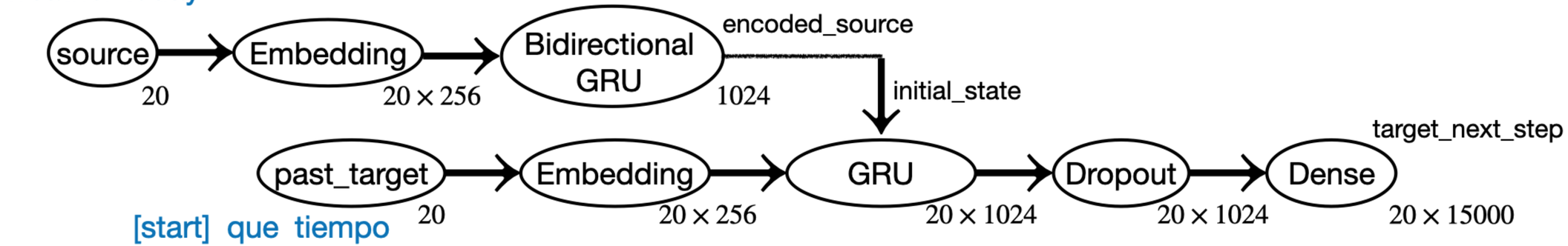
Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 3$

how is the weather today



[start] que tiempo hace

Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

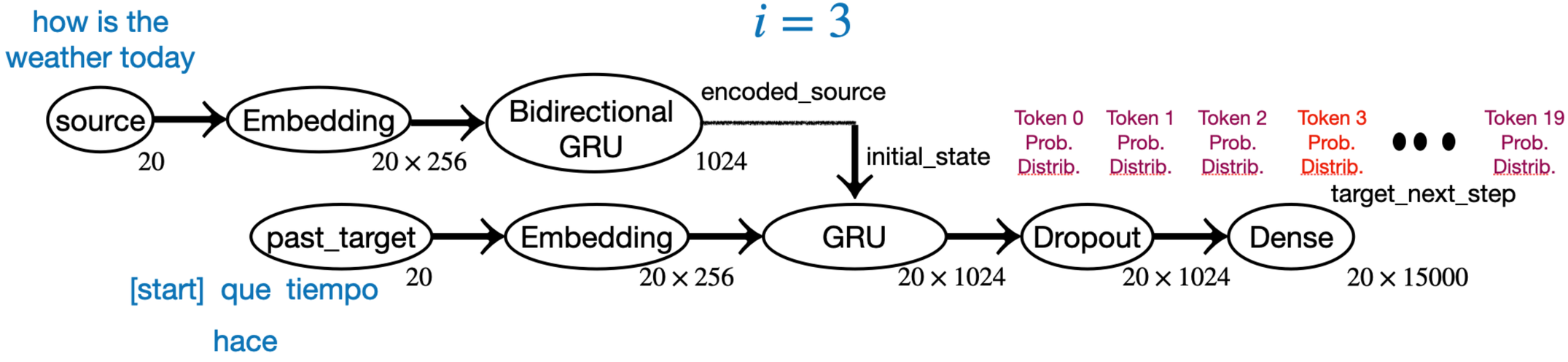
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 3$



Try 1 for Machine Translation: RNN

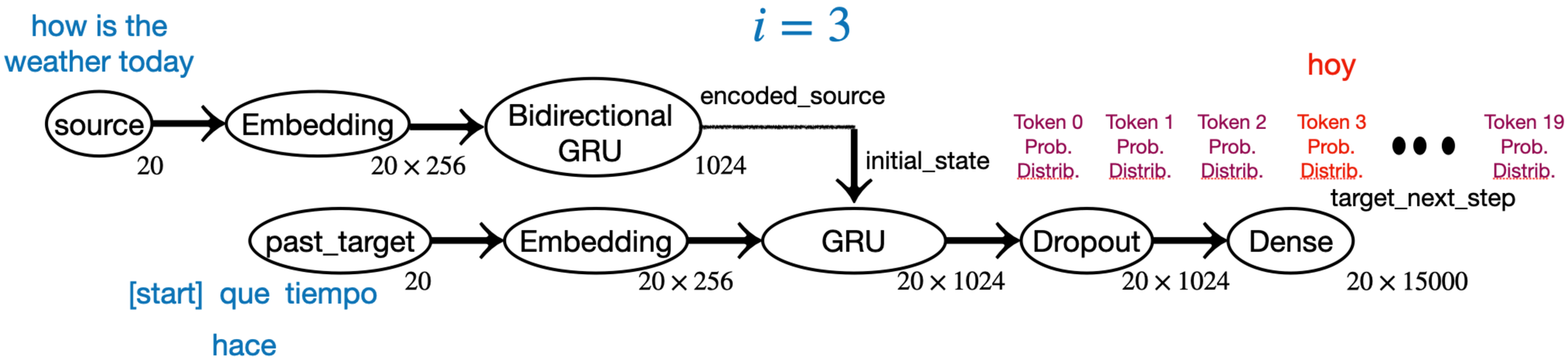
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

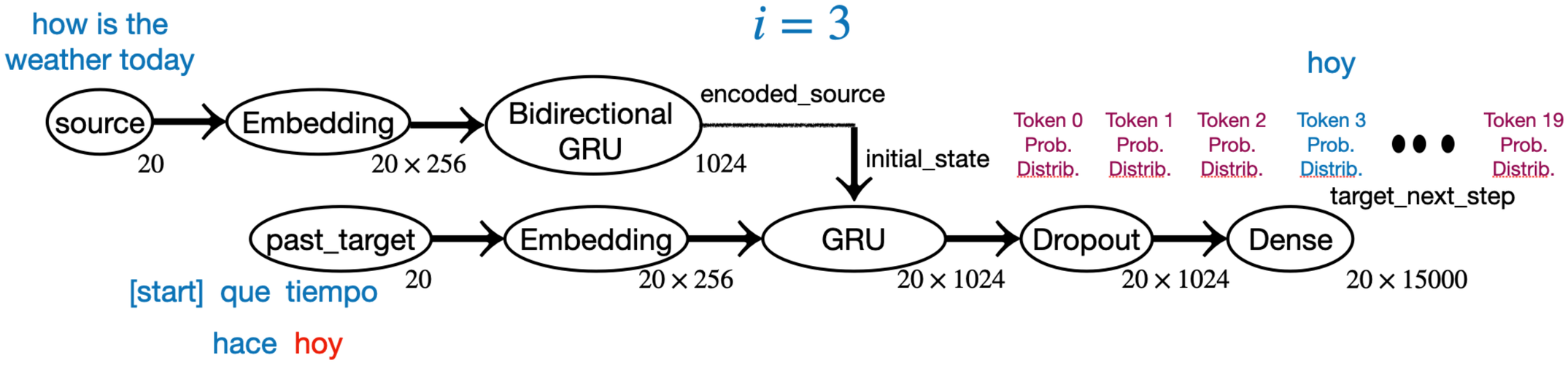
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

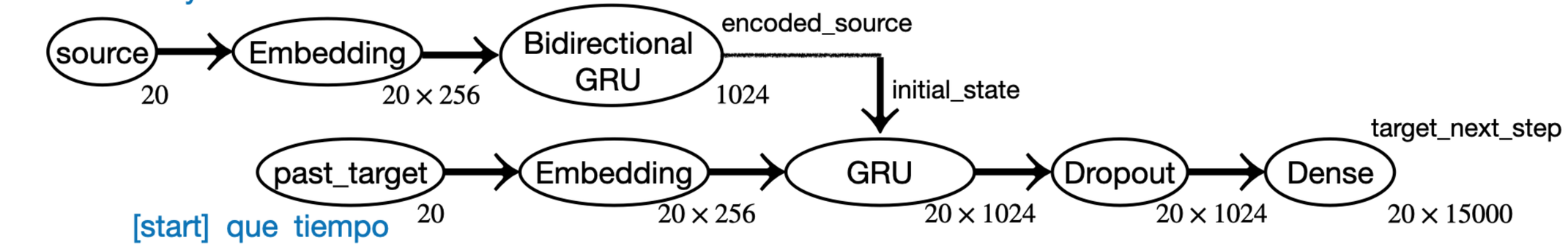
Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 4$

how is the weather today



[start] que tiempo hace hoy

Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

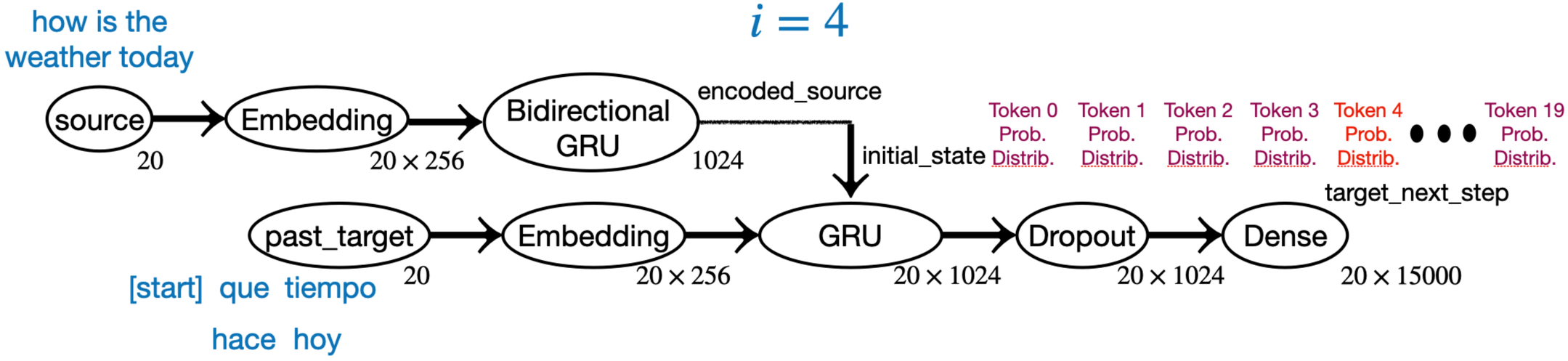
Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 4$



Try 1 for Machine Translation: RNN

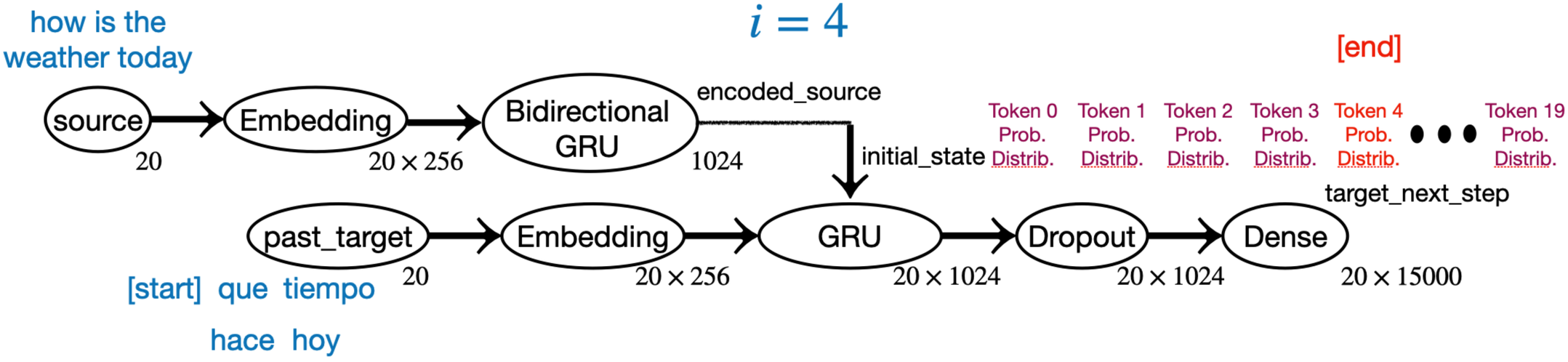
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

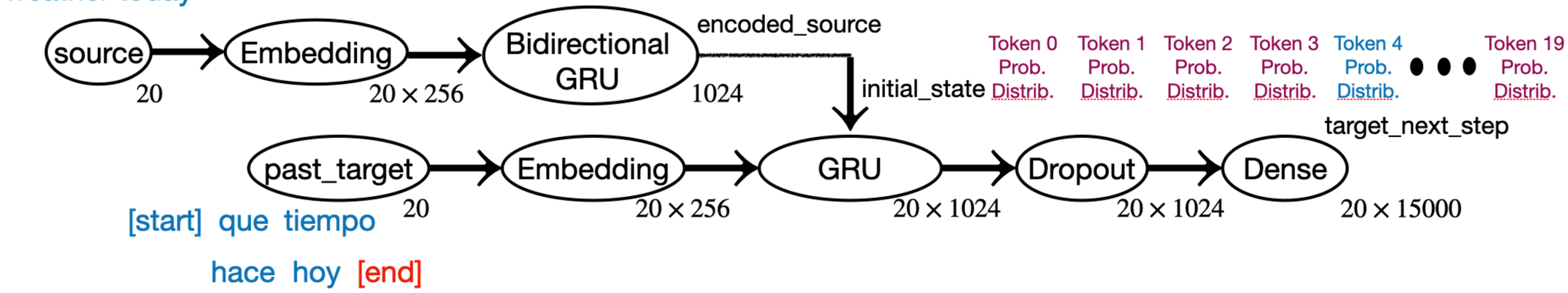
Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 4$

how is the weather today

[start] que tiempo hace hoy [end]



Try 1 for Machine Translation: RNN

```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

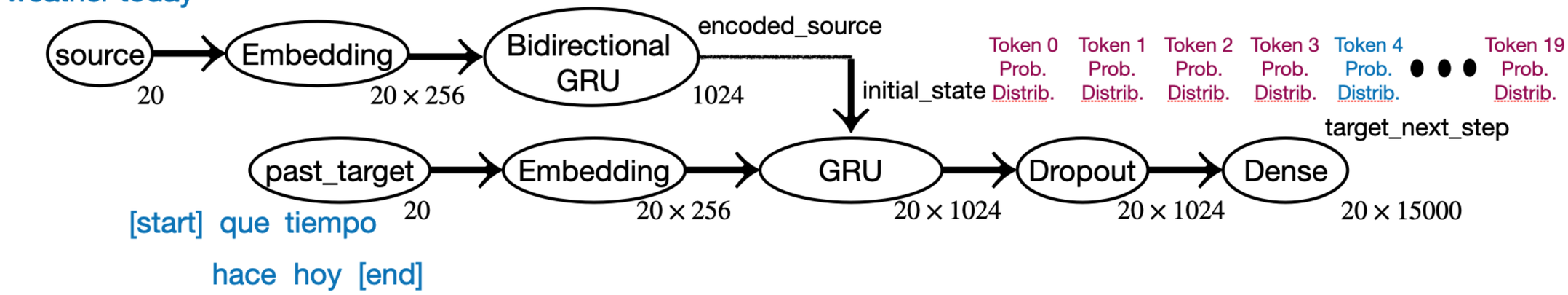
Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

$i = 4$

how is the weather today

[start] que tiempo hace hoy [end]



Try 1 for Machine Translation: RNN

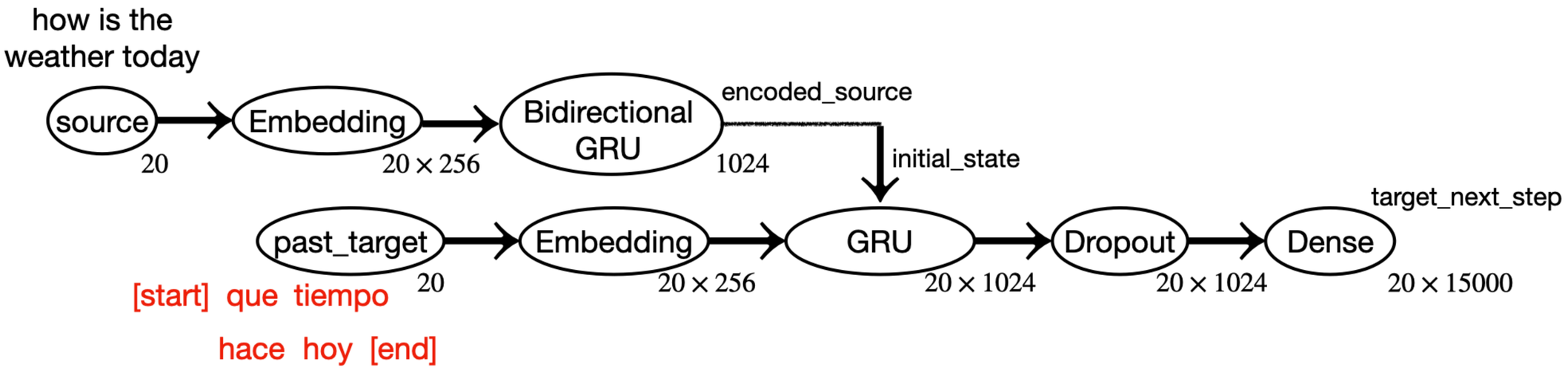
```
def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Seed token

Sample the next token.

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character



Try 1 for Machine Translation: RNN

Here are our translation results. Our model works decently well for a toy model, though it still makes many basic mistakes.

Listing 11.32 Some sample results from the recurrent translation model

```
Who is in this room?
[start] quién está en esta habitación [end]
-
That doesn't sound too dangerous.
[start] eso no es muy difícil [end]
-
No one will stop me.
[start] nadie me va a hacer [end]
-
Tom is friendly.
[start] tom es un buen [UNK] [end]
```

Try 1 for Machine Translation: RNN

There are many ways this toy model could be improved: We could use a deep stack of recurrent layers for both the encoder and the decoder (note that for the decoder, this makes state management a bit more involved). We could use an LSTM instead of a GRU. And so on. Beyond such tweaks, however, the RNN approach to sequence-to-sequence learning has a few fundamental limitations:

- The source sequence representation has to be held entirely in the encoder state vector(s), which puts significant limitations on the size and complexity of the sentences you can translate. It's a bit as if a human were translating a sentence entirely from memory, without looking twice at the source sentence while producing the translation.
- RNNs have trouble dealing with very long sequences, since they tend to progressively forget about the past—by the time you've reached the 100th token in either sequence, little information remains about the start of the sequence.

  That means RNN-based models can't hold onto long-term context, which can be essential for translating long documents.

These limitations are what has led the machine learning community to embrace the Transformer architecture for sequence-to-sequence problems. Let's take a look.

Quiz questions:

1. How does machine translation using RNN work?

2. What are possible ways to improve its performance?

Roadmap of this lecture (continued):

3. Sequence-to-Sequence Learning

4. Seq2Seq for Machine Translation

4.1 Try 1 for Machine Translation: RNN

4.2 Try 2 for Machine Translation: Transformer

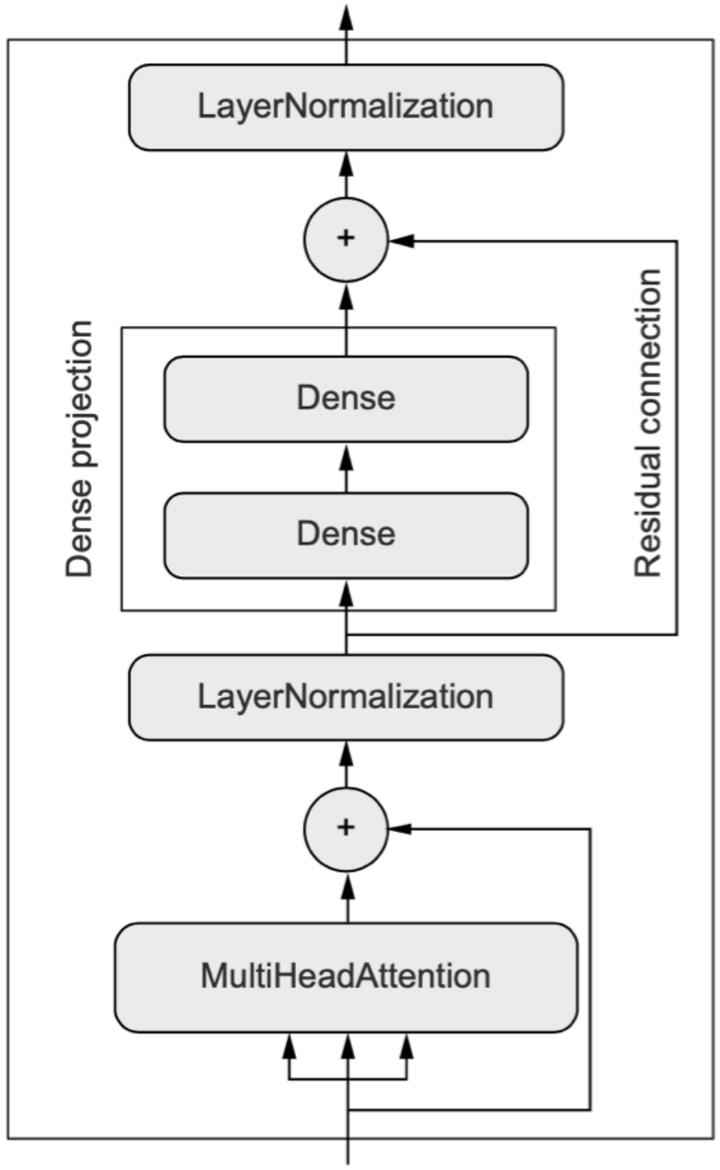
Try 2 for Machine Translation: Transformer

Unlike our previous RNN encoder, the Transformer encoder keeps the encoded representation in a sequence format: it's a sequence of context-aware embedding vectors.

The first half of the model is the Transformer encoder, which we have seen.

The second half of the model is the *Transformer decoder*. Just like the RNN decoder, it reads tokens 0…N in the target sequence and tries to predict token N+1. Crucially, while doing this, it uses neural attention to identify which tokens in the encoded source sentence are most closely related to the target token it's currently trying to predict—perhaps not unlike what a human translator would do. Recall the query-key-value model: in a Transformer decoder, the target sequence serves as an attention "query" that is used to to pay closer attention to different parts of the source sequence (the source sequence plays the roles of both keys and values).

The Transformer Encoder



Transformer Decoder

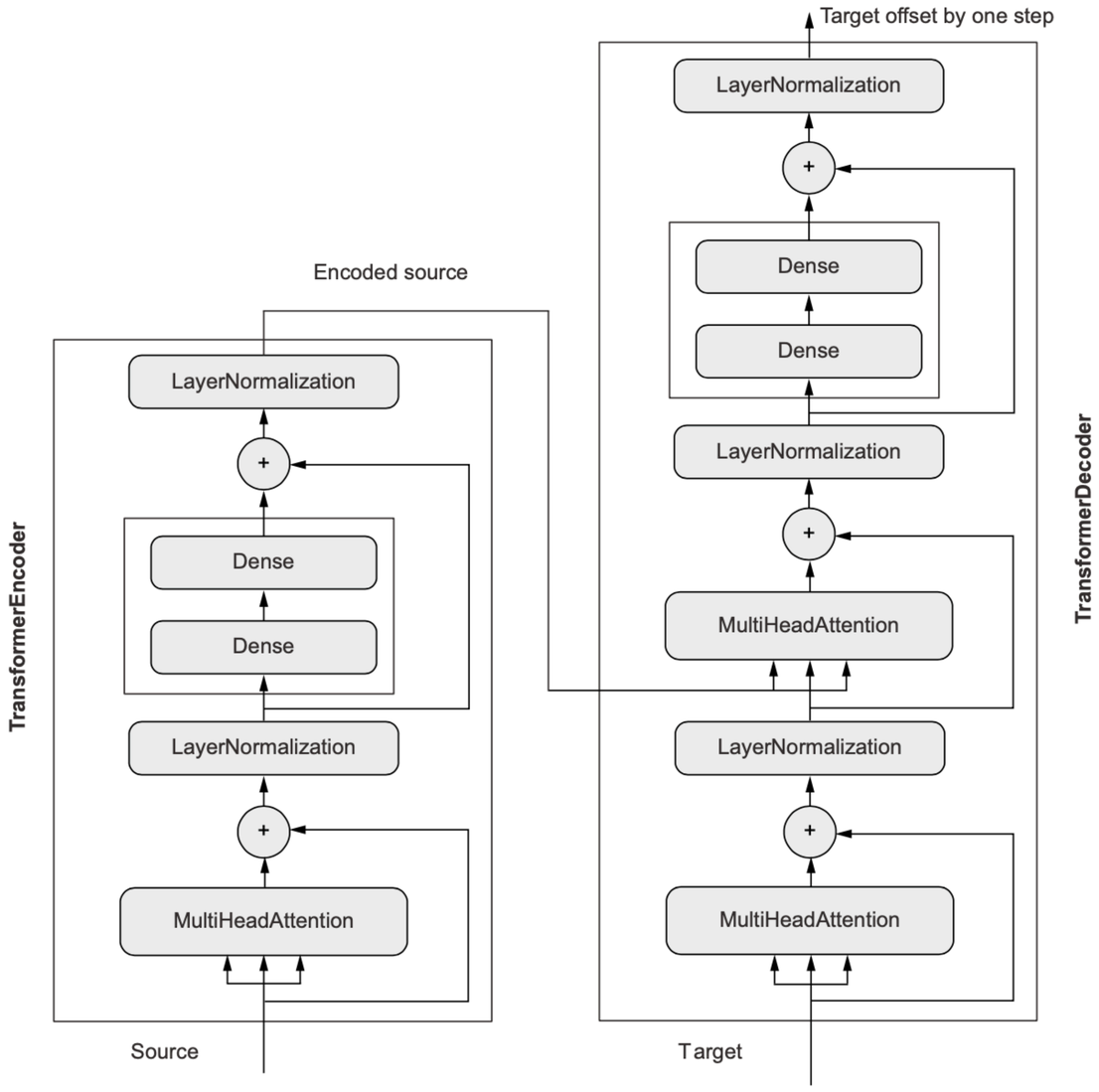


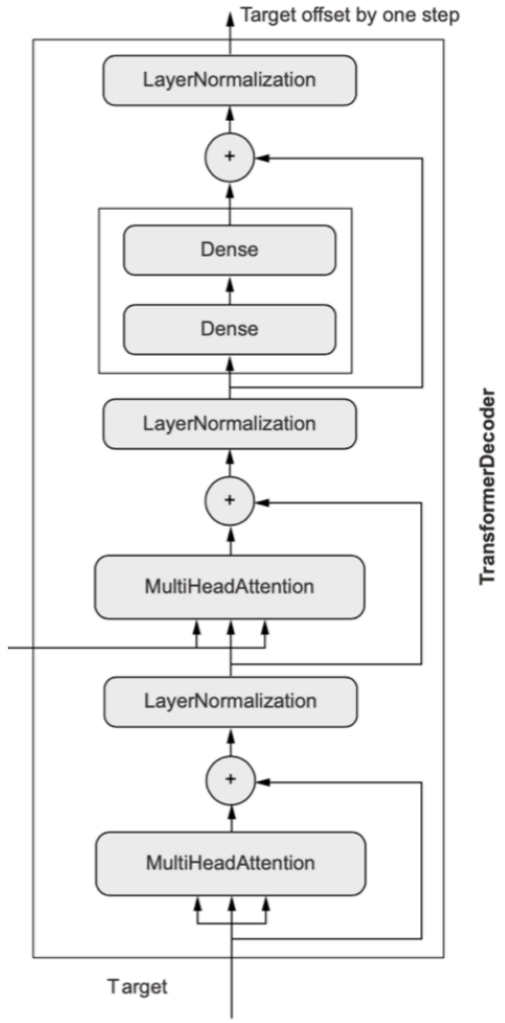
Figure 11.14 The `TransformerDecoder` is similar to the `TransformerEncoder`, except it features an additional attention block where the keys and values are the source sequence encoded by the `TransformerEncoder`. Together, the encoder and the decoder form an end-to-end Transformer.

Transformer Decoder

Listing 11.33 The `TransformerDecoder`

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

This attribute ensures that the layer will propagate its input mask to its outputs; masking in Keras is explicitly opt-in. If you pass a mask to a layer that doesn't implement compute_mask() and that doesn't expose this supports_masking attribute, that's an error.



Transformer Decoder

The `call()` method is almost a straightforward rendering of the connectivity diagram from figure 11.14. But there's an additional detail we need to take into account: *causal padding*. Causal padding is absolutely critical to successfully training a sequence-to-sequence Transformer. Unlike an RNN, which looks at its input one step at a time, and thus will only have access to steps 0...*N* to generate output step *N* (which is token *N*+1 in the target sequence), the `TransformerDecoder` is order-agnostic: it looks at the entire target sequence at once. If it were allowed to use its entire input, it would simply learn to copy input step *N*+1 to location *N* in the output. The model would thus achieve perfect training accuracy, but of course, when running inference, it would be completely useless, since input steps beyond *N* aren't available.

The fix is simple: we'll mask the upper half of the pairwise attention matrix to prevent the model from paying any attention to information from the future—only information from tokens 0...*N* in the target sequence should be used when generating target token *N*+1. To do this, we'll add a `get_causal_attention_mask(self, inputs)` method to our `TransformerDecoder` to retrieve an attention mask that we can pass to our `MultiHeadAttention` layers.

Transformer Decoder

Listing 11.34 `TransformerDecoder` method that generates a causal mask

Generate matrix of shape (sequence_length, sequence_length) with 1s in one half and 0s in the other.

```
def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
mult = tf.concat(
    [tf.expand_dims(batch_size, -1),
     tf.constant([1, 1], dtype=tf.int32)], axis=0)
return tf.tile(mask, mult)
```

Replicate it along the batch axis to get a matrix of shape (batch_size, sequence_length, sequence_length).

Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
    padding_mask = tf.minimum(padding_mask, causal_mask)
attention_output_1 = self.attention_1(
    query=inputs,
    value=inputs,
    key=inputs,
    attention_mask=causal_mask)
attention_output_1 = self.layernorm_1(inputs + attention_output_1)
attention_output_2 = self.attention_2(
    query=attention_output_1,
    value=encoder_outputs,
    key=encoder_outputs,
    attention_mask=padding_mask,
)
attention_output_2 = self.layernorm_2(
    attention_output_1 + attention_output_2)
proj_output = self.dense_proj(attention_output_2)
return self.layernorm_3(attention_output_2 + proj_output)
```

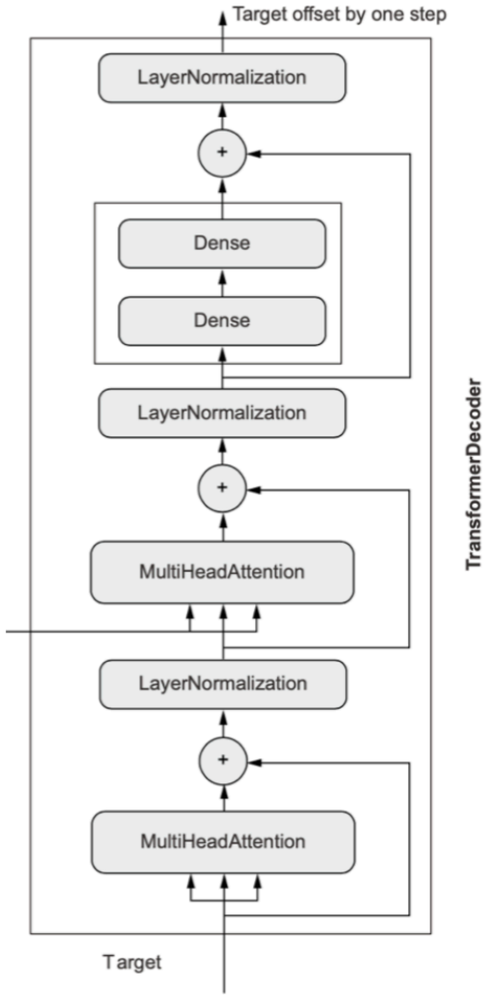
Retrieve the causal mask.

Merge the two masks together.

Prepare the input mask (that describes padding locations in the target sequence).

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

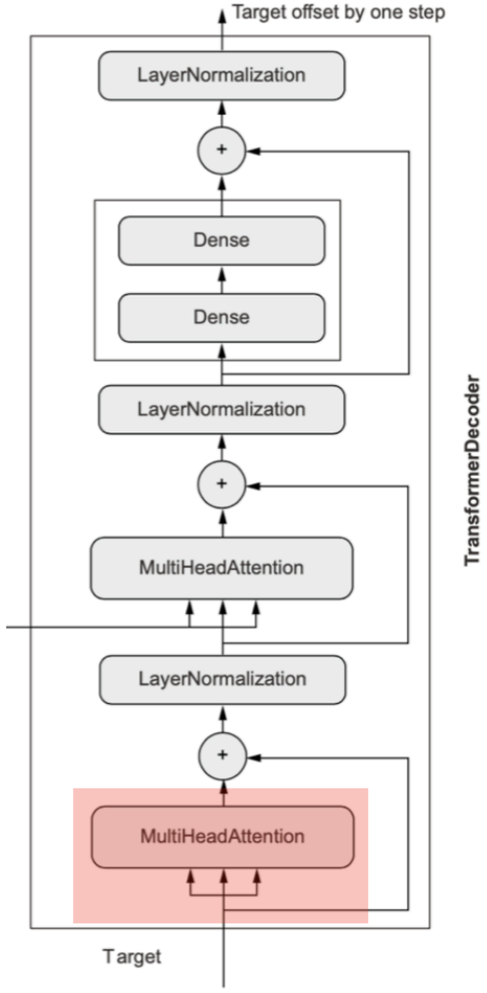
Retrieve the causal mask.

Prepare the input mask (that describes padding locations in the target sequence).

Merge the two masks together.

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

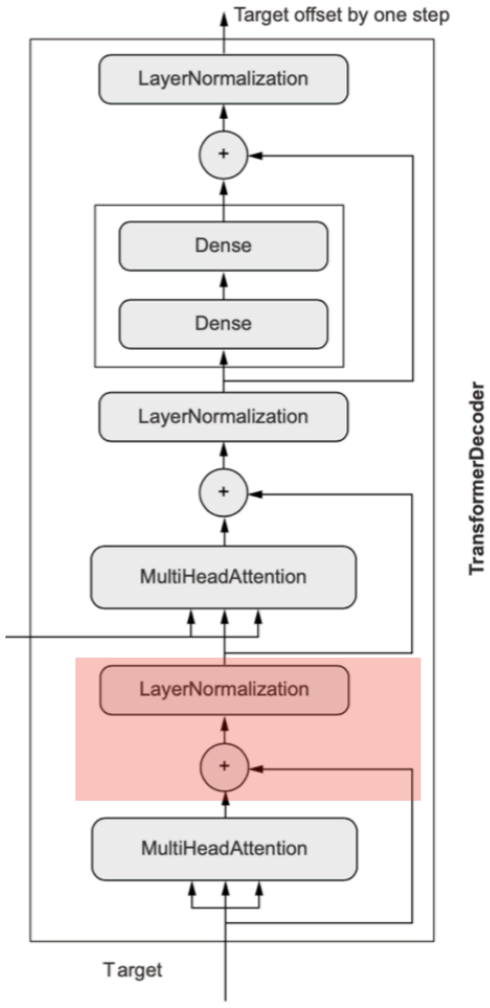
Retrieve the causal mask.

Merge the two masks together.

Prepare the input mask (that describes padding locations in the target sequence).

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

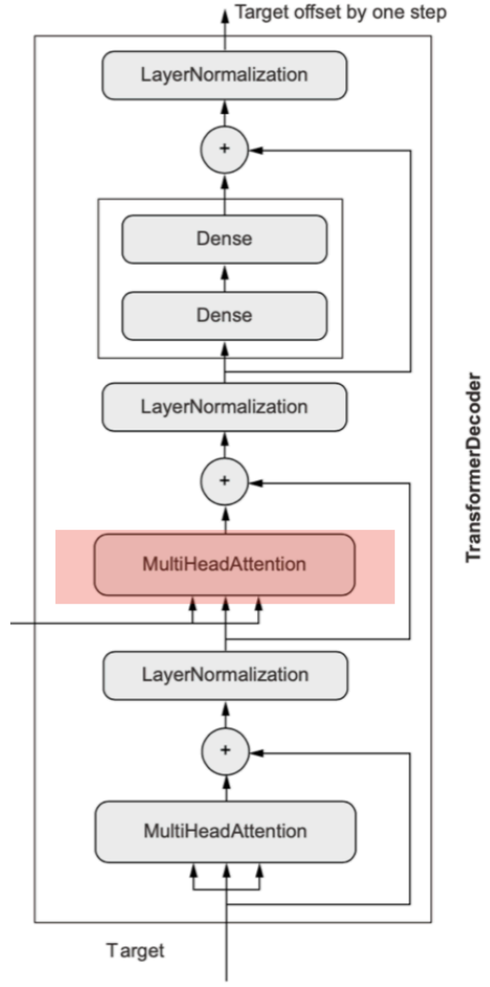
Retrieve the causal mask.

Prepare the input mask (that describes padding locations in the target sequence).

Merge the two masks together.

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

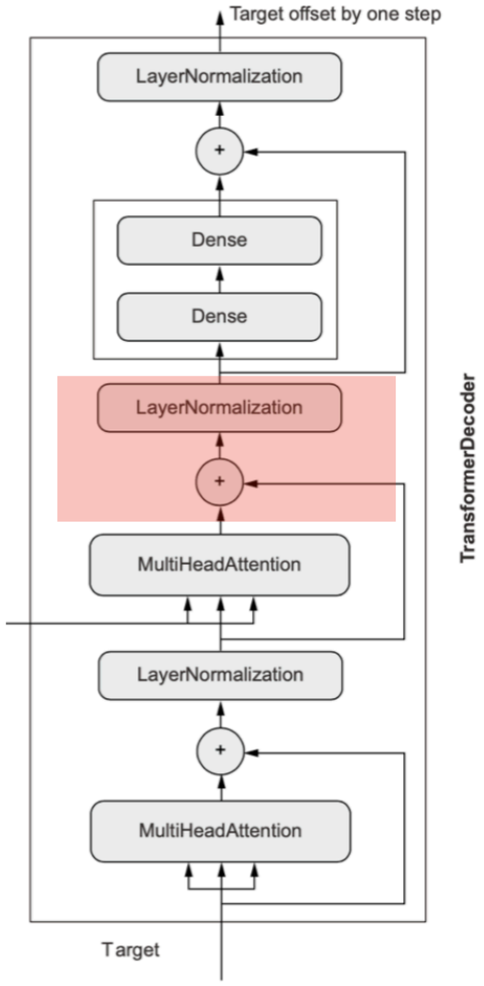
Retrieve the causal mask.

Prepare the input mask (that describes padding locations in the target sequence).

Merge the two masks together.

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

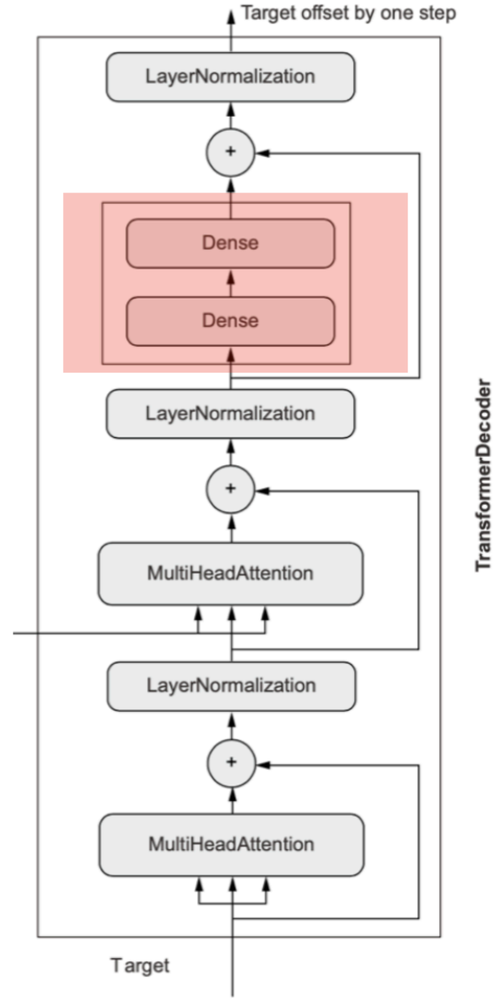
Retrieve the causal mask.

Merge the two masks together.

Prepare the input mask (that describes padding locations in the target sequence).

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Transformer Decoder

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

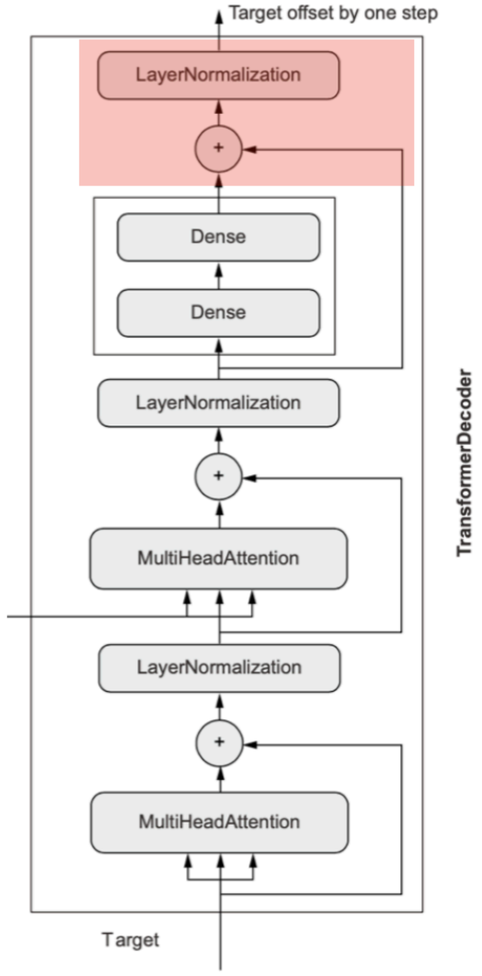
Retrieve the causal mask.

Merge the two masks together.

Prepare the input mask (that describes padding locations in the target sequence).

Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.

Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.



Try 2 for Machine Translation: Transformer

PUTTING IT ALL TOGETHER: A TRANSFORMER FOR MACHINE TRANSLATION

The end-to-end Transformer is the model we'll be training. It maps the source sequence and the target sequence to the target sequence one step in the future. It straightforwardly combines the pieces we've built so far: `PositionalEmbedding` layers, the `TransformerEncoder`, and the `TransformerDecoder`. Note that both the `TransformerEncoder` and the `TransformerDecoder` are shape-invariant, so you could be stacking many of them to create a more powerful encoder or decoder. In our example, we'll stick to a single instance of each.

Try 2 for Machine Translation: Transformer

Listing 11.36 End-to-end Transformer

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)

decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Encode the source sentence.

Encode the target sentence and combine it with the encoded source sentence.

Predict a word for each output position.

Try 2 for Machine Translation: Transformer

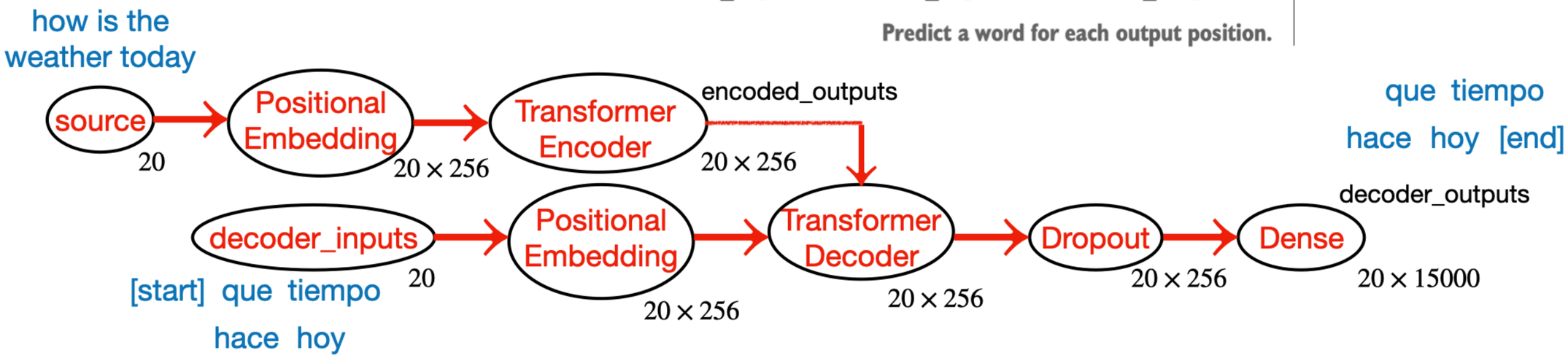
Listing 11.36 End-to-end Transformer

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)

decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Encode the source sentence.

Encode the target sentence and combine it with the encoded source sentence.

Predict a word for each output position.



Try 2 for Machine Translation: Transformer

We're now ready to train our model—we get to 67% accuracy, a good deal above the GRU-based model.

Listing 11.37 Training the sequence-to-sequence Transformer

```
transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

Try 2 for Machine Translation: Transformer

Finally, let's try using our model to translate never-seen-before English sentences from the test set. The setup is identical to what we used for the sequence-to-sequence RNN model.

Listing 11.38 Translating new sentences with our Transformer model

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization(
            [decoded_sentence])[:, :-1]
        predictions = transformer(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

Sample the next token.

Exit condition

Convert the next token prediction to a string, and append it to the generated sentence.

Try 2 for Machine Translation: Transformer

Listing 11.39 Some sample results from the Transformer translation model

```
This is a song I learned when I was a kid.
[start] esta es una canción que aprendí cuando era chico [end]
-
She can play the piano.
[start] ella puede tocar piano [end]
-
I'm not who you think I am.
[start] no soy la persona que tú creo que soy [end]
-
It may have rained a little last night.
[start] puede que llueve un poco el pasado [end]
```

While the source sentence wasn't gendered, this translation assumes a male speaker. Keep in mind that translation models will often make unwarranted assumptions about their input data, which leads to algorithmic bias. In the worst cases, a model might hallucinate memorized information that has nothing to do with the data it's currently processing.

Quiz questions:

1. What is the architecture of a Transformer decoder?

2. How does machine translation using Transformer work?