Deep Learning for Text

Dr. Md. Shahidur Rahman

Professor, DoCSE, SUST

This lecture covers

- Preprocessing text data for machine learning applications
- Bag-of-words approaches and sequence-modeling approaches for text processing
- The Transformer architecture
- Sequence-to-sequence learning

Natural Language Processing

Modern NLP is about: using machine learning and large datasets to give computers the ability not to understand language, 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.

The toolset of NLP

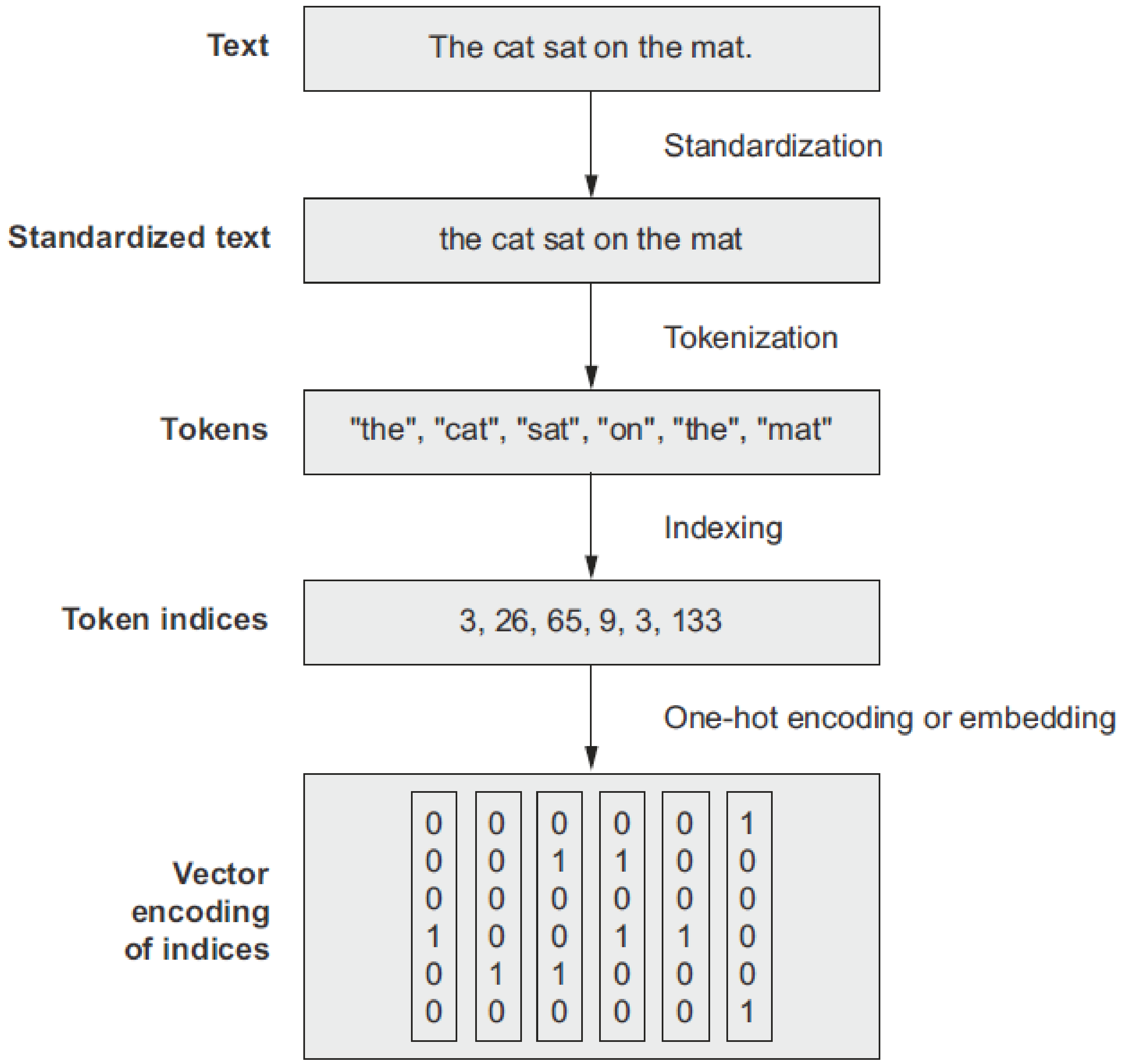
- Decision trees, logistic regression—only saw slow evolution from the 1990s to the early 2010s.
- Around 2014–2015, researchers began to investigate the language-understanding capabilities of recurrent neural networks, in particular LSTM—a sequence-processing algorithm
- In early 2015, Keras made available the first open source, easy-to-use implementation of LSTM
- From 2015 to 2017, recurrent neural networks dominated the booming NLP scene. Bidirectional LSTM models set the state of the art on many important tasks, from summarization to question-answering to machine translation.
- Around 2017–2018, a new architecture rose to replace RNNs: the Transformer which unlocked considerable progress in a short period of time, and today most NLP systems are based on them.

Preparing text data

Deep learning models can only process numeric tensors:

- They can’t take raw text as input. Vectorizing text is the process of transforming text into numeric tensors.
- 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.

From raw text to vectors



Text standardization

- Consider these two sentences:
  - “sunset came. i was staring at the Mexico sky. Isnt nature splendid??”
  - “Sunset came; I stared at the México sky. Isn’t nature splendid?”
- They’re very similar—yet, if you were to convert them to byte strings, they would end up with very different representations.
- Text standardization is a basic form of feature engineering that aims to erase encoding differences
  - One of the most widespread standardization schemes is “convert to lowercase and remove punctuation characters”
  - Another common transformation is to convert special characters to a standard form, such as replacing “é” with “e,” “æ” with “ae,” and so on.
  - An advanced standardization pattern is stemming: converting variations of a term (such as different conjugated forms of a verb) into a single shared representation
  - **Our sentences finally end up with an identical encoding: “sunset came i [stare] at the mexico sky isnt nature splendid”**

Text standardization…

- With these standardization techniques, the model requires less training data and generalizes better—it won’t need abundant examples of both “Sunset” and “sunset” to learn that they mean the same thing.
- Of course, standardization may also erase some information, so the context is important:
  - For instance, if you’re writing a model that extracts questions from interview articles, it should definitely treat “?” as a separate token instead of dropping it.

Text splitting (tokenization)

Once your text is standardized, you need to break it up into units (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 like in text generation or speech recognition.

Text splitting models

There are two kinds of text-processing models:

- Models that care about word order, called sequence models
  - For building a sequence model, you'll use word-level tokenization
- Models that treat input words as a set, discarding their original order, called bag-of-words models.
  - For building a bag-of-words model, you'll use N-gram tokenization.

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.
- 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:

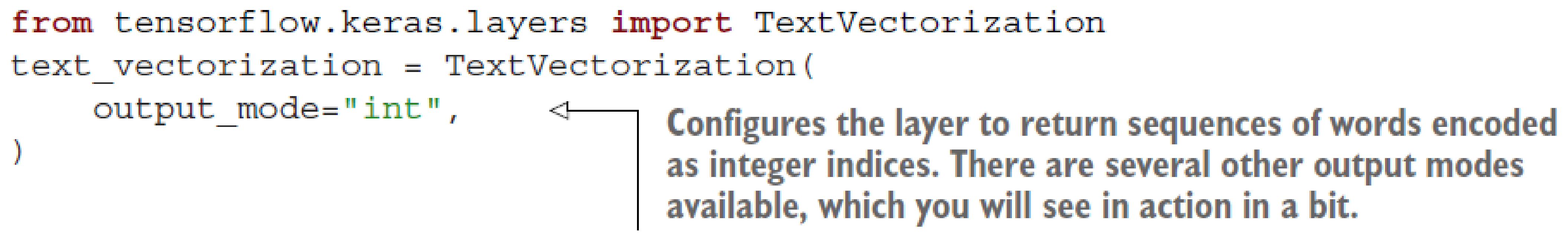
  {"the", "the cat", "cat", "cat sat", "the cat sat", "sat", "sat on", "on", "cat sat on", …}
- This family of tokenization methods is called bag-of-words (or bag-of-N-grams).
- Bag-of-words isn’t an order-preserving tokenization method it tends to be used in shallow language-processing models.
- Deep learning models like 1DCNN, RNN, and Transformers are capable of learning representations for groups of words and characters by looking at continuous word or character sequences.

Vocabulary indexing

- Once your text is split into tokens, you need to encode each token into a numerical representation.
  - Build an index of all terms found in the training data (the “vocabulary”), and assign a unique integer to each entry in the vocabulary.
  - Convert that integer into a vector encoding that can be processed by a neural network, like a one-hot vector.
- It’s common to restrict the vocabulary to only the top 20,000 or 30,000 most common words found in the training data. Indexing rare terms would result in an excessively large feature space

Using the TextVectorization layer

- The Keras TextVectorization layer for indexing is fast and efficient
- By default, the TextVectorization layer will use the setting “convert to lowercase and remove punctuation” for **text standardization**, and “split on whitespace” **for tokenization**.

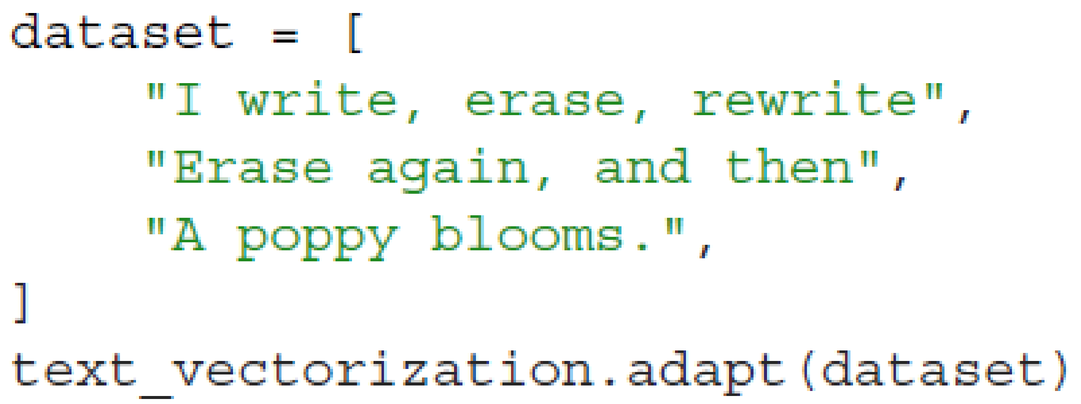


```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization(
    output_mode="int",
)
```

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.

Using 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)
```

- 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.

Two approaches for representing groups of words: Sets and sequences

- The simplest way you do is just discard order and treat text as an unordered set of words—this gives you **bag-of-words models**.
- If words should be processed strictly in the order in which they appear, one at a time, like steps in a timeseries—you could then leverage the **recurrent models.**
- A hybrid approach, the **Transformer architecture,** is technically order-agnostic, yet it injects word-position information into the representations it processes, which enables it to simultaneously look at different parts of a sentence (unlike RNNs) while still being order-aware.
- Because they take into account word order, both **RNNs and Transformers** are called **sequence models**.

Processing words as a set: The bag-of-words approach

- The simplest way to encode a piece of text for processing by a machine learning model is to discard order and treat it as a set (a "bag") of tokens.
- Individual words (unigrams), or groups of consecutive token (N-grams) can be processed
- Single words (**unigrams**) with binary encoding
  - If you use a bag of single words, the sentence "the cat sat on the mat" becomes {"cat", "mat", "on", "sat", "the"}
  - The main advantage of this encoding is that you can represent an entire text as a single vector, where each entry is **a presence indicator for a given word**.
- Limit the vocabulary to a **sensible MAX** of most frequent words. Otherwise, tens of thousands of terms that only occur once or twice and thus aren't informative will be indexed. In general, **20,000** is the right vocabulary size for **text classification**.

Bag-of-words: an Example

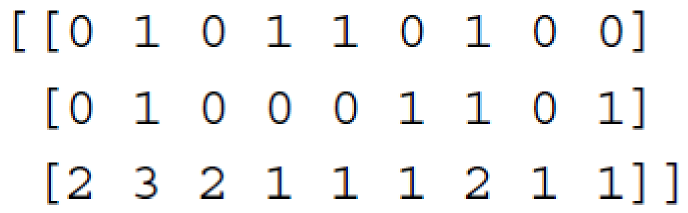
Text data (documents or sentences)

'The sun is shining'
'The weather is sweet'
'The sun is shining, the weather is sweet, and one and one is two’

- **Vocabulary Contents:**

  {'and': 0,
  'two': 7,
  'shining': 3,
  'one': 2,
  'sun': 4,
  'weather': 8,
  'the': 6,
  'sweet': 5,
  'is': 1}

Bag-of-words Vectors:



```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Bigrams with binary encoding

- You can re-inject local order information into your bag-of-words representation using N-grams (most commonly, bigrams).
- With **bigrams**, our sentence becomes {"the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the mat", "mat"}

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

- We now get 90.4% test accuracy, turns out local order is pretty important!

Bigrams with TF-IDF encoding

- 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, **terms that appear in almost every document** (like “the” or “a”) aren’t that informative, while terms **that appear only in a small subset** of all texts (like “Herzog”) are very distinctive.
- TF-IDF is a metric that 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.

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d)$$

$$idf(t,d) = \log \frac{\mathrm{n_d}}{1 + df(d,t)}$$

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

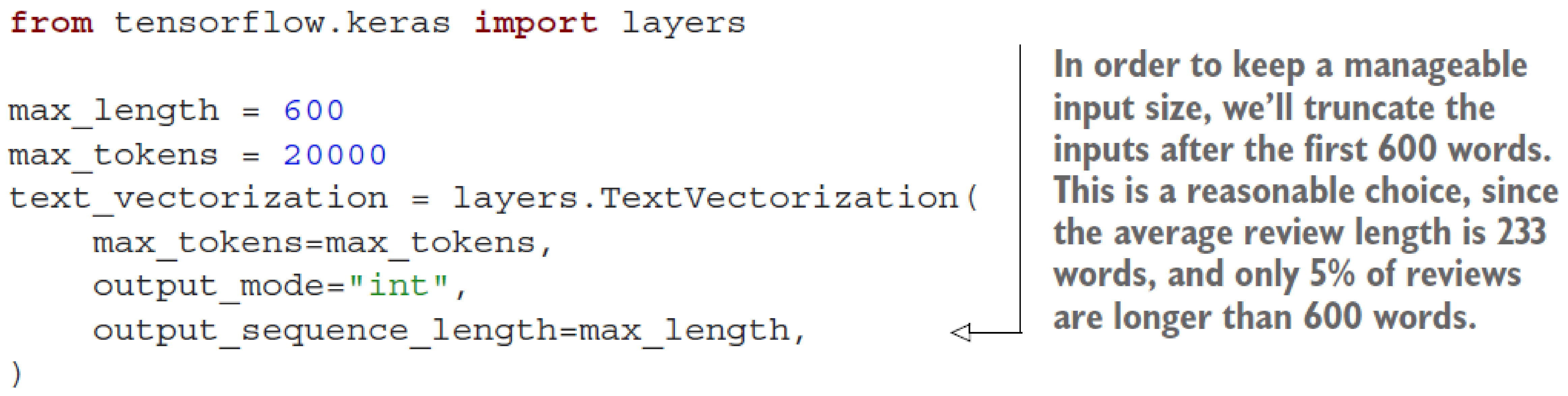
- This gets us an 89.8% test accuracy, it doesn’t seem to be particularly helpful in this case. However, for many text-classification datasets, it performs better.

Processing words as a sequence: The sequence model approach

To implement a sequence model:

- Start by representing the input samples as sequences of integer indices (one integer standing for one word).
- Map each integer to a vector to obtain vector sequences.
- Finally, feed these sequences of vectors into a stack of layers that could cross-correlate features from adjacent vectors, such as a 1D convnet, a RNN, or a Transformer.
- Nowadays sequence modeling is almost universally done with Transformers

A sequence model built on one-hot encoded vector sequences

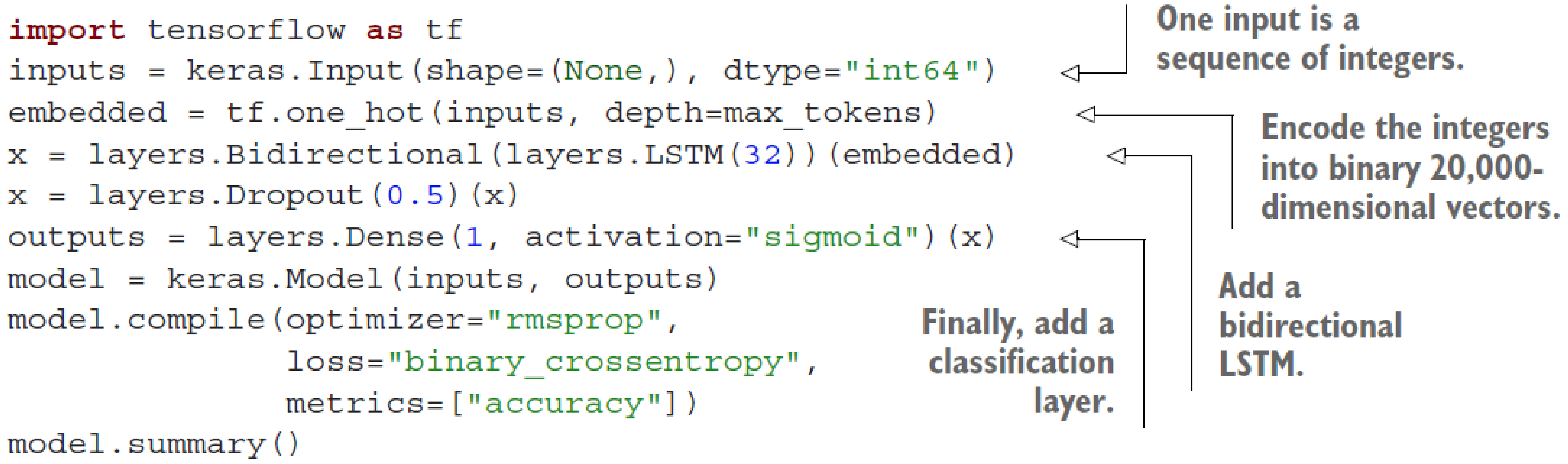


```
from tensorflow.keras import layers

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

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.

- The simplest way to convert our integer sequences to vector sequences is to one-hot encode the integers
- On top of these one-hot vectors, we'll add a simple bidirectional LSTM.



```
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.

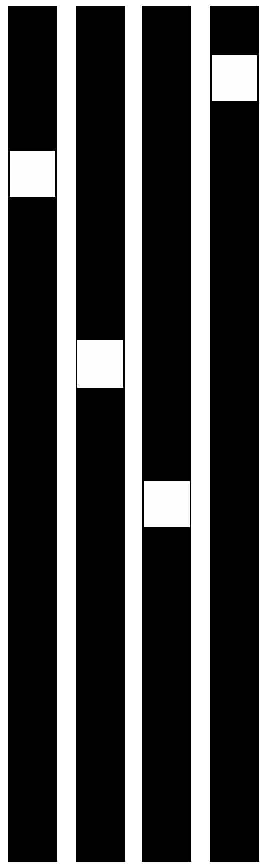
Add a bidirectional LSTM.

Finally, add a classification layer.

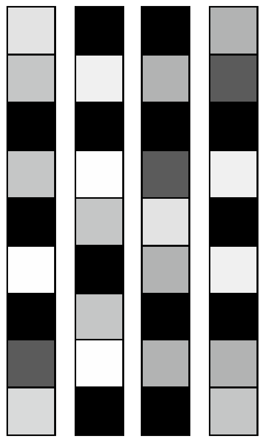
- The model only gets to **87% test accuracy**—it doesn't perform as well as our unigram model.
- Clearly, using **one-hot encoding** to turn words into vectors, which was the simplest thing we could do, **wasn't a great idea**.

Understanding word embeddings

- One-hot vectors are all orthogonal to one another (meaning tokens are independent from each other)
- This assumption is clearly wrong. The geometric relationship between two word vectors should reflect the semantic relationship between these words.
- Words that mean different things should lie far away from each other, whereas related words should be closer (e.g., synonyms should be embedded into similar word vectors)
- Word embeddings are vector representations of words that map human language into a structured geometric space.
- The vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional. Whereas, **word embeddings are low-dimensional floating-point vectors (dense vectors).**



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



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

Using pretrained word embeddings

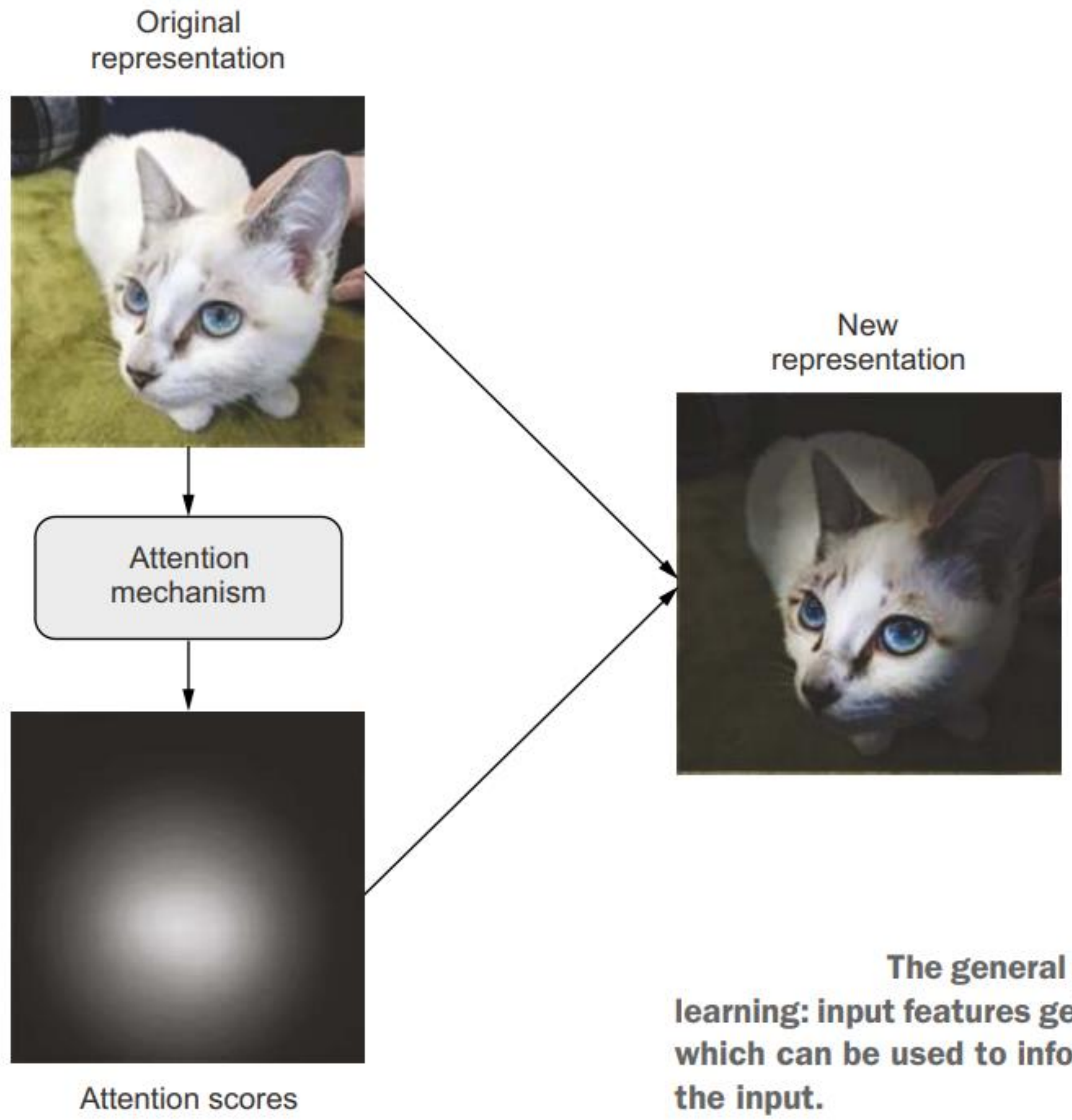
- Sometimes you have so little training data available that you can’t use your data alone. In such case, you can load embedding vectors from a precomputed embedding space.
- One of the most famous and successful word-embedding schemes: the Word2Vec algorithm (https://code.google.com/archive/p/word2vec), developed by Tomas Mikolov at Google in 2013.
- Another popular one is called Global Vectors for Word Representation (GloVe, https://nlp.stanford.edu/projects/glove), which was developed by Stanford researchers in 2014.

The Transformer architecture

- Transformers were introduced in the seminal paper “Attention is all you need” by Vaswani et al. in 2017.
- A simple mechanism called “neural attention” is used to build powerful sequence models that didn’t feature any recurrent layers or convolution layers.
- Neural attention has fast become one of the most influential ideas in deep learning.

Understanding self-attention

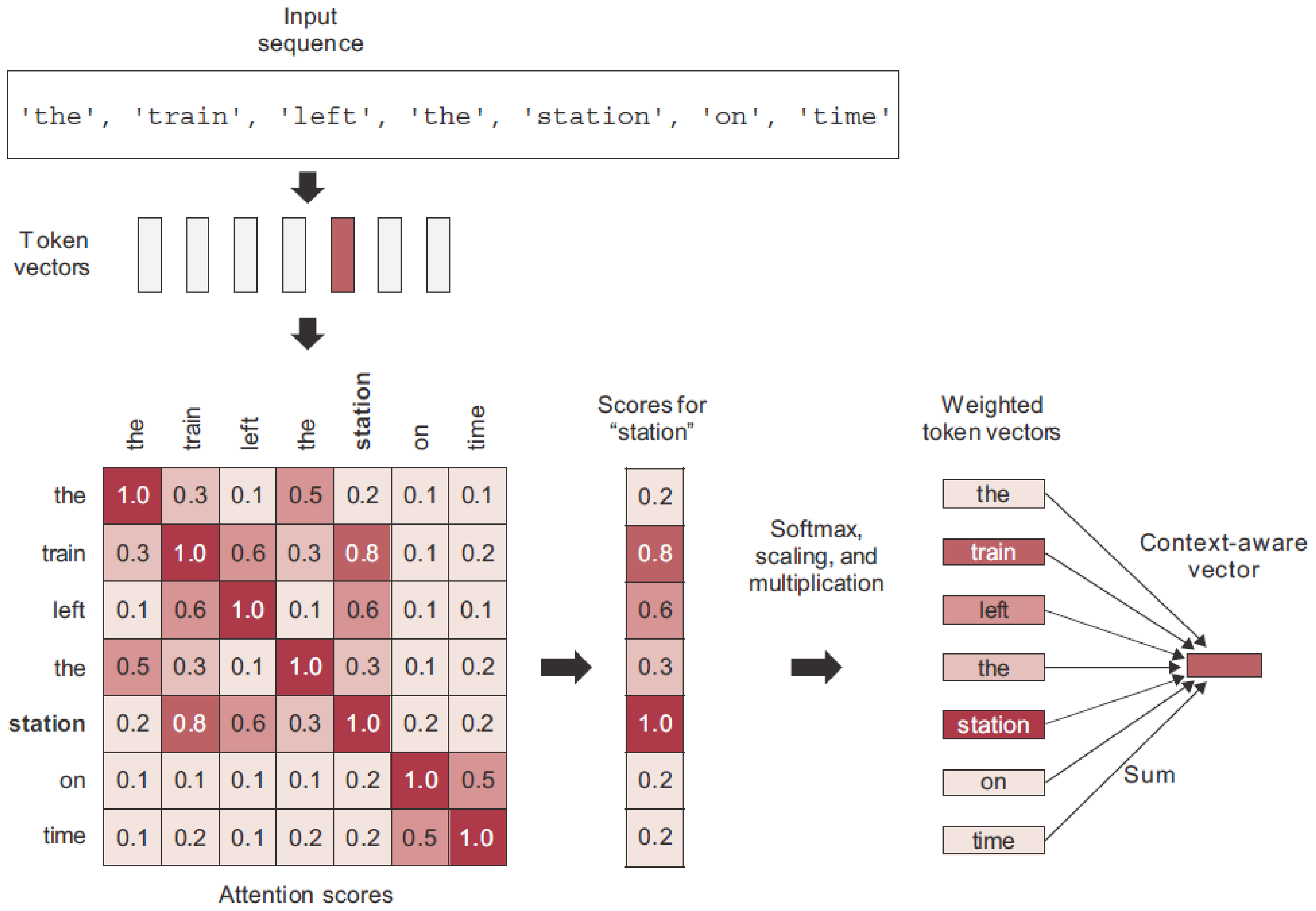
- Idea: not all input information seen by a model is equally important to the task at hand, so models should “pay more attention” to some features and “pay less attention” to other features.
- There are many different forms of attention that vary from approach to approach (e.g., Maxpooling, TF-IDF)



The general concept of "attention" in deep learning: input features get assigned "attention scores," which can be used to inform the next representation of the input.

Understanding self-attention…

- 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?



	the	train	left	the	station	on	time
the	1.0	0.3	0.1	0.5	0.2	0.1	0.1
train	0.3	1.0	0.6	0.3	0.8	0.1	0.2
left	0.1	0.6	1.0	0.1	0.6	0.1	0.1
the	0.5	0.3	0.1	1.0	0.3	0.1	0.2
station	0.2	0.8	0.6	0.3	1.0	0.2	0.2
on	0.1	0.1	0.1	0.1	0.2	1.0	0.5
time	0.1	0.2	0.1	0.2	0.2	0.5	1.0

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.

Self-Attention Encoding

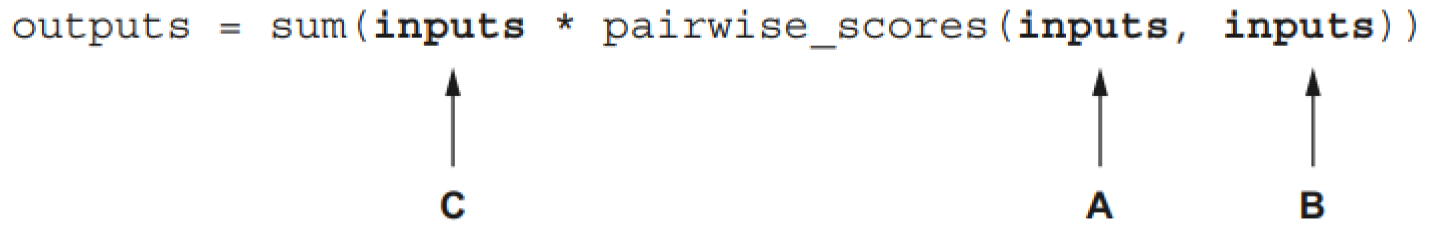
- Step 1 is to compute relevancy scores between the vector for "station" and every other word in the sentence. These are our "attention scores."
  - We're simply going to use the dot product between two word vectors as a measure of the strength of their relationship.
- Step 2 is to compute the sum of all word vectors in the sentence, weighted by our relevancy scores.
  - Words closely related to "station" will contribute more to the sum (including the word "station" itself), while irrelevant words will contribute almost nothing.
  - The resulting vector is our new representation for "station": a representation that incorporates the surrounding context.
  - In particular, it includes part of the "train" vector, clarifying that it is, in fact, a "train station."
- You'd repeat this process for every word in the sentence, producing a new sequence of vectors encoding the sentence.

Generalized self-attention: the query-key-value model

- The Transformer architecture was originally developed for machine translation, where you'll 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.

The query-key-value model…

- The self-attention mechanism 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)."

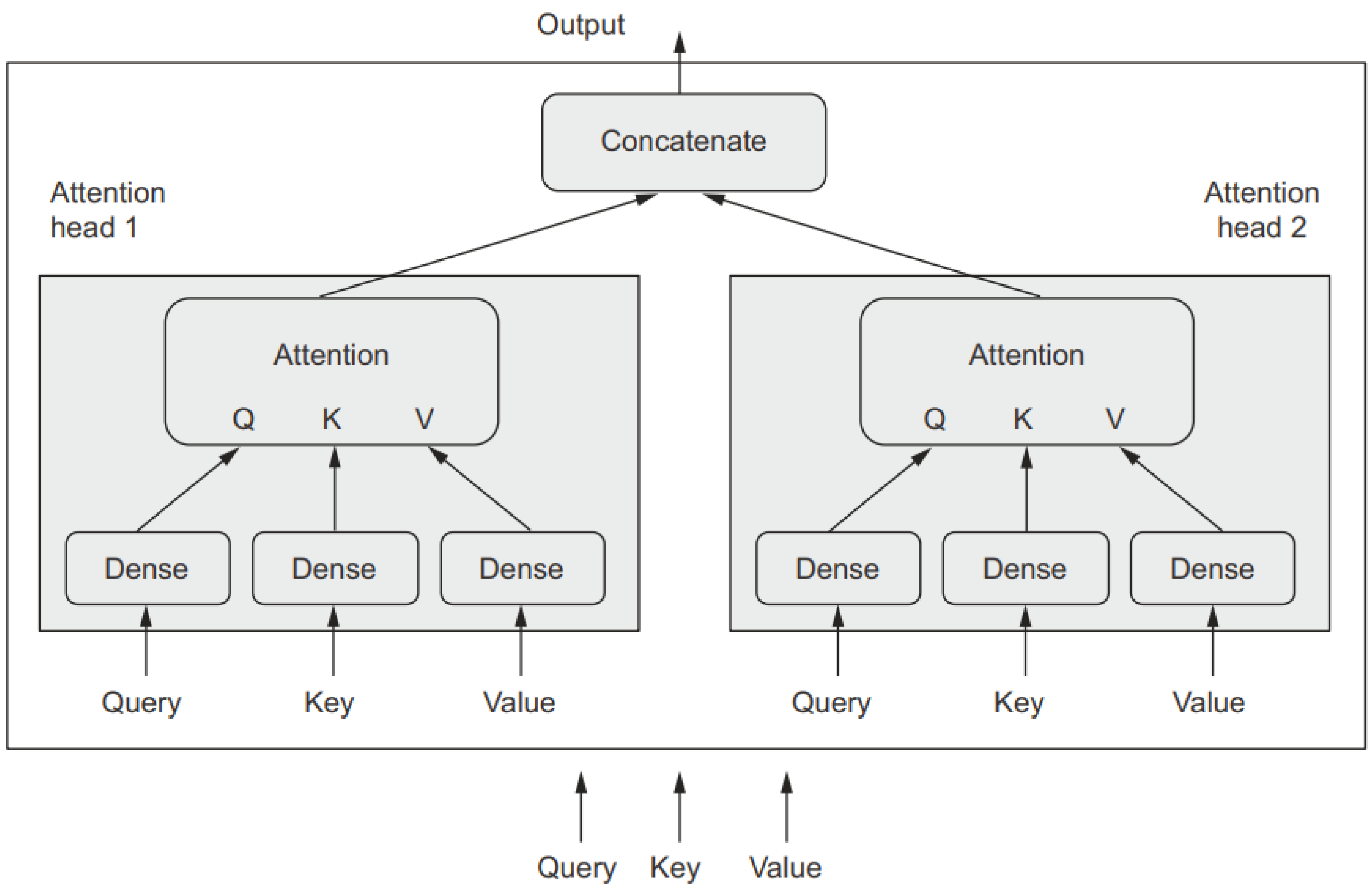
```
outputs = sum(values * pairwise_scores(query, keys))
```

- In general case, you could be doing this with three different sequences: "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":

Multi-head attention

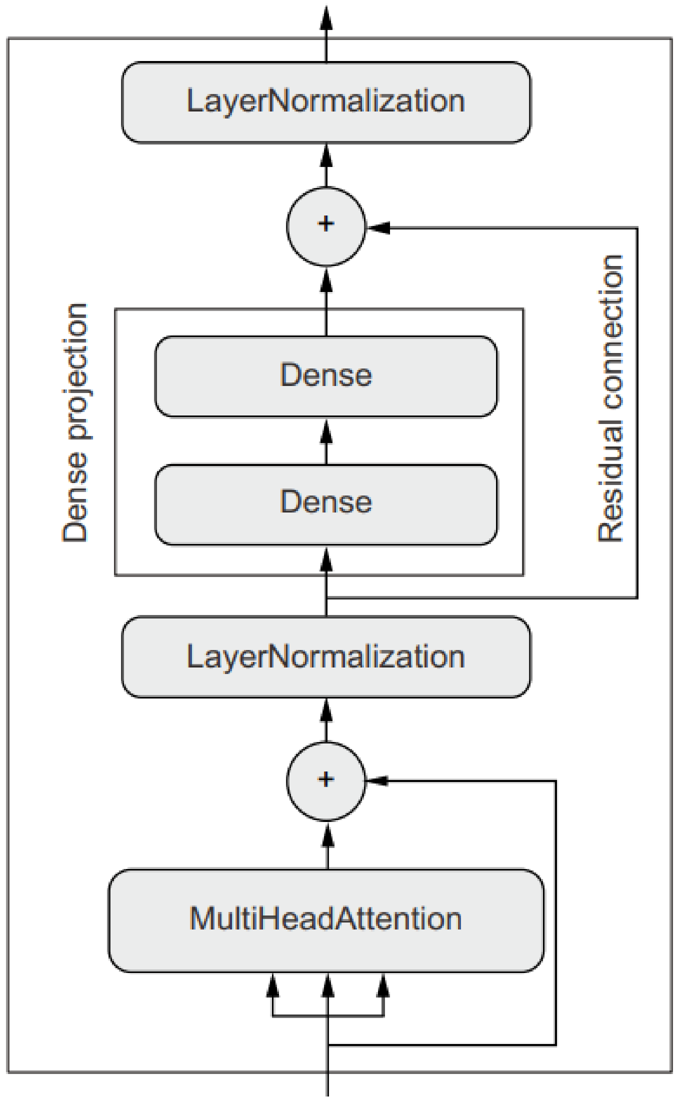
- “Multi-head attention” is an extra tweak to the self-attention mechanism
- The “multi-head” name refers to the fact that the output space of the self-attention layer gets factored into a set of independent subspaces, learned separately: the initial query, key, and value are sent through three independent sets of dense projections, resulting in three separate vectors.
- Each vector is processed via neural attention, and the three outputs are concatenated back together into a single output sequence. Each such subspace is called a “head.”

The MultiHeadAttention layer



The Transformer encoder

- The TransformerEncoder chains a MultiHeadAttention layer with a dense projection and adds normalization as well as residual connections.
- Adding extra dense projections is useful, so we also apply one or two to the output of the attention mechanism.
- Residual connections is to make sure we don't destroy any valuable information along the way. They're a must for any sufficiently deep architecture.
- Normalization layers are supposed to help gradients flow better during backpropagation.



The original Transformer architecture consists of two parts:

- **a Transformer encoder** that processes the source sequence, and
- **a Transformer decoder** that uses the source sequence to generate a translated version
- 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.

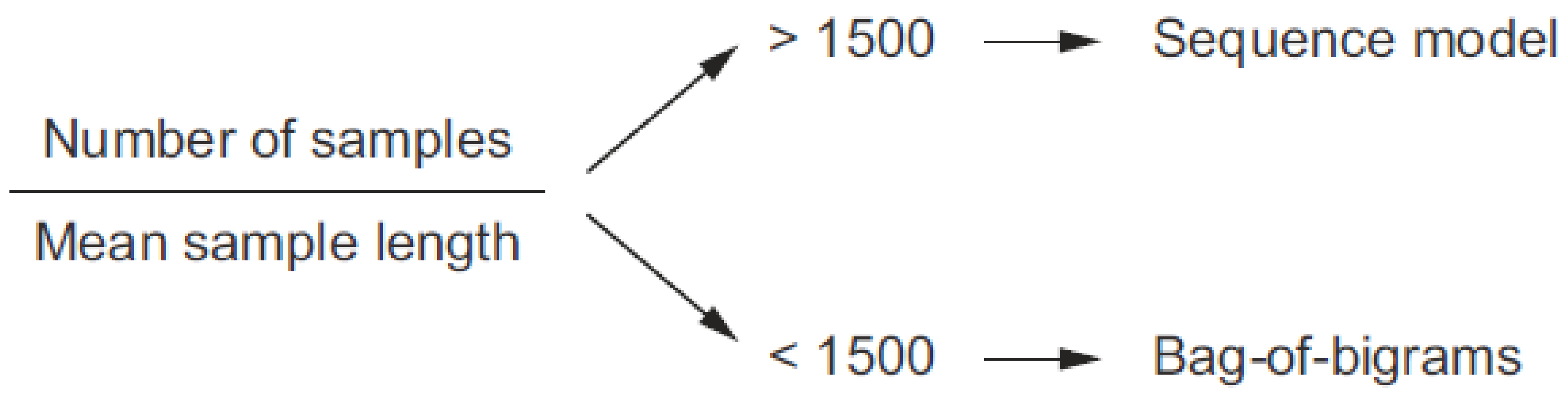
Using positional encoding to re-inject order information

- The idea behind positional encoding is very simple: to give the model access to word order information
- 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.
  - 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.
  - Because the positions can potentially be very large integers, we'll use position embedding vectors.

Classification using the Transformer encoder

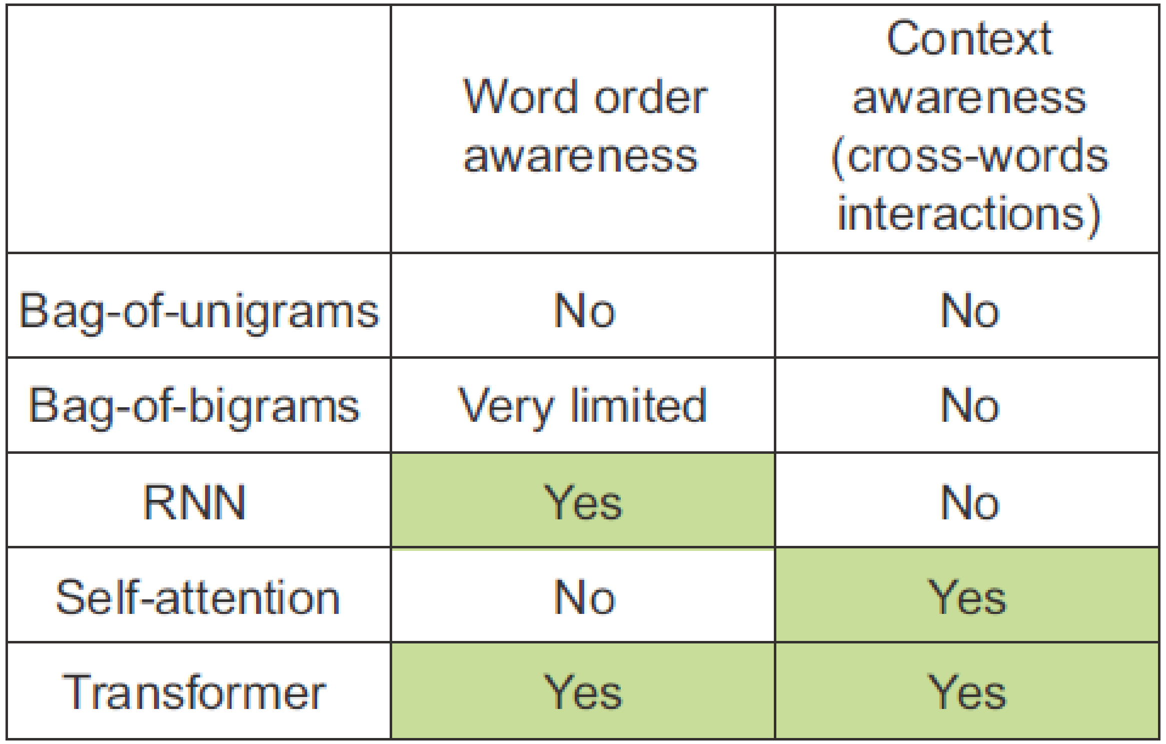
- We get to 88.3% test accuracy, a solid improvement that clearly demonstrates the value of word order information for text classification.
- This is our best sequence model so far! However, it’s still one notch below the bag-of-words approach.

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



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

Features of different types of NLP models



	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

Beyond text classification: Sequence-to-sequence learning

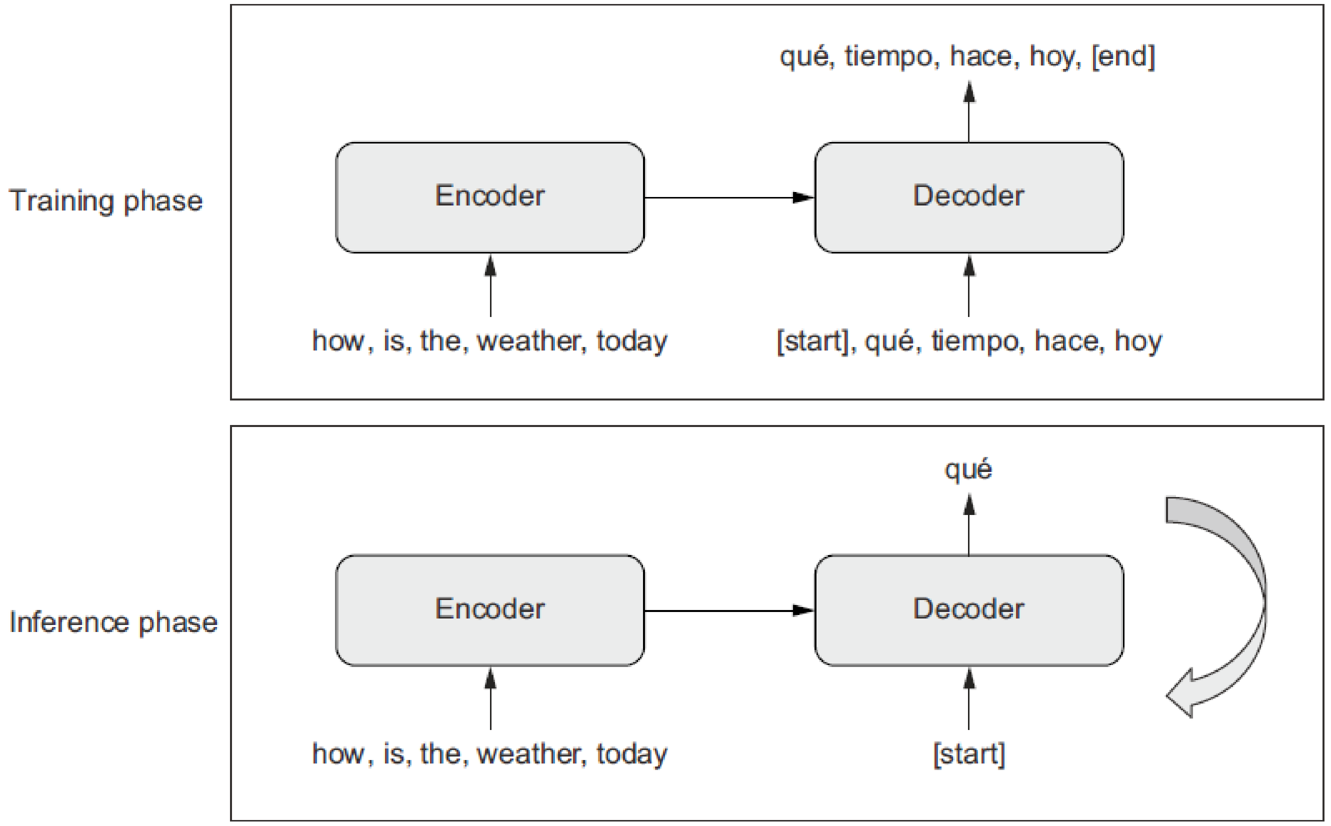
A sequence-to-sequence model takes a sequence as input (often a sentence or paragraph) and translates it into a different sequence. This is the task at the heart of many of the most successful applications of NLP:

- 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.

General template behind sequence-to-sequence models

- 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.

Sequence-to-sequence learning


qué, tiempo, hace, hoy, [end]
Training phase
Encoder
Decoder
how, is, the, weather, today
[start], qué, tiempo, hace, hoy
qué
Inference phase
Encoder
Decoder
how, is, the, weather, today
[start]

- 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.

Sequence-to-sequence learning…

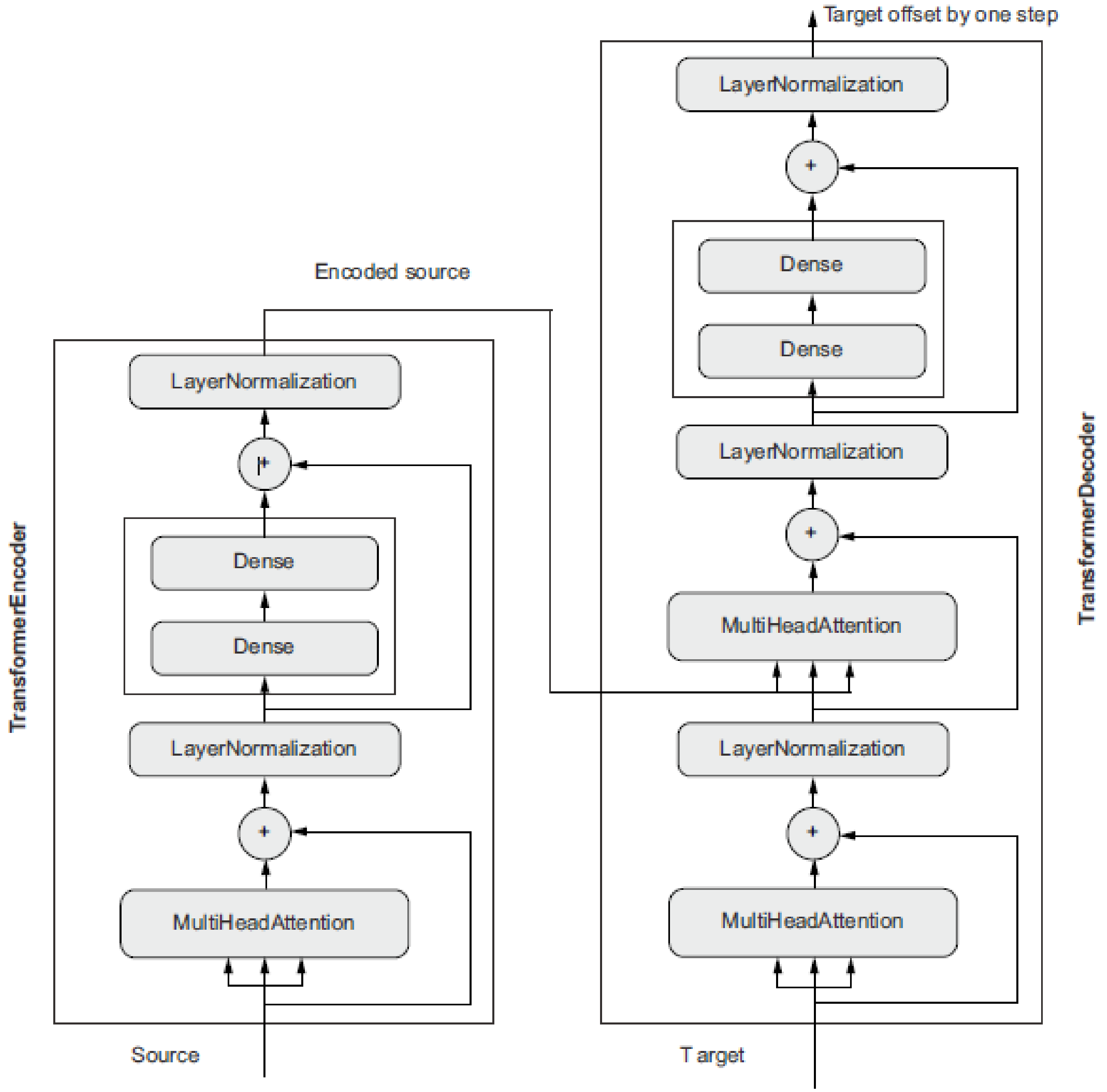
During inference, we don’t have access to the target sequence—we predict it from scratch. We generate it one token at a time:

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 with Transformer: A machine translation example

- RNN-based models can't hold onto long-term context, which can be essential for translating long documents.
- Sequence-to-sequence learning is the task where Transformer really shines.
- Neural attention enables Transformer models to successfully process sequences that are considerably longer and more complex than those RNNs can handle.
- Transformer encoder uses self-attention to produce context-aware representations of each token in an input sequence.
- In a sequence-to-sequence learning, the Transformer encoder would naturally play the role of the encoder, which reads the source sequence and produces an encoded representation of it.

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.



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.
- 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.

Summary

- There are two kinds of NLP models: bag-of-words models process sets of words or N-grams without taking into account their order, and sequence models process word order.
- A bag-of-words model is made of Dense layers, while a sequence model could be an RNN, a 1D convnet, or a Transformer.
- When it comes to text classification, the ratio between the number of samples in your training data and the mean number of words per sample can help you determine whether you should use a bag-of-words model or a sequence model.
- Word embeddings are vector spaces where semantic relationships between words are modeled as distance relationships between vectors that represent those words.
- Sequence-to-sequence learning is a generic, powerful learning framework that can be applied to solve many NLP problems, including machine translation.
- A sequence to-sequence model is made of an encoder, which processes a source sequence, and a decoder, which tries to predict future tokens in target sequence by looking at past tokens, with the help of the encoder-processed source sequence.

Summary…

- Neural attention is a way to create context-aware word representations. It’s the basis for the Transformer architecture.
- The Transformer architecture, which consists of a TransformerEncoder and a TransformerDecoder, yields excellent results on sequence-to-sequence tasks.
- The first half, the TransformerEncoder, can also be used for text classification or any sort of single-input NLP task.