Instituto Politécnico Nacional
"La Técnica al Servicio de la Patria"

DIPLOMADO EN INTELIGENCIA ARTIFICIAL

TRANSFORMERS

PARTE 2 - RNN, LSTM Y GRU

ALAN BADILLO SALAS

MÓDULO 11

CONTENIDO

CORPUS Y TOKENIZACIÓN

RNN, LSTM Y GRU

SELF-ATTENTION Y TRANSFORMERS

ARQUITECTURA TRANSFORMER

PREENTRENAMIENTO Y FINE-TUNING

LLMS - ENTRENAMIENTO, ÉTICA Y FUTURO

Problema

La inferencia sobre textos permite resolver tareas como la **clasificación de textos** para predecir categorías y mostrar publicidad, la **similitud entre textos** para predecir el plagio o la **respuesta de entradas de texto a salidas de texto** para chatbots.

Antes de los **transformers** se usaron las **redes neuronales recurrentes** para hacer predicciones sobre las secuencias y lograr infererir las próximas secuencias o clasificar los textos.

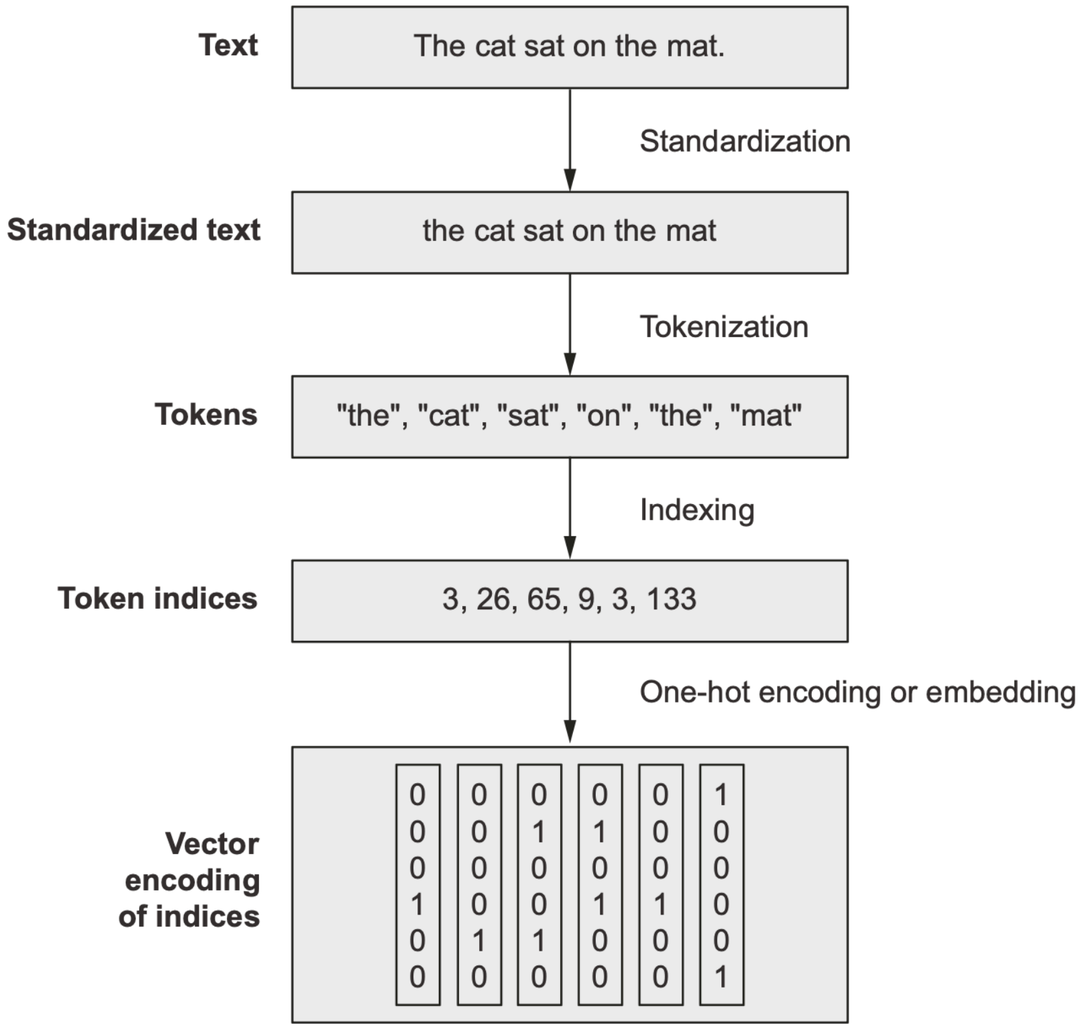


Figure 11.1 From raw text to vectors

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?"

One of the simplest and most widespread standardization schemes is "convert to lowercase and remove punctuation characters." Our two sentences would become

- "sunset came i was staring at the mexico sky isnt nature splendid"
- "sunset came i stared at the méxico sky isnt nature splendid"

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.

Vocabulary indexing

Once your text is split into tokens, you need to encode each token into a numerical representation. You could potentially do this in a stateless way, such as by hashing each token into a fixed binary vector, but in practice, the way you'd go about it is to 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)


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

¿Cuál es el problema de la codificación one-hot?

Using the TextVectorization layer

Every step I've introduced so far would be very easy to implement in pure Python. Maybe you could write something like this:

However, using something like this wouldn't be very performant. In practice, you'll work with the Keras `TextVectorization` layer, which is fast and efficient and can be dropped directly into a `tf.data` pipeline or a Keras model.

This is what the `TextVectorization` layer looks like:

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

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


>>> 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"
```

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 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.

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.

Two approaches for representing groups of words: Sets and sequences

How a machine learning model should represent *individual words* is a relatively uncontroversial question: they're categorical features (values from a predefined set), and we know how to handle those. They should be encoded as dimensions in a feature space, or as category vectors (word vectors in this case). A much more problematic question, however, is how to encode *the way words are woven into sentences*: word order.

The problem of order in natural language is an interesting one: unlike the steps of a timeseries, words in a sentence don't have a natural, canonical order. Different languages order similar words in very different ways. For instance, the sentence structure of English is quite different from that of Japanese. Even within a given language, you can typically say the same thing in different ways by reshuffling the words a bit. Even further, if you fully randomize the words in a short sentence, you can still largely figure out what it was saying—though in many cases significant ambiguity seems to arise. Order is clearly important, but its relationship to meaning isn't straightforward.

How to represent word order is the pivotal question from which different kinds of NLP architectures spring. The simplest thing you could do is just discard order and treat text as an unordered set of words—this gives you *bag-of-words models*. You could also decide that 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 from the last chapter. Finally, a hybrid approach is also possible: 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*.

Historically, most early applications of machine learning to NLP just involved bag-of-words models. Interest in sequence models only started rising in 2015, with the rebirth of recurrent neural networks. Today, both approaches remain relevant. Let's see how they work, and when to leverage which.

We'll demonstrate each approach on a well-known text classification benchmark: the IMDB movie review sentiment-classification dataset. In chapters 4 and 5, you worked with a prevectorized version of the IMDB dataset; now, let's process the raw IMDB text data, just like you would do when approaching a new text-classification problem in the real world.

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.

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.

Two approaches for representing groups of words: Sets and sequences

🔷 **En RNNs**

- Las **RNN, LSTM, GRU** procesan **secuencialmente** los tokens: primero el 1°, luego el 2°, etc.
- Cada estado oculto depende del **orden anterior**, por lo que **NO son order-agnostic**.
- Ejemplo:
  - Frase: *"El perro muerde al gato"*
  - Si cambias el orden a *"El gato muerde al perro"*, la RNN produce representaciones distintas.

👉 Las RNN son **order-sensitive**.

🔷 **En Transformers**

- El mecanismo de **self-attention** en su forma básica **no conoce el orden de los tokens**.
- Los tokens se procesan en paralelo y el modelo solo aprende relaciones de atención **sin noción de posición**.
- Eso los hace **order-agnostic por diseño**.
- Para que un transformer distinga entre *"el perro muerde al gato"* y *"el gato muerde al perro"*, se introducen:
  - **Positional encodings** (en Vaswani et al., 2017)
  - O embeddings posicionales aprendibles (BERT, RoBERTa, etc.)

👉 Un transformer **sin positional encoding** sería completamente order-agnostic (como si tratara el texto como un "bag of words").

Preparing the IMDB movie reviews data

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/
......pos/
......neg/
...test/
......pos/
......neg/
```

For instance, the train/pos/ directory contains a set of 12,500 text files, each of which contains the text body of a positive-sentiment movie review to be used as training data. The negative-sentiment reviews live in the "neg" directories. In total, there are 25,000 text files for training and another 25,000 for testing.

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
```

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.

```
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.

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

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.

```
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.

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

```
>>> 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">
```

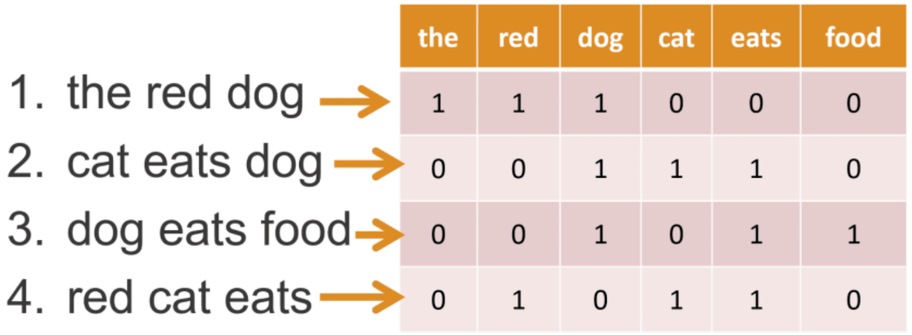
Processing words as a set: The bag-of-words approach

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. You could either look at individual words (unigrams), or try to recover some local order information by looking at groups of consecutive token (N-grams).

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"}
```



Preprocessing our datasets with a `TextVectorization` layer

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.

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

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

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

```
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)
```

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

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.

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
```

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.

This gets us to a test accuracy of 89.2%: not bad! Note that in this case, since the dataset is a balanced two-class classification dataset (there are as many positive samples as negative samples), the "naive baseline" we could reach without training an actual model would only be 50%. Meanwhile, the best score that can be achieved on this dataset without leveraging external data is around 95% test accuracy.

Bigrams with binary encoding

Of course, discarding word order is very reductive, because even atomic concepts can be expressed via multiple words: the term “United States” conveys a concept that is quite distinct from the meaning of the words “states” and “united” taken separately. For this reason, you will usually end up re-injecting local order information into your bag-of-words representation by looking at N-grams rather than single words (most commonly, bigrams).

With bigrams, our sentence becomes

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

Configuring the `TextVectorization` layer to return bigrams

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

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)
]

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}")
```

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")
```

Práctica 7 - 1gram vs 2gram

Intituto Politécnico Nacional

Centro de Investigación en Computación

Departamento de Diplomados y Extensión Profesional

Diplomado en Inteligencia Artificial

- Módulo 11 - Parte II

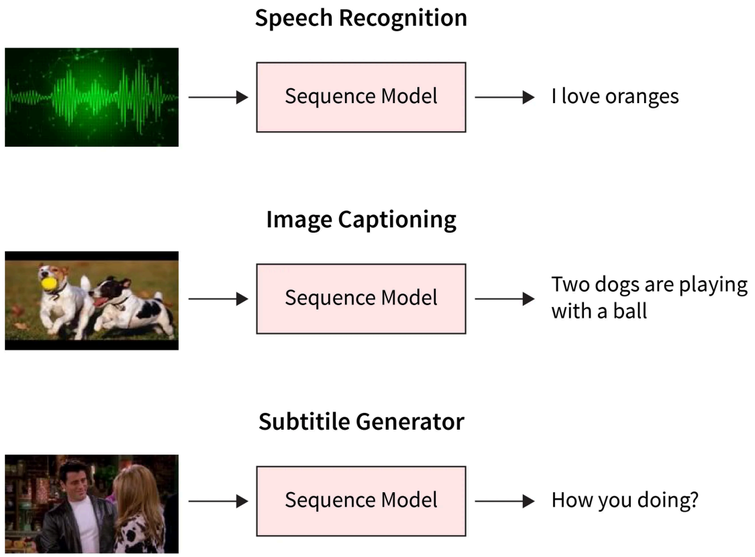
Profesor: Alan Badillo Salas (badillosalas@outlook.com)

Processing words as a sequence: The sequence model approach

These past few examples clearly show that word order matters: manual engineering of order-based features, such as bigrams, yields a nice accuracy boost. Now remember: the history of deep learning is that of a move away from manual feature engineering, toward letting models learn their own features from exposure to data alone. What if, instead of manually crafting order-based features, we exposed the model to raw word sequences and let it figure out such features on its own? This is what *sequence models* are about.

To implement a sequence model, you'd start by representing your input samples as sequences of integer indices (one integer standing for one word). Then, you'd map each integer to a vector to obtain vector sequences. Finally, you'd 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.

For some time around 2016–2017, bidirectional RNNs (in particular, bidirectional LSTMs) were considered to be the state of the art for sequence modeling. Since you're already familiar with this architecture, this is what we'll use in our first sequence model examples. However, nowadays sequence modeling is almost universally done with Transformers, which we will cover shortly. Oddly, one-dimensional convnets were never very popular in NLP, even though, in my own experience, a residual stack of depthwise-separable 1D convolutions can often achieve comparable performance to a bidirectional LSTM, at a greatly reduced computational cost.



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.

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}")
```

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.

Práctica 8 - Bidirectional LSTM

Instituto Politécnico Nacional

Centro de Investigación en Computación

Departamento de Diplomados y Extensión Profesional

Diplomado en Inteligencia Artificial

- Módulo 11 - Parte II

Profesor: Alan Badillo Salas (badillosalas@outlook.com)

Understanding word embeddings

Crucially, when you encode something via one-hot encoding, you're making a feature-engineering decision. You're injecting into your model a fundamental assumption about the structure of your feature space. That assumption is that *the different tokens you're encoding are all independent from each other*: indeed, one-hot vectors are all orthogonal to one another. And in the case of words, that assumption is clearly wrong. Words form a structured space: they share information with each other. The words "movie" and "film" are interchangeable in most sentences, so the vector that represents "movie" should not be orthogonal to the vector that represents "film"—they should be the same vector, or close enough.

To get a bit more abstract, the *geometric relationship* between two word vectors should reflect the *semantic relationship* between these words. For instance, in a reasonable word vector space, you would expect synonyms to be embedded into similar word vectors, and in general, you would expect the geometric distance (such as the cosine distance or L2 distance) between any two word vectors to relate to the "semantic distance" between the associated words. Words that mean different things should lie far away from each other, whereas related words should be closer.

Word embeddings are vector representations of words that achieve exactly this: they map human language into a structured geometric space.

Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (the same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors); see figure 11.2. It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

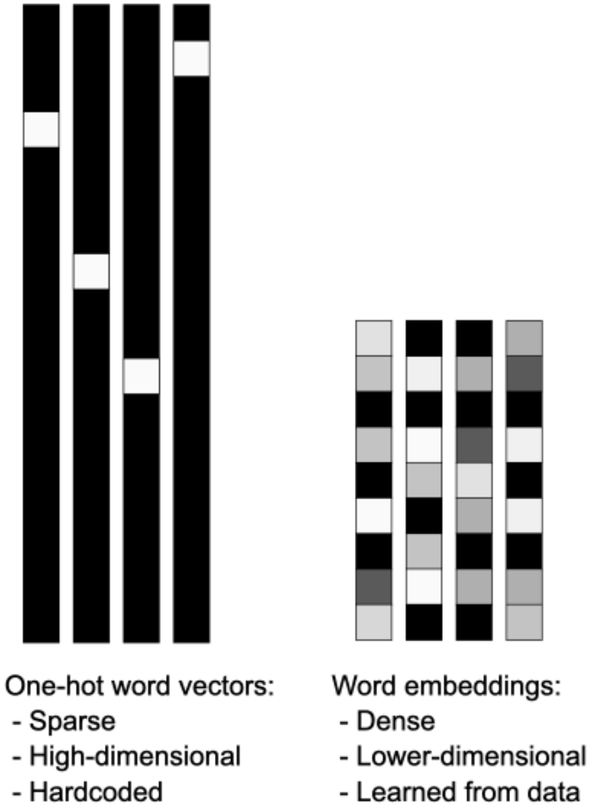


Figure 11.2 Word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded. Word embeddings are dense, relatively low-dimensional, and learned from data.

Besides being *dense* representations, word embeddings are also *structured* representations, and their structure is learned from data. Similar words get embedded in close locations, and further, specific *directions* in the embedding space are meaningful. To make this clearer, let's look at a concrete example.

In figure 11.3, four words are embedded on a 2D plane: *cat*, *dog*, *wolf*, and *tiger*. With the vector representations we chose here, some semantic relationships between these words can be encoded as geometric transformations. For instance, the same vector allows us to go from *cat* to *tiger* and from *dog* to *wolf*: this vector could be interpreted as the "from pet to wild animal" vector. Similarly, another vector lets us go from *dog* to *cat* and from *wolf* to *tiger*, which could be interpreted as a "from canine to feline" vector.

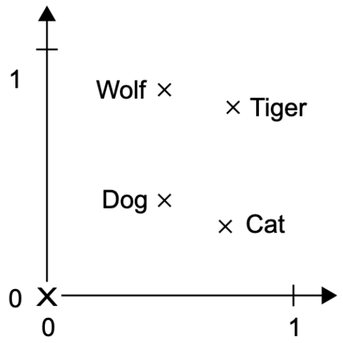


Figure 11.3 A toy example of a word-embedding space

In real-world word-embedding spaces, common examples of meaningful geometric transformations are "gender" vectors and "plural" vectors. For instance, by adding a "female" vector to the vector "king," we obtain the vector "queen." By adding a "plural" vector, we obtain "kings." Word-embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

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*.

Learning word embeddings with the Embedding layer

Is there some ideal word-embedding space that would perfectly map human language and could be used for any natural language processing task? Possibly, but we have yet to compute anything of the sort. Also, there is no such a thing as *human language*—there are many different languages, and they aren't isomorphic to one another, because a language is the reflection of a specific culture and a specific context. But more pragmatically, what makes a good word-embedding space depends heavily on your task: the perfect word-embedding space for an English-language movie-review sentiment-analysis model may look different from the perfect embedding space for an English-language legal-document classification model, because the importance of certain semantic relationships varies from task to task.

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.

The `Embedding` layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, looks up these integers in an internal dictionary, and returns the associated vectors. It's effectively a dictionary lookup (see figure 11.4).

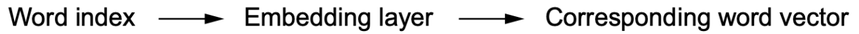


Figure 11.4 The `Embedding` layer

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)`.

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).

When you instantiate an `Embedding` layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer. During training, these word vectors are gradually adjusted via backpropagation, structuring the space into something the downstream model can exploit. Once fully trained, the embedding space will show a lot of structure—a kind of structure specialized for the specific problem for which you're training your model.

Let's build a model that includes an `Embedding` layer and benchmark it on our task.

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}")
```

It trains much faster than the one-hot model (since the LSTM only has to process 256-dimensional vectors instead of 20,000-dimensional), and its test accuracy is comparable (87%). However, we're still some way off from the results of our basic bigram model. Part of the reason why is simply that the model is looking at slightly less data: the bigram model processed full reviews, while our sequence model truncates sequences after 600 words.

Práctica 9 - Embedding LSTM

Intituto Politécnico Nacional

Centro de Investigación en Computación

Departamento de Diplomados y Extensión Profesional

Diplomado en Inteligencia Artificial

- Módulo 11 - Parte II

Profesor: Alan Badillo Salas (badillosalas@outlook.com)

RECURSOS Y CONTACTO

Alan Badillo Salas

badillosalas@outlook.com

https://github.com/dragonnomada/ipn-cic-diplomado-ia-2025