RNN Sentiment Analysis

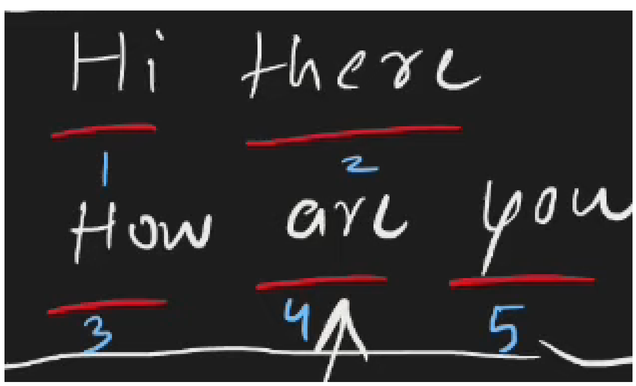
- First, convert the text data in **vectors/numbers**

2 Techniques:

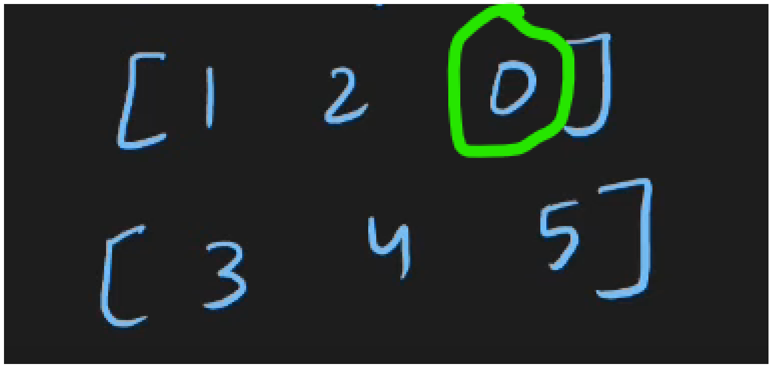
1. Integer Encoding
2. **Embeddings**

Integer Encoding

- You form a vocabulary of unique words
- Provide a integer value to each word



- Replace the sentences with their integer values
- Padding → Make the size same for all the sentences
  - Add zero



Python Code:

```
import numpy as np

docs = ['go india',
        'india india',
        'hip hip hurray',
        'jeetega bhai jeetega india jeetega',
        'bharat mata ki jai',
        'kohli kohli',
        'sachin sachin',
        'dhoni dhoni',
        'modi ji ki jai',
        'inquilab zindabad']
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(oov_token='<nothing>')
```

`Tokenizer` is used to **convert text into numbers** so that you can feed it to neural networks (because models can't understand raw words).

`oov_token='<nothing>'`

- `oov` **= "out of vocabulary"**
- When you give the tokenizer some new text to process **that includes unknown words** (words it didn't see during training), it replaces those with the token `'<nothing>'`

So if your tokenizer was trained on:

```
texts = ["I love pizza"]
```

Then you give it:

```
"I love sushi"
```

It will map `"sushi"` to `'<nothing>'`, since `"sushi"` wasn't in the original vocab.

Fit:

```
tokenizer.fit_on_texts(docs)
```

```
tokenizer.word_index
```

```
{'<nothing>': 1,
 'india': 2,
 'jeetega': 3,
 'hip': 4,
 'ki': 5,
 'jai': 6,
 'kohli': 7,
 'sachin': 8,
 'dhoni': 9,
 'go': 10,
 'hurray': 11,
 'bhai': 12,
 'bharat': 13,
 'mata': 14,
 'modi': 15,
 'ji': 16,
 'inquilab': 17,
 'zindabad': 18}
```

Word Count:

```
tokenizer.word_counts
```

```
OrderedDict([('go', 1),
             ('india', 4),
             ('hip', 2),
             ('hurray', 1),
             ('jeetega', 3),
             ('bhai', 1),
             ('bharat', 1),
             ('mata', 1),
             ('ki', 2),
             ('jai', 2),
             ('kohli', 2),
             ('sachin', 2),
             ('dhoni', 2),
             ('modi', 1),
             ('ji', 1),
             ('inquilab', 1),
             ('zindabad', 1)])
```

No. of rows in document:

```
tokenizer.document_count

10
```

Generate sequences:

```
sequences = tokenizer.texts_to_sequences(docs)
sequences
```

```
[[10, 2],
 [2, 2],
 [4, 4, 11],
 [3, 12, 3, 2, 3],
 [13, 14, 5, 6],
 [7, 7],
 [8, 8],
 [9, 9],
 [15, 16, 5, 6],
 [17, 18]]
```

`'go': 10, india': 2` : go india → `[10,2]`

Padding:

`from keras.` `utils import pad_sequences`

```
sequences = pad_sequences(sequences,padding='post')
sequences
```

```
array([[10,  2,  0,  0,  0],
       [ 2,  2,  0,  0,  0],
       [ 4,  4, 11,  0,  0],
       [ 3, 12,  3,  2,  3],
       [13, 14,  5,  6,  0],
       [ 7,  7,  0,  0,  0],
       [ 8,  8,  0,  0,  0],
       [ 9,  9,  0,  0,  0],
       [15, 16,  5,  6,  0],
       [17, 18,  0,  0,  0]])
```

`padding='post'` → Zeros will be added at the end

IMDB Sentiment Analysis

```
from keras.datasets import imdb
from keras import Sequential
```

```
from keras.layers import Dense,SimpleRNN,Embedding,Flatten
```

```
(X_train,y_train),(X_test,y_test) = imdb.load_data()
```

```
X_train
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5,
25, 100, 43, 838, 112, 50, 670, 22665, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 21631, 336, 385,
39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920,
4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626,
18, 19193, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12,
16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407,
```

> **The data is already preprocessed & integer encoded.**

Pad & Trim the review to 50 words to save time:

```
X_train = pad_sequences(X_train,padding='post',maxlen=50)
X_test = pad_sequences(X_test,padding='post',maxlen=50)
```

```
X_train.shape

(25000, 50)
```

Build a Model:

```
model = Sequential()

model.add(SimpleRNN(32,input_shape=(50,1),return_sequences=False))
model.add(Dense(1,activation='sigmoid'))
```

```
model.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ simple_rnn (SimpleRNN)          │ (None, 32)             │         1,088 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 1)              │            33 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 1,121 (4.38 KB)

 Trainable params: 1,121 (4.38 KB)

 Non-trainable params: 0 (0.00 B)
```

Parameter	Meaning
`SimpleRNN(32)`	Adds a basic RNN layer with **32 units** (neurons) — it outputs a vector of size `32`.
`input_shape=(50, 1)`	This means the input is a sequence of **50 time steps**, and each step has **1 feature** (like 1 number at each time step).
`return_sequences=False`	This means the RNN will only return the **last output** (not the full sequence). Useful for things like classification.

`input_shape=(50,1)`

- `50` → Numbers (Timesteps)
    - 50 words in a sentence
- `1` → Number of **features** at each time step

✅ `return_sequences=False`

> ➤ Only returns the last output in the sequence.

👀 So for input shape `(batch_size, 50, 1)`:

- You give the RNN **a sequence of 50 time steps**
- It **processes all 50 steps**, but only returns the **final output at time step 50**

```
Input:     x₁ → x₂ → x₃ → ... → x₅₀
Hidden:     h₁   h₂   h₃       h₅₀
Output:                         ↑
                     Return only h₅₀
```

➡️ Output shape: (batch_size, units)
(for example:
`(None, 32)` if you used `SimpleRNN(32)`)

✅ `return_sequences=True`

> ➤ Returns all outputs from all time steps

```
Input:     x₁ → x₂ → x₃ → ... → x₅₀
Output:    h₁   h₂   h₃       h₅₀
```

➡️ Output shape:
`(batch_size, time_steps, units)`
(e.g.
`(None, 50, 32)`)

🤔 So when to use which?

Use Case	`return_sequences`
Text classification	`False` ✔️ (only final output matters)

Use Case	`return_sequences`
Many-to-one (e.g. sentiment)	`False` ✔
Many-to-many (e.g. translation)	`True` ✔ (you need output from each time step)
Stacked RNN layers	`True` on first RNN, `False` on last ✔

Compile:

```
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

model.fit(X_train,y_train,epochs=5,validation_data=(X_test,y_test))
```

```
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.5041 - loss: 0.7097 - val_accuracy: 0.5078 - val_loss: 0.6940
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.5053 - loss: 0.6930 - val_accuracy: 0.5070 - val_loss: 0.6937
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.5122 - loss: 0.6926 - val_accuracy: 0.5054 - val_loss: 0.6946
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.5072 - loss: 0.6926 - val_accuracy: 0.5074 - val_loss: 0.6932
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.5094 - loss: 0.6919 - val_accuracy: 0.5079 - val_loss: 0.6931
```

Embedding

What are Embeddings?

- Embeddings are a technique for representing discrete variables (like words or categories) as continuous vectors in a lower-dimensional space.
- They capture semantic relationships and patterns in the data.

Key Characteristics:

1. **Dense Representation**: Convert sparse one-hot encodings into dense vectors
2. **Lower Dimensionality**: Reduce dimensionality compared to one-hot encoding
3. **Learned Semantics**: Similar items get similar vector representations
4. **Task-Specific**: Embeddings are learned for specific tasks/datasets

❌ Wrong way: One-Hot Encoding

A simple way is to give every word a unique index, then use **one-hot vectors**:

- "cat" → `[0, 0, 1, 0, 0]`
- "dog" → `[0, 1, 0, 0, 0]`

But this has **major problems**:

- Vectors are long (if you have 10,000 words, each vector is 10,000 long).
- All vectors are orthogonal (completely unrelated).
- No **semantic meaning** (cat and dog are both animals, but one-hot can't capture that).

✅ Better way: Embeddings

Embeddings solve these problems by:

- Mapping each word to a **dense vector** of smaller dimension (e.g., 50 or 100).
  - **dense vector → Has only non-zero values**
- Learning those vectors so that **similar words have similar vectors**.

For example:

- "cat" → `[0.2, -0.1, 0.7]`
- "dog" → `[0.21, -0.09, 0.68]` (very close)
- "car" → `[0.8, 0.3, -0.6]` (very different)

🔷 How Does an Embedding Work (Technically)?

Let's say:

- You have a vocabulary of **10,000 words.**
- You want embeddings of size **50**.

You create a **matrix of size (10,000 × 50)**. Each **row corresponds to one word's vector**.

This is called the **embedding matrix**.

Then:

- "cat" = word index 123
- Its embedding is just: **the 123rd row of the matrix**

The matrix is stored as a **lookup table** — like a dictionary.

During training, this matrix is updated (via backpropagation) so that the vectors learn **meaningful positions**.

🔷 How Are Embeddings Learned?

1. **Initially**, all vectors are random.
2. During training, your model (e.g., RNN or transformer) tries to minimize some loss (e.g., predicting next word).
3. Backpropagation updates the embedding matrix so that words that behave similarly in context are pulled closer.

This means embeddings are **not fixed** (unless you freeze them). They are **learned** like any other neural net weight.

Pretrained Embeddings

Instead of learning from scratch, you can use pretrained embeddings trained on huge corpora:

- **Word2Vec**
- **GloVe**
- **fastText**
- **BERT embeddings** (contextual, dynamic)

This is useful when you have **small datasets** or want better generalization.

Visual Analogy

Think of a **city map**:

- Each word is a **house**.
- Embedding gives each house a **GPS coordinate** on a 3D map.
- Words that live in the same “semantic neighborhood” are **close together**.

Embedding Properties

1. **Dimensionality**: Typical sizes:
    - Word embeddings: 50-300 dimensions
    - Category embeddings: 10-50 dimensions
2. **Interpretability**: Dimensions may capture meaningful features
3. **Algebraic Properties**: Can perform vector arithmetic (e.g., king - man + woman ≈ queen)

Python code for Embedding Layers

> 💡 **Make sure the data entered is integer encoded.**

Integer encoding: 👇

```
docs = ['go india',
        'india india',
        'hip hip hurray',
        'jeetega bhai jeetega india jeetega',
        'bharat mata ki jai',
        'kohli kohli',
        'sachin sachin',
        'dhoni dhoni',
```

```
        'modi ji ki jai',
        'inquilab zindabad']
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer()
```

```
tokenizer.fit_on_texts(docs)
```

```
tokenizer.word_index
```

```
{'india': 1,
 'jeetega': 2,
 'hip': 3,
 'ki': 4,
 'jai': 5,
 'kohli': 6,
 'sachin': 7,
 'dhoni': 8,
 'go': 9,
 'hurray': 10,
 'bhai': 11,
 'bharat': 12,
 'mata': 13,
 'modi': 14,
 'ji': 15,
 'inquilab': 16,
 'zindabad': 17}
```

```
len(tokenizer.word_index)

17
```

- There are 17 unique words

Generate sequences:

```
sequences = tokenizer.texts_to_sequences(docs)
sequences
```

```
[[9, 1],
 [1, 1],
 [3, 3, 10],
 [2, 11, 2, 1, 2],
 [12, 13, 4, 5],
 [6, 6],
 [7, 7],
 [8, 8],
 [14, 15, 4, 5],
 [16, 17]]
```

Padding:

```
from keras.utils import pad_sequences
sequences = pad_sequences(sequences,padding='post')
sequences
```

```
array([[ 9,  1,  0,  0,  0],
       [ 1,  1,  0,  0,  0],
       [ 3,  3, 10,  0,  0],
       [ 2, 11,  2,  1,  2],
       [12, 13,  4,  5,  0],
       [ 6,  6,  0,  0,  0],
       [ 7,  7,  0,  0,  0],
       [ 8,  8,  0,  0,  0],
       [14, 15,  4,  5,  0],
       [16, 17,  0,  0,  0]])
```

Add a Embedding layer in model:

```
from keras import Sequential
from keras.layers import Embedding

model = Sequential()
model.add(Embedding(input_dim=18,output_dim=2)

model.summary()
```

```
Model: "sequential_2"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding_2 (Embedding)         │ (None, 17, 2)          │            34 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 34 (136.00 B)

 Trainable params: 34 (136.00 B)

 Non-trainable params: 0 (0.00 B)
```

`input_dim=18` → This integer represents the **size of the vocabulary**.

- In simpler terms, it's the **total number of unique words** or items in your dataset that you want to embed.

`output_dim=2` → This integer specifies the **dimensionality of the embedding space**.

- It determines the size of the dense vector that will be used to represent each word or item from your vocabulary.

- In this case, each of the 17 unique items will be mapped to a 2-dimensional vector. For example, the first item (index 0) might be represented by the vector `[0.1, -0.5]`, the second item (index 1) by `[0.8, 0.2]`, and so on.
- **A higher `output_dim` allows the model to learn more complex relationships and capture more semantic information about the input items. However, it also increases the number of parameters in the model.**

`input_length=5` → This integer defines the **length of the input sequences**. It tells the embedding layer that the input it will receive will consist of sequences of 5 integers each.

- For example, one input to this layer might be the sequence `[3, 1, 9, 0, 15]`, where each number is an index referring to one of the 17 vocabulary items.

> 💡 !! **Delete this. It's deprecated.**

Compile:

```
model.compile('adam','accuracy')
```

Predict:

```
pred = model.predict(sequences)
print(pred)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
[[[ 0.04850895  0.01730514]
  [ 0.00802112 -0.03806441]
  [ 0.01159269  0.04719603]      Sentence 1
  [ 0.01159269  0.04719603]
  [ 0.01159269  0.04719603]]

 [[ 0.00802112 -0.03806441]
  [ 0.00802112 -0.03806441]
  [ 0.01159269  0.04719603]
  [ 0.01159269  0.04719603]      Sentence 2
  [ 0.01159269  0.04719603]]

 [[-0.01268284 -0.04667665]
  [-0.01268284 -0.04667665]
  [-0.02775811 -0.03309484]      Sentence 3
  [ 0.01159269  0.04719603]
  [ 0.01159269  0.04719603]]
```

IMDB Prediction using embedding:

```
from keras.datasets import imdb
from tensorflow.keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences
from keras import Sequential
from keras.layers import Dense,SimpleRNN,Embedding,Flatten
```

Load data:

```
(X_train,y_train),(X_test,y_test) = imdb.load_data(num_words=10000))
```

`num_words=10000` → We are taking top 10000 words

Pad & Trim the review to 50 words to save time:

```
X_train = pad_sequences(X_train,padding='post',maxlen=50)
```

```
X_test = pad_sequences(X_test,padding='post',maxlen=50)
```

```
X_train.shape

(25000, 50)
```

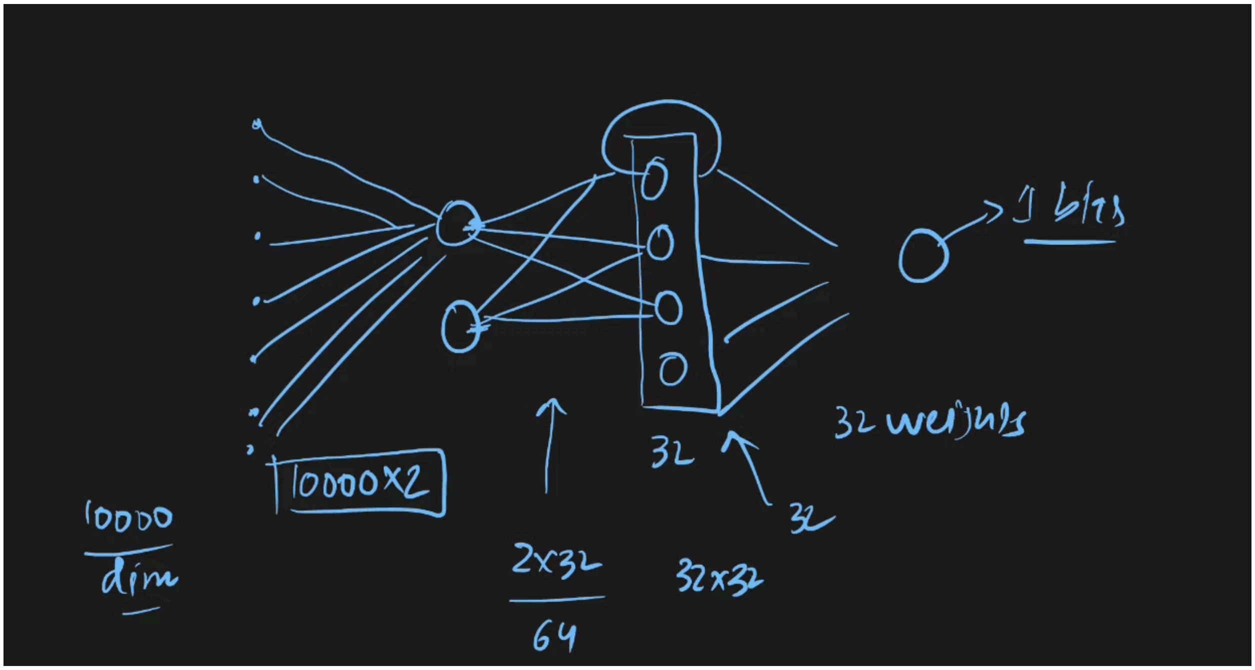
Build a Model:

```
model = Sequential()
model.add(Embedding(10000,2))
model.add(SimpleRNN(32,return_sequences=False))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
Model: "sequential_27"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding_25 (Embedding)         │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ simple_rnn_17 (SimpleRNN)        │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_17 (Dense)                 │ ?                      │   0 (unbuilt) │
└──────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 0 (0.00 B)

 Trainable params: 0 (0.00 B)

 Non-trainable params: 0 (0.00 B)
```

`10000` → Vocab size

`2` → For every word, I want a dense vector of dimension 2



Compile:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
```

Clip X_train & X_test to 9999

Predict:

```
history = model.fit(X_train, y_train,epochs=5,validation_data=(X_test,y_test))
```

```
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - acc: 0.5738 - loss: 0.6550 - val_acc: 0.7974 - val_loss: 0.4471
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - acc: 0.8343 - loss: 0.3843 - val_acc: 0.8084 - val_loss: 0.4257
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - acc: 0.8760 - loss: 0.3116 - val_acc: 0.8086 - val_loss: 0.4321
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - acc: 0.8925 - loss: 0.2778 - val_acc: 0.8027 - val_loss: 0.4553
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - acc: 0.9064 - loss: 0.2440 - val_acc: 0.7993 - val_loss: 0.4631
```

Now, call `model.summary()` again:

```
model.summary()
```

```
Model: "sequential_29"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape             ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding_27 (Embedding)         │ (None, 50, 2)            │        20,000 │
├──────────────────────────────────┼──────────────────────────┼───────────────┤
│ simple_rnn_18 (SimpleRNN)        │ (None, 32)               │         1,120 │
├──────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_18 (Dense)                 │ (None, 1)                │            33 │
└──────────────────────────────────┴──────────────────────────┴───────────────┘

 Total params: 63,461 (247.90 KB)

 Trainable params: 21,153 (82.63 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 42,308 (165.27 KB)
```

- This time you see the parameters