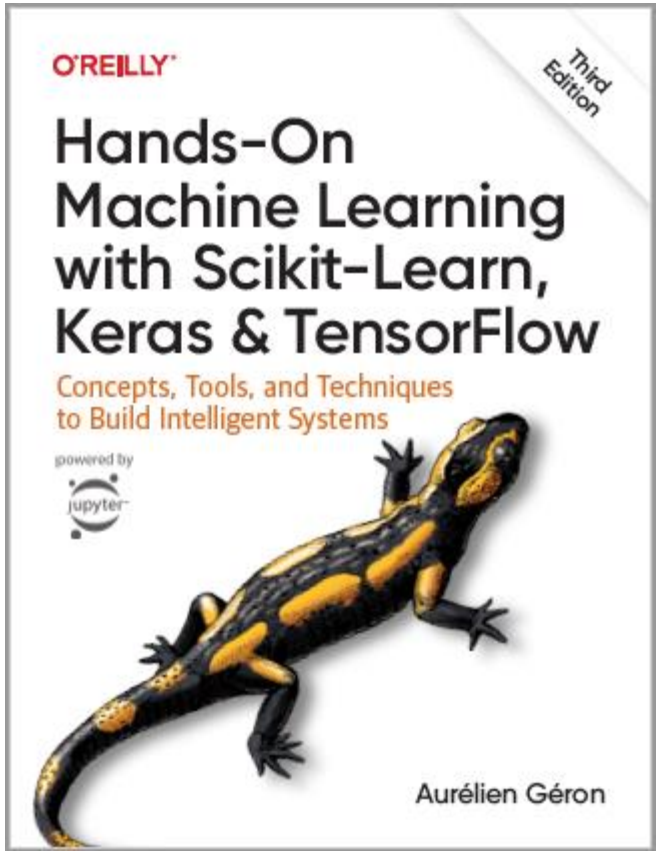
Natural Language Processing with RNNs and Attention

Prof. Gheith Abandah

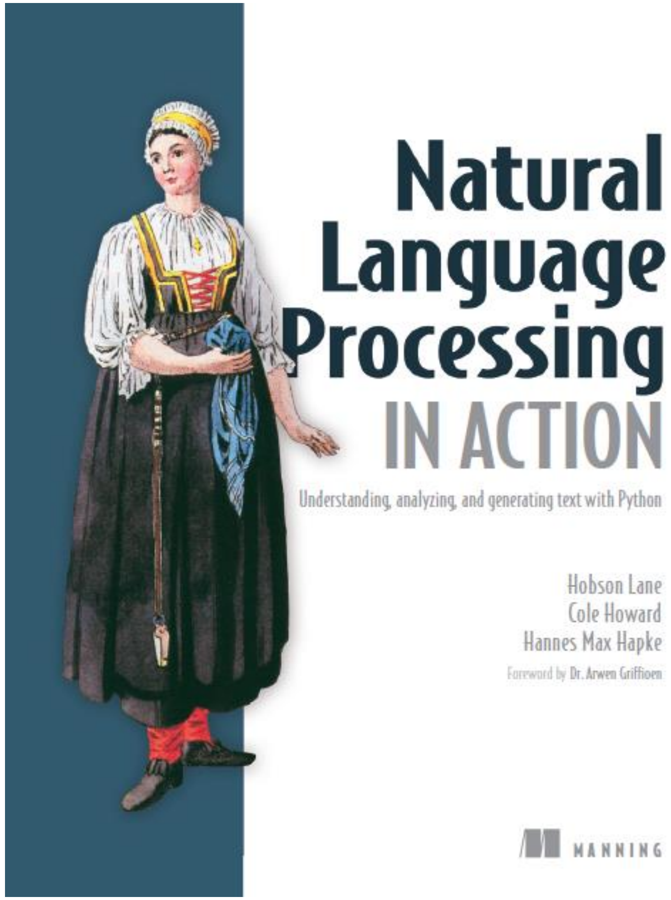
Reference 1

- Chapter 16: **Natural Language Processing with RNNs and Attention**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 3rd Edition, 2022
  - Material: https://github.com/ageron/handson-ml3

Reference 2



- Chapter 10: **Sequence-to-sequence models and attention**

- H. Lane, C. Howard, and H. Hapke, **Natural Language Processing in Action:** Understanding, analyzing, and generating text with Python, Manning, 2019.

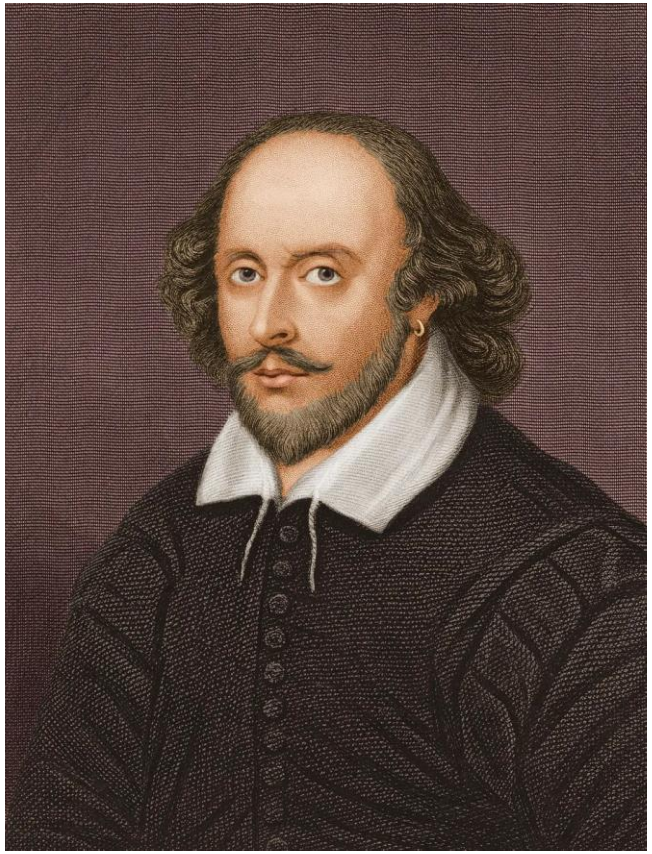
Outline

1. Generating Shakespearean Text Using a Character RNN
2. Sentiment Analysis
3. An Encoder–Decoder Network for Neural Machine Translation
4. Attention Mechanisms
5. Transformer Models
6. Summary

1. Generating Shakespearean Text Using a Character RNN

Procedure

1. Creating the training dataset
2. Building and training the char-RNN model
3. Generating fake Shakespearean text
4. Stateful RNN



1.1 Creating the Training Dataset

1. Download and read **`shakespeare.txt`**.
2. Split into characters and encode the characters.
3. Convert the long sequence of character IDs into input/target window pairs.

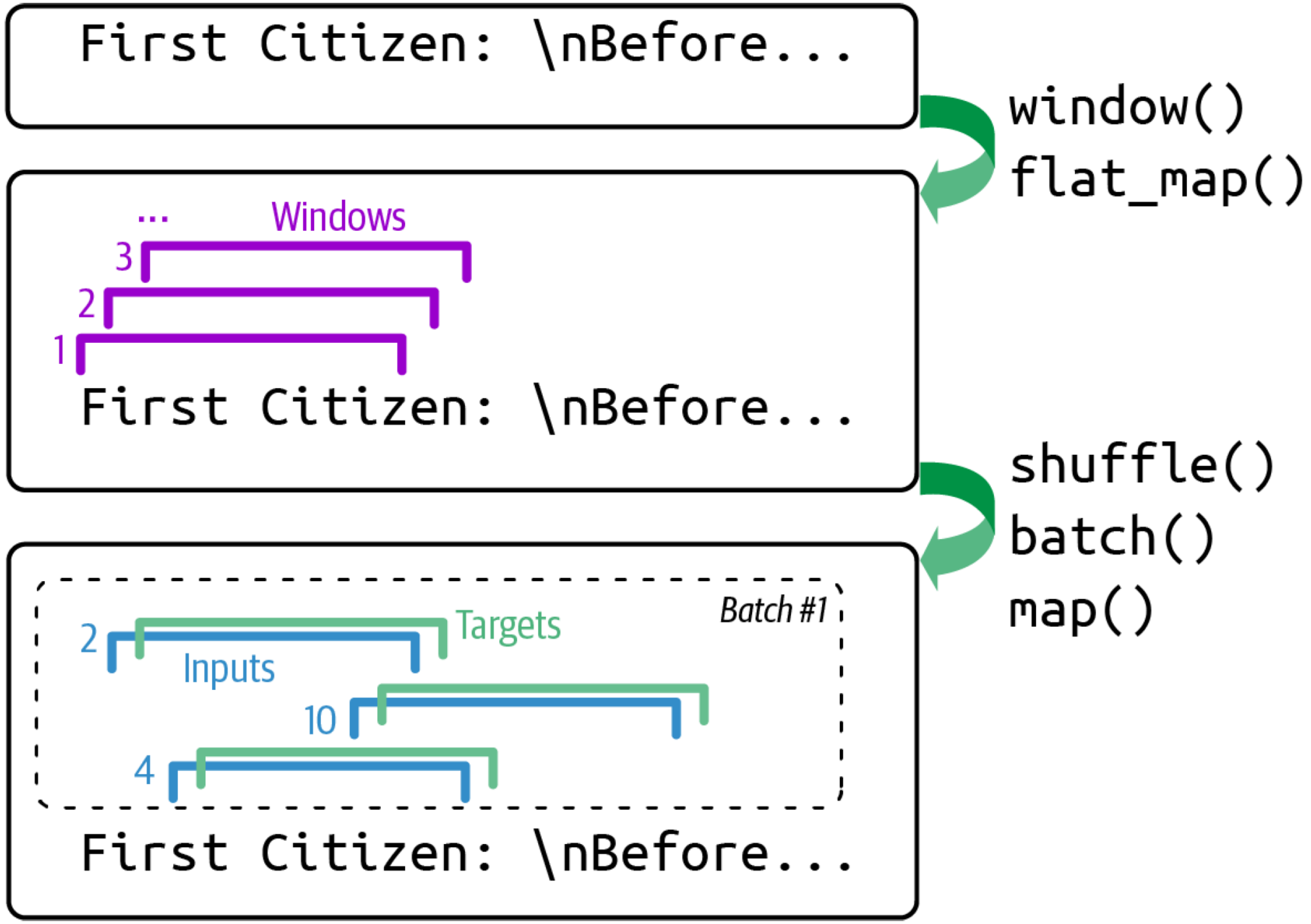
Download and read **`shakespeare.txt`**.

```
import tensorflow as tf

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

print(shakespeare_text[:80])
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

Split into characters and encode the characters.

```
text_vec_layer =
tf.keras.layers.TextVectorization(split="character",
                                  standardize="lower")

text_vec_layer.adapt([shakespeare_text])
encoded = text_vec_layer([shakespeare_text])[0]

encoded -= 2  # drop tokens 0 (pad) and 1 (unknown),
              #  which we will not use
n_tokens = text_vec_layer.vocabulary_size() - 2
              # number of distinct chars = 39
dataset_size = len(encoded)
              # total number of chars = 1,115,394
```

Convert the long sequence of character IDs into input/target window pairs.



Convert the long sequence of character IDs into input/target window pairs.

```
# Function to convert a long sequence of character IDs
#   into a dataset of input/target window pairs
def to_dataset(sequence, length, shuffle=False,
               seed=None, batch_size=32):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=1,
                   drop_remainder=True)
    ds = ds.flat_map(lambda window_ds:
                     window_ds.batch(length + 1))
    if shuffle:
        ds = ds.shuffle(100_000, seed=seed)
    ds = ds.batch(batch_size)
    return ds.map(lambda window: (window[:, :-1],
                  window[:, 1:])).prefetch(1)
```

Convert the long sequence of character IDs into input/target window pairs.

```
length = 100
tf.random.set_seed(42)
train_set = to_dataset(encoded[:1_000_000],
                       length=length, shuffle=True,
                       seed=42)
valid_set = to_dataset(encoded[1_000_000:1_060_000],
                       length=length)
test_set = to_dataset(encoded[1_060_000:],
                      length=length)
```

1.2 Building and Training the Char-RNN Model

```
model = Sequential([
    Embedding(input_dim=n_tokens, output_dim=16),
    GRU(128, return_sequences=True),
    Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="nadam",
              metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy",
    save_best_only=True)
history = model.fit(train_set,
                    validation_data=valid_set,
                    epochs=10,
                    callbacks=[model_ckpt])
```

1.3 Generating Fake Shakespearean Text

- RNN output generation is often deterministic, producing the most probable next token.
- Deterministic outputs may lead to repetitive or predictable sequences.
- Randomness can be introduced to diversify output and improve creativity.
- **Temperature** parameter controls the level of randomness in output generation.
- **Low temperature**
  - Produces more confident predictions.
  - Higher probability tokens are favored, leading to more deterministic output.
- **High temperature**
  - Increases randomness.
  - Allows lower probability tokens to have a higher chance of being selected.

Direct text generation

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Lambda(lambda X: X - 2), # no PAD or UNK
    model
])

y_proba = shakespeare_model.predict(
            ["To be or not to b"])[0, -1]
y_pred = tf.argmax(y_proba)
         # choose the most probable character ID
text_vec_layer.get_vocabulary()[y_pred + 2]
e

# Problem: Predicts the same sequence always
```

Functions to pick the next char and extend a text.

```
def next_char(text, temperature=1):
    y_proba = shakespeare_model.predict([text])[0, -1:]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits,
                                    num_samples=1)[0, 0]
    return text_vec_layer.get_vocabulary()[char_id + 2]

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

Experimenting with temperature

```
print(extend_text("To be or not to be", temperature=0.01))
```

```
To be or not to be the duke
as it is a proper strange death,
and the
```

```
print(extend_text("To be or not to be", temperature=1))
```

```
To be or not to behold?
second push:
gremio, lord all, a sistermen,
```

```
print(extend_text("To be or not to be", temperature=100))
```

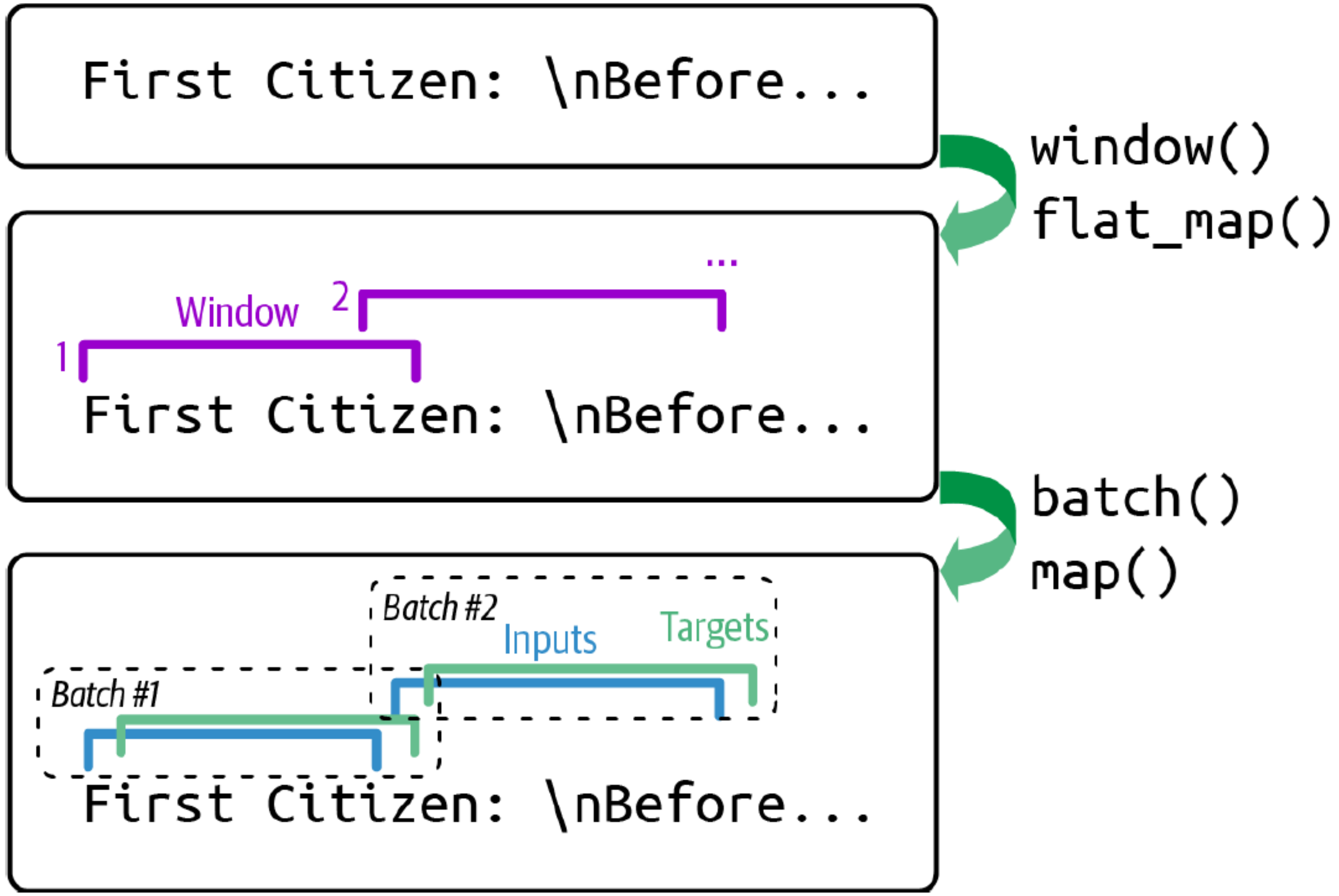
```
To be or not to bef ,mt'&o3fpadm!$
wh!nse?bws3est--vgerdjw?c-y-ewznq
```

1.4 Stateful RNN

- **Stateless RNNs**: at each training iteration the model starts with a hidden state full of zeros.
- **Stateful RNN**: preserve this final state after processing a training batch and use it as the initial state for the next training batch.
- The model learns long-term patterns despite only backpropagating through short sequences.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens,
                              output_dim=16,
                              batch_input_shape=[1, None]),
    tf.keras.layers.GRU(128, return_sequences=True,
                        stateful=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

Preparing a dataset of consecutive sequence fragments for a stateful RNN



Preparing a dataset of consecutive sequence fragments for a stateful RNN

```
def to_dataset_for_stateful_rnn(sequence, length):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=length,
                   drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(length
                      + 1)).batch(1)
    return ds.map(lambda window: (window[:, :-1],
                   window[:, 1:])).prefetch(1)
```

Training the stateful RNN

```
# At the end of each epoch, we need to reset the states before
#  we go back to the beginning of the text.
class ResetStatesCallback(tf.keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="nadam", metrics=["accuracy"])

history = model.fit(stateful_train_set,
                    validation_data=stateful_valid_set,
                    epochs=10, callbacks=[ResetStatesCallback(),
                                          model_ckpt])
```

Outline

1. Generating Shakespearean Text Using a Character RNN
2. Sentiment Analysis
3. An Encoder–Decoder Network for Neural Machine Translation
4. Attention Mechanisms
5. Transformer Models
6. Summary

2. Sentiment Analysis

- IMDb movie reviews

Procedure

1. Creating the training dataset
2. Building and training the RNN model
3. Masking
4. Reusing pretrained embeddings and language models

2.1 Creating the training dataset

```
import tensorflow_datasets as tfds
# The IMDb dataset has 50,000 movie reviews in English
#  25,000 for training, 25,000 for testing
raw_train_set, raw_valid_set, raw_test_set = tfds.load(
    name="imdb_reviews",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
train_set = raw_train_set.shuffle(5000,
    seed=42).batch(32).prefetch(1)
valid_set = raw_valid_set.batch(32).prefetch(1)
test_set = raw_test_set.batch(32).prefetch(1)
```

2.1 Creating the training dataset

```
# Simple tokenization using spaces for token boundaries
# Limit the vocabulary to 1,000 tokens
# Very rare words are not important for this task
vocab_size = 1000

text_vec_layer =
tf.keras.layers.TextVectorization(max_tokens=vocab_size)

text_vec_layer.adapt(train_set.map(lambda reviews,
                                   labels: reviews))
```

2.2 Building and training the RNN model

```
embed_size = 128
model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size),
    tf.keras.layers.GRU(128),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy",
              optimizer="nadam", metrics=["accuracy"])
history = model.fit(train_set,
                    validation_data=valid_set, epochs=2)
```

2.3 Masking

- The accuracy of the previous model is only about 50%.
- When `TextVectorization` converts reviews to sequences of token IDs, it pads the shorter sequences using the padding token (with ID 0).
- When the `GRU` layer goes through many padding tokens, it ends up forgetting what the review was about!
- **Masking** makes the model ignore the padding tokens.

2.3 Masking

```
# Validation accuracy = 87% after 5 epochs
embed_size = 128

model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size,
                              mask_zero=True),
    tf.keras.layers.GRU(128, dropout=0.2),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
```

2.4 Reusing pretrained embeddings and language models

- Can use **Google's Universal Sentence Encoder**
- **Task**: Encodes text into high-dimensional vectors for various NLP tasks like classification, similarity, clustering.
- **Input**: Variable length English text (sentences, phrases, short paragraphs).
- **Output**: 512-dimensional vector capturing text meaning.
- **Training**: Optimized for sentences, trained on diverse data sources and tasks for broad NLP applicability.
- **Advantage**: Models meaning of entire sequences, not just individual words (compared to word embedding models).
- Available on **TensorFlow Hub Library** (https://tensorflow.org/hub).

2.4 Reusing pretrained embeddings and language models

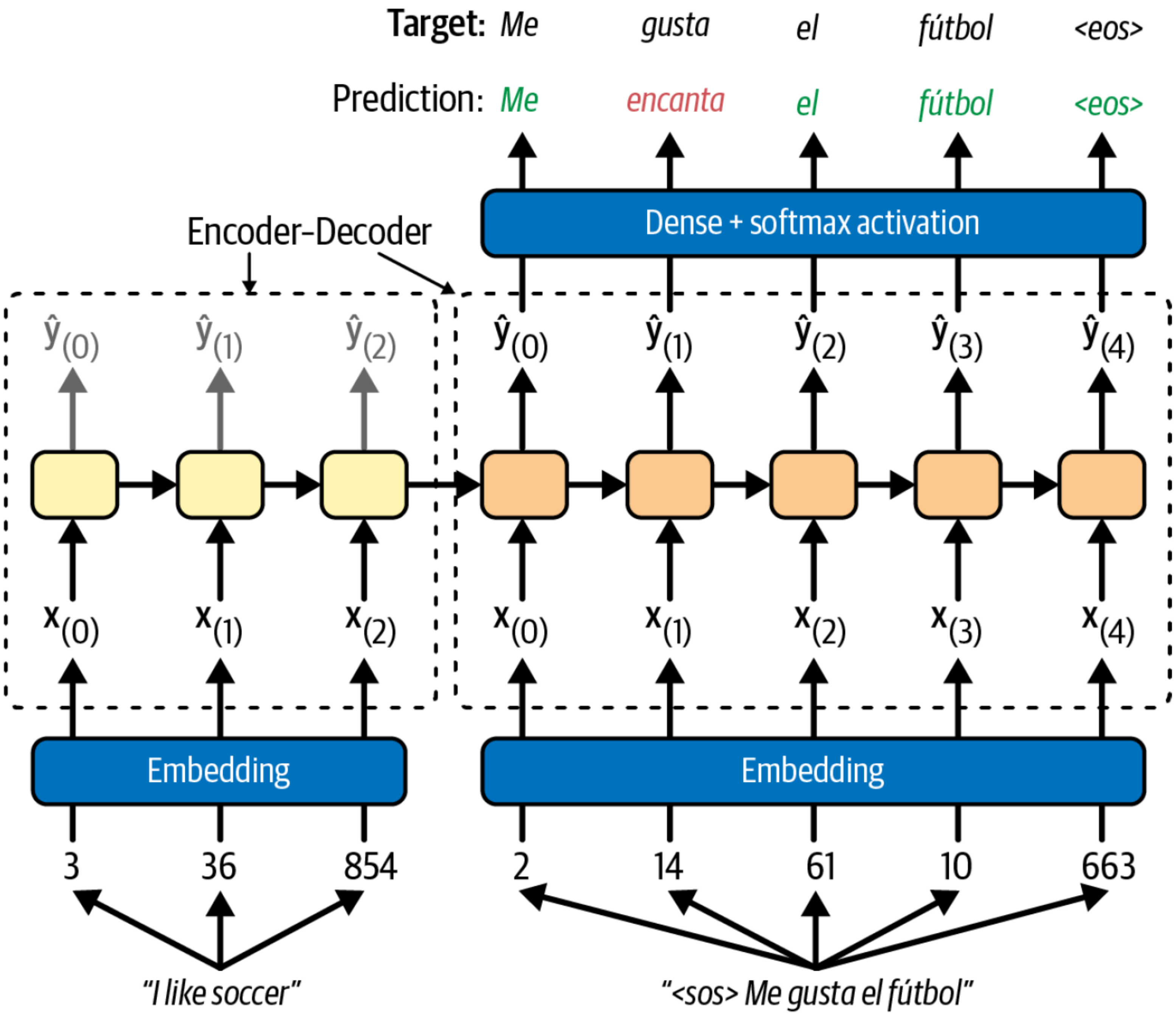
```
# Validation accuracy = 90% after 10 epochs
import os
import tensorflow_hub as hub

os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
url = "https://tfhub.dev/google/universal-sentence-
encoder/4"
model = tf.keras.Sequential([
    hub.KerasLayer(url, trainable=True, dtype=tf.string,
                   input_shape=[]),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
```

Outline

1. Generating Shakespearean Text Using a Character RNN
2. Sentiment Analysis
3. An Encoder–Decoder Network for Neural Machine Translation
4. Attention Mechanisms
5. Transformer Models
6. Summary

3. An Encoder–Decoder Network for Neural Machine Translation

- **Encoder**: Analyzes source sentence (*e.g.*, English). Uses RNNs (like LSTMs) to capture word relationships and context. Summarizes the sentence into a "**context vector**."
- **Decoder**: Generates target sentence (*e.g.*, Spanish). Uses the context vector and predicts words one-by-one, considering previous predictions.
- **Benefits**
  - **High Accuracy**: Captures complex sentence structures and context better than traditional methods.
  - **Flexible**: Handles variable-length sentences and translates across different languages effectively.


Target: Me gusta el fútbol <eos>
Prediction: Me encanta el fútbol <eos>
Dense + softmax activation
Encoder-Decoder
ŷ(0) ŷ(1) ŷ(2)
ŷ(0) ŷ(1) ŷ(2) ŷ(3) ŷ(4)
x(0) x(1) x(2)
x(0) x(1) x(2) x(3) x(4)
Embedding
Embedding
3 36 854
2 14 61 10 663
"I like soccer"
"<sos> Me gusta el fútbol"

3. An Encoder–Decoder Network for Neural Machine Translation

- English to Spanish translation

Procedure

1. Creating the training dataset
2. Building and training the model
3. Translating English to Spanish
4. Bidirectional RNNs
5. Beam Search

3.1 Creating the training dataset

```
url =
"https://storage.googleapis.com/download.tensorflow.org/
data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip",
                               origin=url,
                               cache_dir="datasets",
                               extract=True)
text = (Path(path).with_name("spa-eng") /
        "spa.txt").read_text()
```

3.1 Creating the training dataset

```
# TextVectorization layer doesn't handle "¡" and "¿"
# Parse the sentence pairs, shuffle,
#  and split  into two separate lists
text = text.replace("¡", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs)
for i in range(3):
    print(sentences_en[i], "=>", sentences_es[i])
```

```
How boring! => Qué aburrimiento!
I love sports. => Adoro el deporte.
Would you like to swap jobs? => Te gustaría que
intercambiemos los trabajos?
```

3.1 Creating the training dataset

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in
                         sentences_es])

text_vec_layer_en.get_vocabulary()[:10]
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']

text_vec_layer_es.get_vocabulary()[:10]
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no',
'tom', 'la']
```

3.1 Creating the training dataset

```
# Split the sequences to train and validation sets
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in
                            sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in
                            sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in
                              sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in
                              sentences_es[100_000:]])
```

3.2 Building and training the model

```
# Encoder
encoder_inputs = tf.keras.layers.Input(shape=[],
                                       dtype=tf.string)
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(
    vocab_size, embed_size, mask_zero=True)
encoder_embeddings = encoder_embedding_layer(
    encoder_input_ids)
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(
    encoder_embeddings)
```

3.2 Building and training the model

```
# Decoder
decoder_inputs = tf.keras.layers.Input(shape=[],
                                       dtype=tf.string)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
decoder_embedding_layer = tf.keras.layers.Embedding(
    vocab_size, embed_size, mask_zero=True)
decoder_embeddings = decoder_embedding_layer(
    decoder_input_ids)
decoder = tf.keras.layers.LSTM(512,
                               return_sequences=True)
decoder_outputs = decoder(decoder_embeddings,
    initial_state=encoder_state)
output_layer = tf.keras.layers.Dense(vocab_size,
    activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

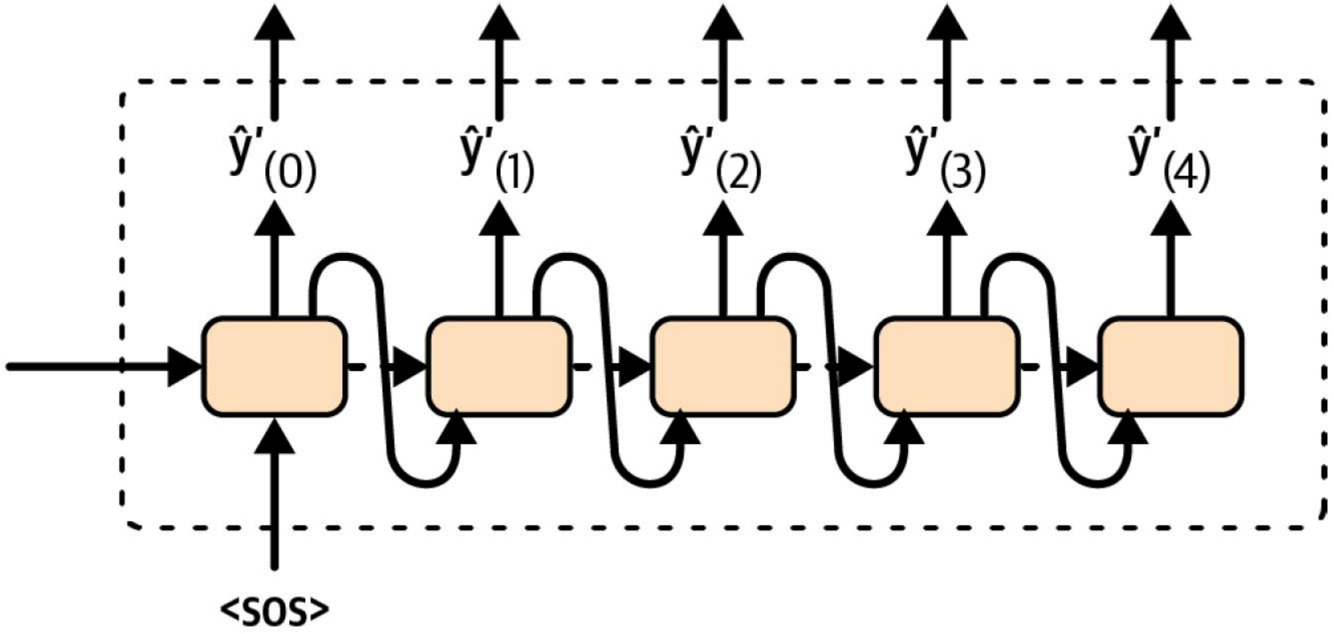
3.2 Building and training the model

```
model = tf.keras.Model(inputs=[encoder_inputs,
                               decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="nadam", metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec),
                           Y_valid))
```

accuracy: 0.8402, val_accuracy: 0.6763

3.3 Translating English to Spanish

- At inference time, the decoder is fed as input the word it just output at the previous time step.



3.3 Translating English to Spanish

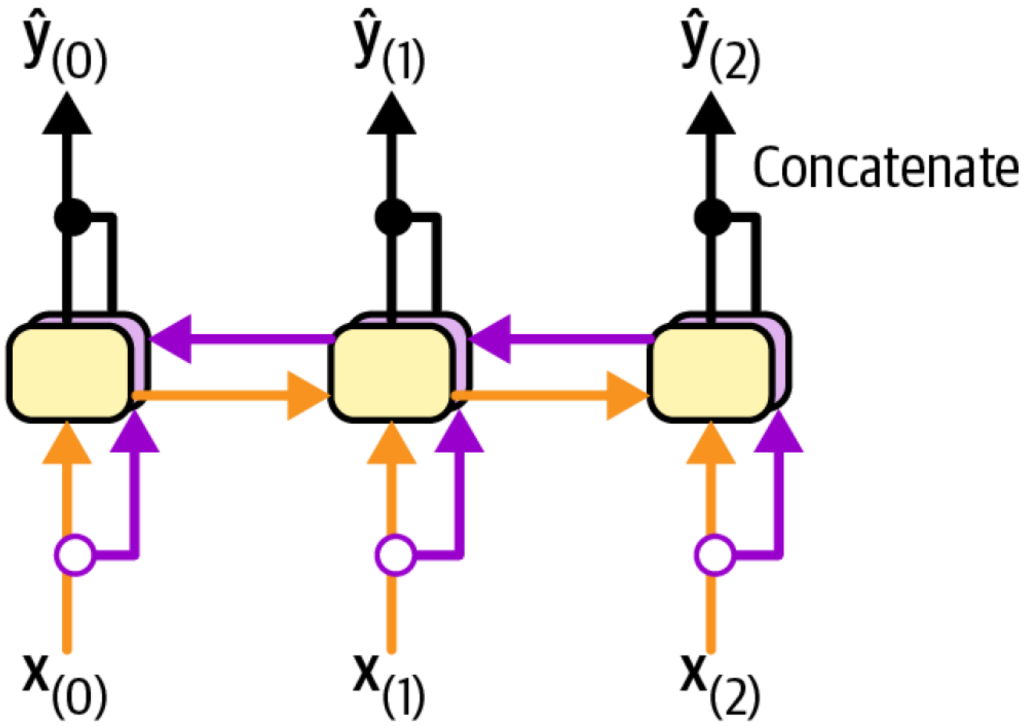
```
def translate(sentence_en):
    t = ""
    for word_idx in range(max_length):
        X = np.array([sentence_en])  # encoder input
        X_dec = np.array(["startofseq " + t])  # dec in
        y_proba = model.predict((X, X_dec))[0, word_idx]
                                        # last token's probas
        predicted_word_id = np.argmax(y_proba)
        predicted_word = text_vec_layer_es.get_vocabulary()
                                             [predicted_word_id]
        if predicted_word == "endofseq":
            break
        t += " " + predicted_word
    return t.strip()
```

3.3 Translating English to Spanish

```
# It works with very short sentences.
translate("I like soccer")
'me gusta el fútbol'

# It struggles with longer sentences.
translate("I like soccer and also going to the beach")
'me gusta el fútbol y a veces mismo al bus'
```

3.4 Bidirectional RNNs

- It is often useful to look ahead at the next words before encoding a given word, e.g., "the right arm", "the right person", and "the right to criticize."
- Use two recurrent layers on the same inputs, one reading the words from **left to right** and the other reading them from **right to left**, then combine their outputs at each time step.



3.4 Bidirectional RNNs

```
tf.random.set_seed(42)  # extra code – ensures
reproducibility on CPU
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_state=True))

# concatenate the two short-term states
#  and the two long-term states
encoder_outputs, *encoder_state = encoder(
    encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1),
                 # short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)]
                 # long-term (1 & 3)
```

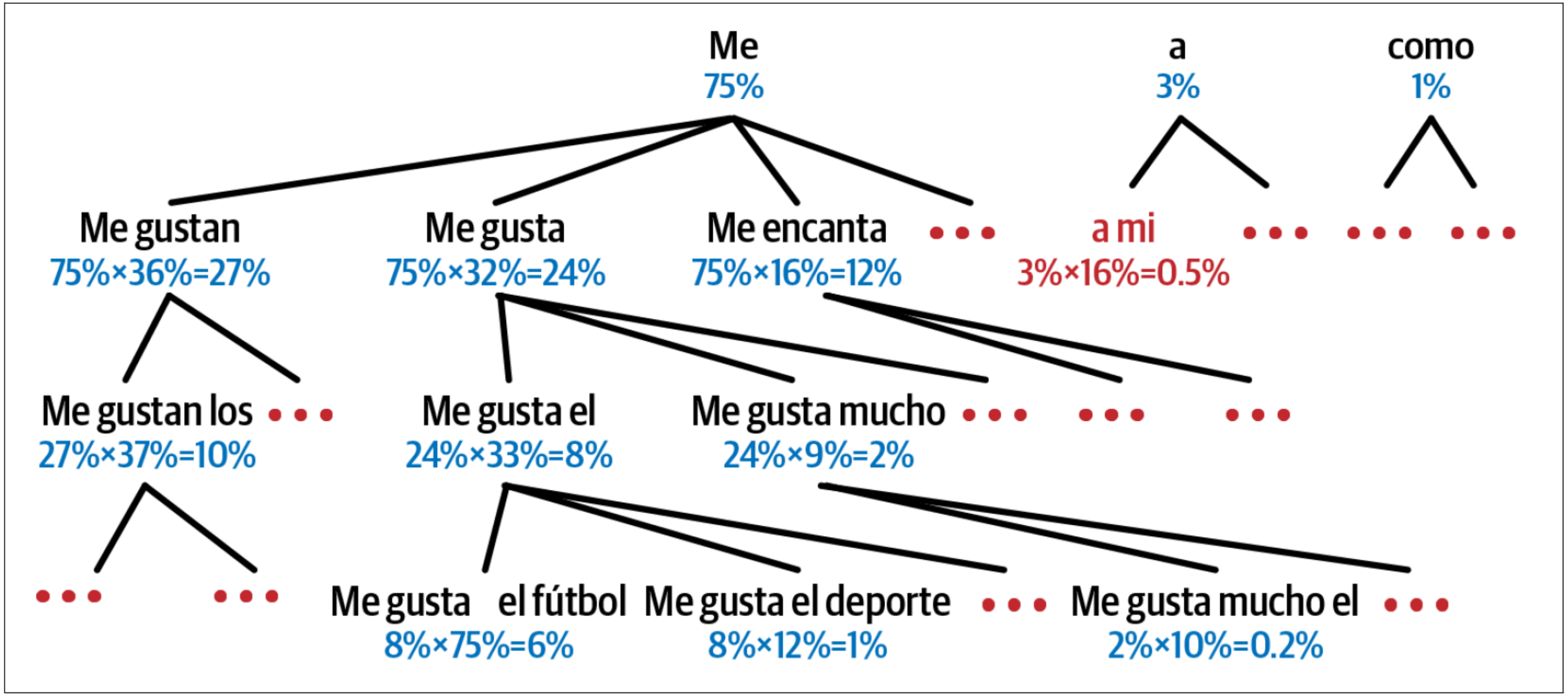
Reduced from 512

accuracy: 0.8577, val_accuracy: 0.6906

3.5 Beam Search

- **Greedy search** in NMT picks the most likely word at each step, potentially leading to locally optimal but bad translations.
- **Beam Search**: Explores multiple translation options simultaneously. Keeps a fixed number of (“beam width”) most probable partial translations at each step.
- **Benefits**
  - **Improved Fluency**: Considers diverse contexts, reducing the risk of getting stuck in poor translations.
  - **More Accurate**: Increases the chance of finding the overall best translation compared to greedy search.

3.5 Beam search of width 3 to translate "I like soccer" to "me gusta el fútbol"



Outline

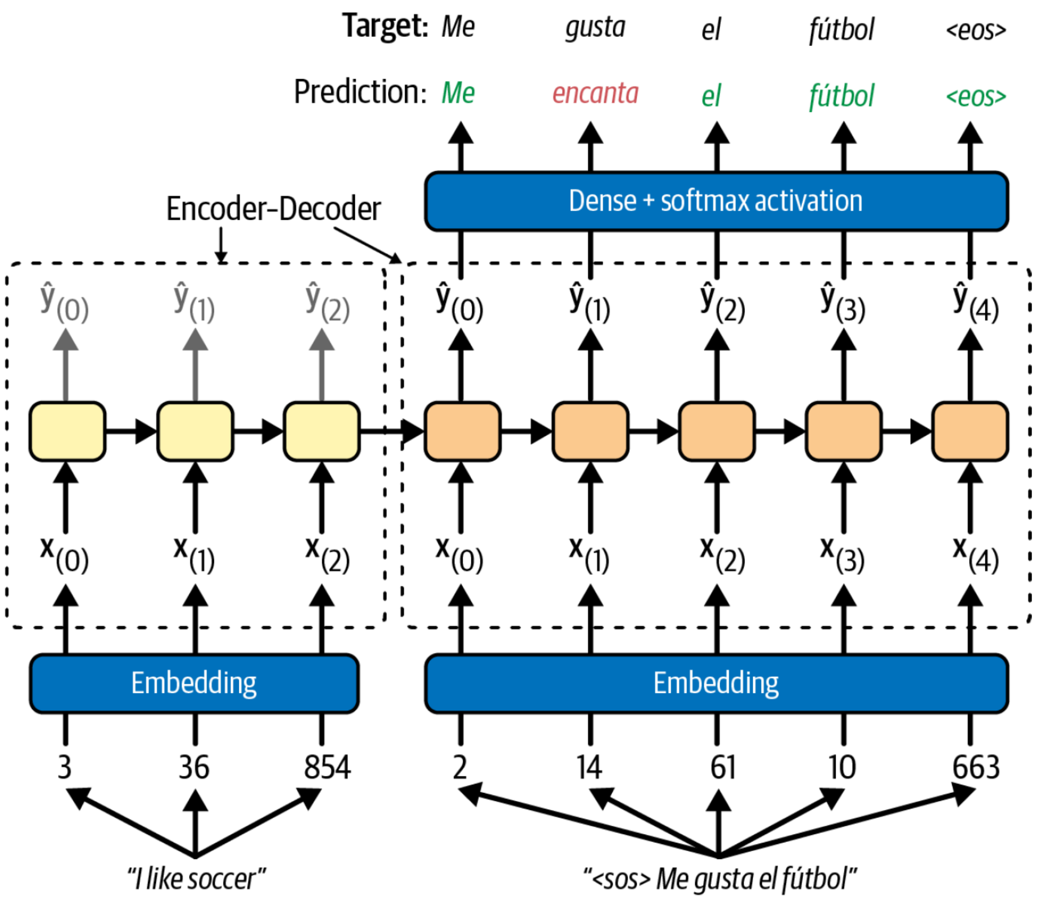
1. Generating Shakespearean Text Using a Character RNN
2. Sentiment Analysis
3. An Encoder–Decoder Network for Neural Machine Translation
4. Attention Mechanisms
5. Transformer Models
6. Summary

4. Attention Mechanisms

- Introduction
- Additive Attention (Bahdanau)
- Multiplicative Attention – Dot (Luong)
- Multiplicative Attention - General (Luong)
- Attention Summary
- Keras Implementation of Dot Product Attention
- Attention Acts as Memory Retrieval Mechanism

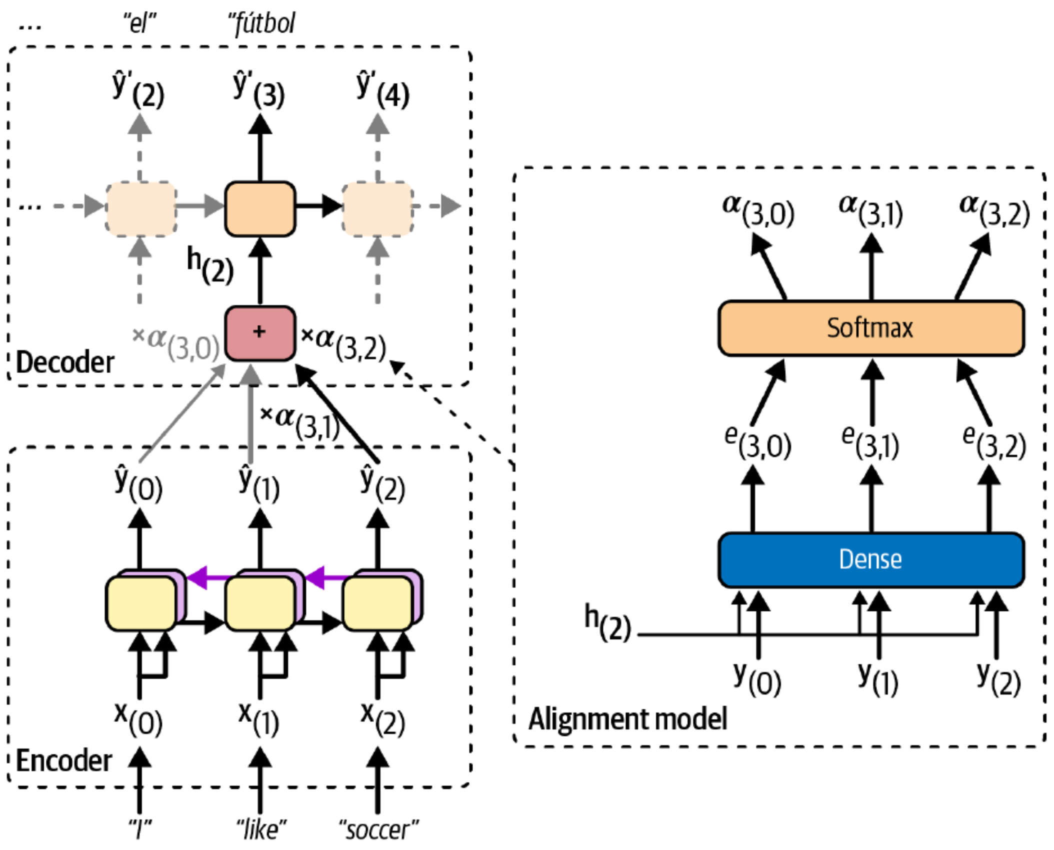
4.1 Attention Mechanisms – Introduction

- The traditional encoder-decoder model has a **limitation**: it encodes the entire input sequence into a single fixed-length vector, which can be challenging for long sequences.
- **Attention mechanisms** address this limitation by allowing the decoder to **focus on specific parts of the input** sequence when generating the output.



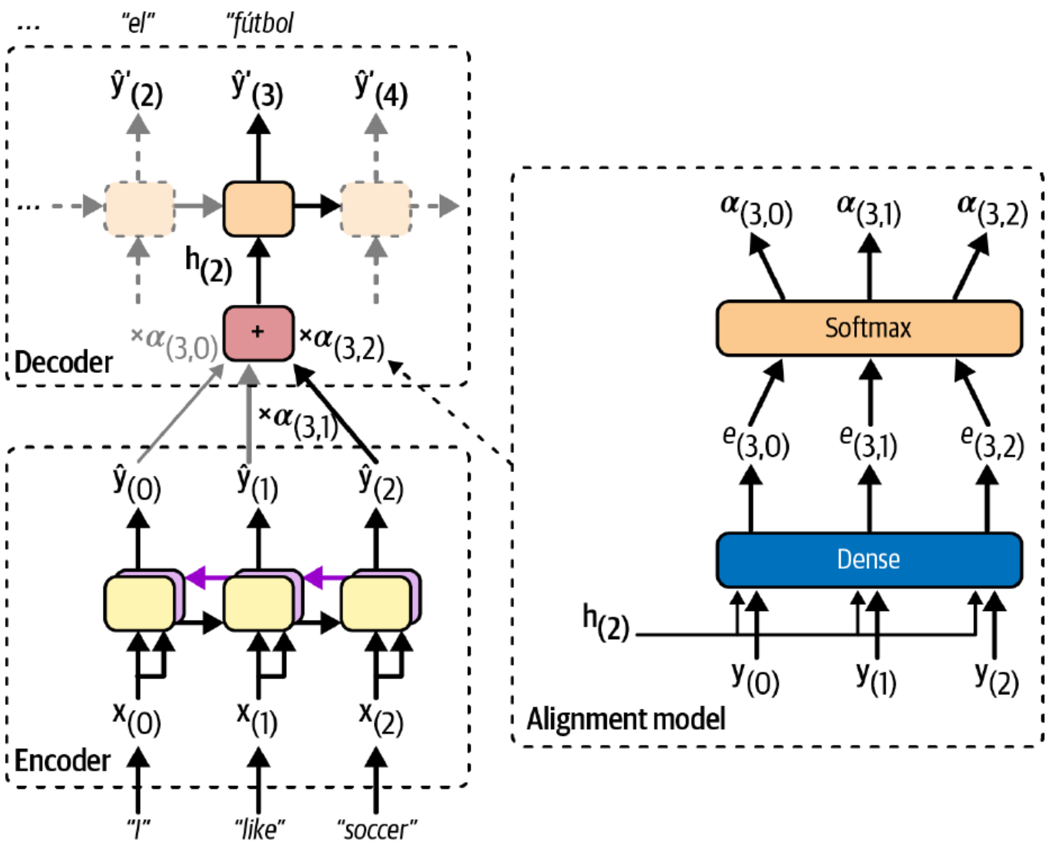
4.2 Additive Attention (Bahdanau)

- Send all the **encoder's outputs** to the decoder.
- The **decoder** computes a weighted sum of all the encoder outputs.
- The **weight** $\alpha_{(t,\, i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step.
- For example, if $\alpha_{(3,2)}$ is larger than $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder pays more attention to the encoder's output for Word 2.



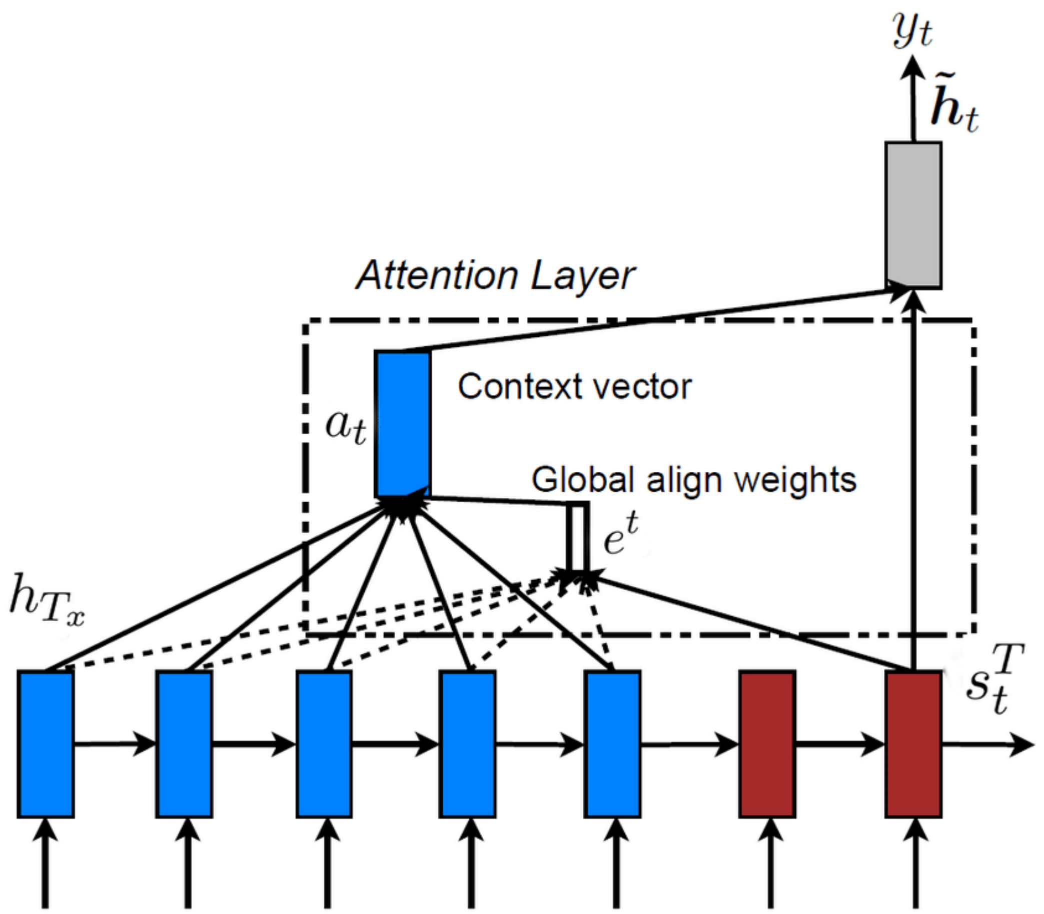
4.2 Additive Attention (Bahdanau)

- The **alignment model** (attention layer) is trained with the rest of the model.
- The **dense layer** outputs a score (or energy) for each encoder output.
- The **softmax** layer makes the weights for a given step add up to 1.
- This is **concatenative** (or **additive**) attention as it concatenates the encoder output with the decoder's previous hidden state.



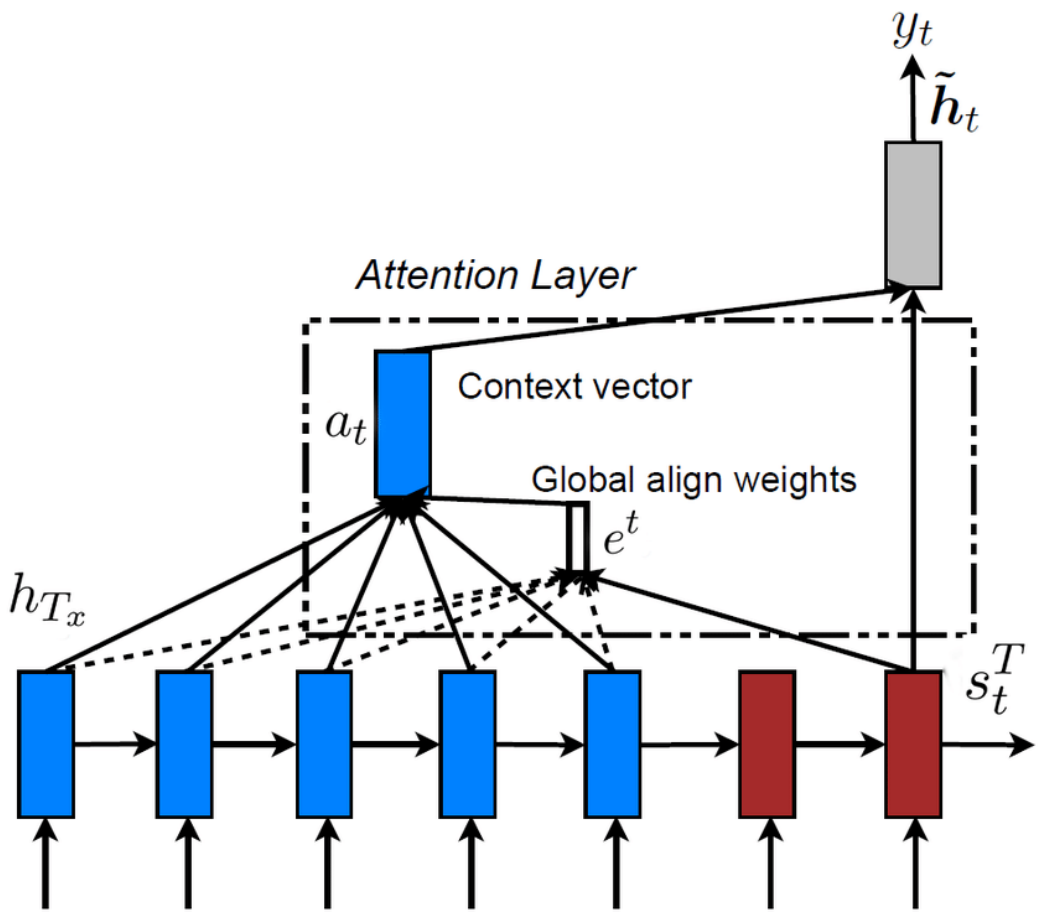
4.3 Multiplicative Attention - Dot (Luong)

- Measures the **similarity** between one of the encoder's outputs and the decoder's hidden state using the **dot product**.
- Uses the decoder's **current hidden state** ($\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$).
- Uses the output of the attention mechanism $\tilde{\mathbf{h}}_{(t)}$ directly to compute the decoder's predictions.



4.4 Multiplicative Attention - General (Luong)

- The encoder outputs first go through a **fully connected layer** (without a bias term) before the dot products are computed.
- Luong *et al*. (2015) compared both dot product approaches with concatenative attention (adding a rescaling parameter vector **v**).
- The **dot product** variants performed **better** than concatenative attention.



4.5 Attention Summary

- The **energy** $e_{(t,\,i)}$ is computed in one of the three mechanisms:
  - Dot product
  - General dot product
  - Concatenative
- **Softmax** is used to get the attention $\alpha_{(t,\,i)}$.
- **Attention** is used to find the decoder's output as the weighted sum of the **encoder's output**.

$$\widetilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,\,i)} \mathbf{y}_{(i)}$$

$$\text{with } \alpha_{(t,\,i)} = \frac{\exp\left(e_{(t,\,i)}\right)}{\sum_{i'} \exp\left(e_{(t,\,i')}\right)}$$

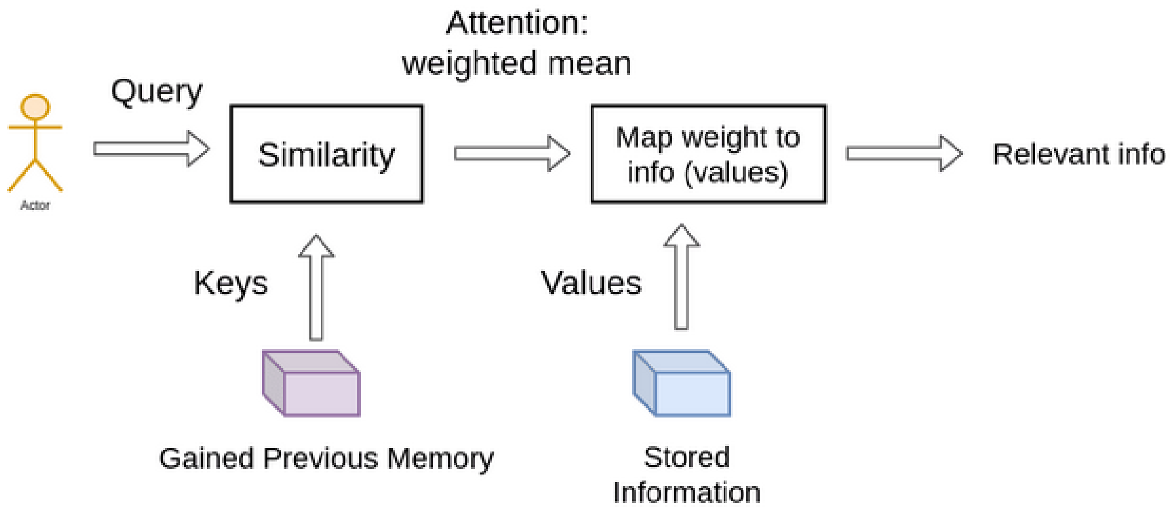
$$\text{and } e_{(t,\,i)} = \begin{cases} \mathbf{h}_{(t)}^{\top} \mathbf{y}_{(i)} & dot \\ \mathbf{h}_{(t)}^{\top} \mathbf{W}\, \mathbf{y}_{(i)} & general \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & concat \end{cases}$$

4.6 Keras Implementation of Dot Product Attention

```
# need to pass all encoder's outputs to the Attention layer
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_sequences=True,
                              return_state=True))

attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs,
                                      encoder_outputs])
output_layer = tf.keras.layers.Dense(vocab_size,
                                      activation="softmax")
Y_proba = output_layer(attention_outputs)
```

4.7 Attention Acts as Memory Retrieval Mechanism

- The Keras `Attention` expects a list as input, containing two or three items: the **queries**, the **keys**, and optionally the **values**.
- If you do not pass any values, then they are automatically equal to the keys.
- The decoder outputs are the queries, and the encoder outputs are both the keys and the values. For each decoder output (query), the attention layer returns a weighted sum of the encoder outputs (keys/values) that are most similar to the decoder output.



Outline

5. Transformer Models

- Attention Is All You Need
- An Avalanche of Transformer Models
- Vision Transformers
- Hugging Face’s Transformers Library

5.1 Attention Is All You Need

- Vaswani *et al.* (2017) created an architecture called the **transformer**, which significantly improved the state-of-the-art in NMT without using any recurrent or convolutional layers, just attention mechanisms.

✓Doesn’t suffer from the vanishing or exploding gradients problems as RNNs

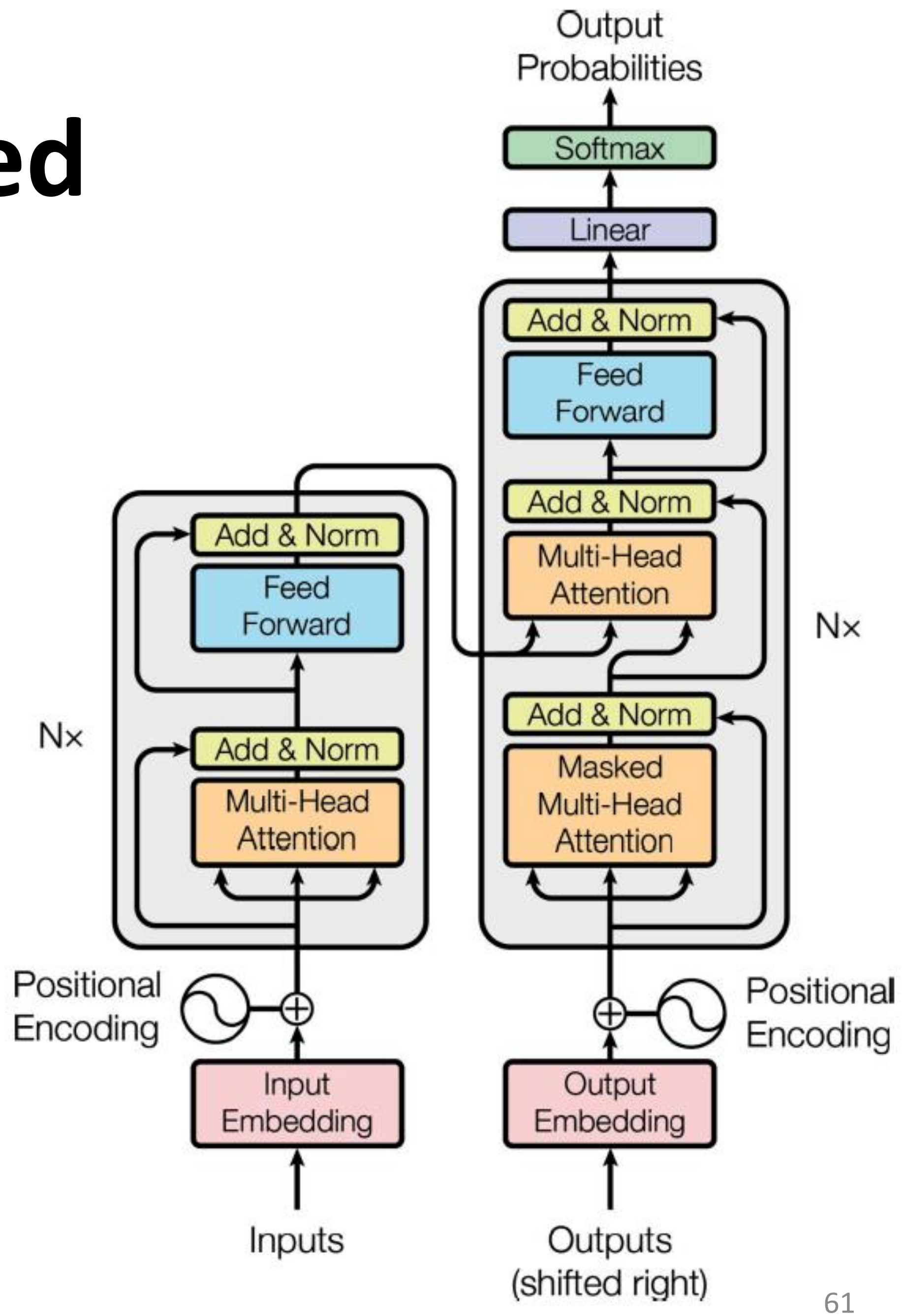
✓It can be trained in fewer steps.

✓It’s easier to parallelize across multiple GPUs.

✓It can better capture long-range patterns.

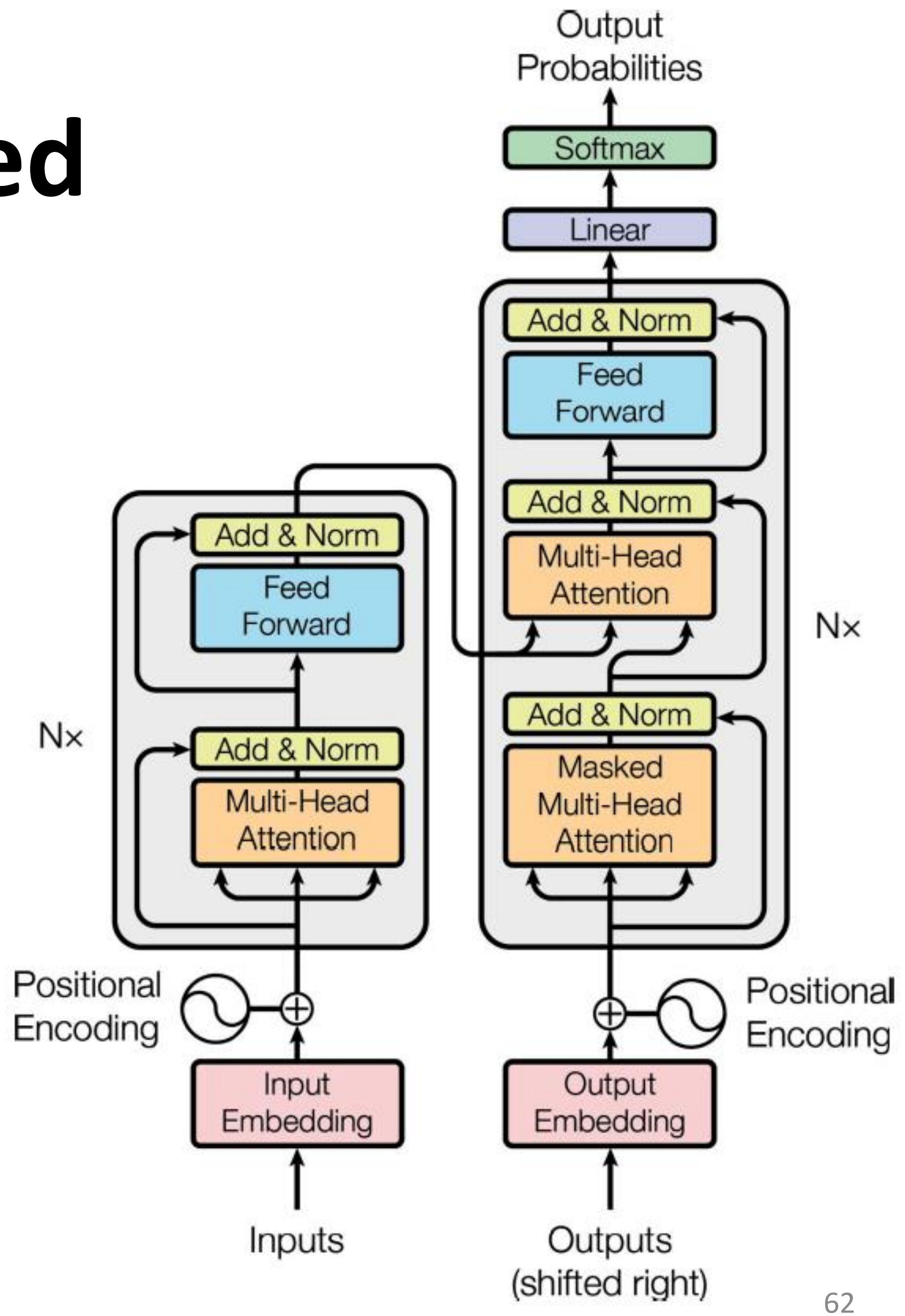
5.1 Attention Is All You Need

- The left part is the **encoder**, and the right part is the **decoder**.
- Each **embedding** layer outputs a 3D tensor of shape [*batch size, sequence length, embedding size*].
- The encoder and the decoder contain modules that are stacked N times. In the paper, $N = 6$.
- The final encoder outputs are fed to the decoder modules.



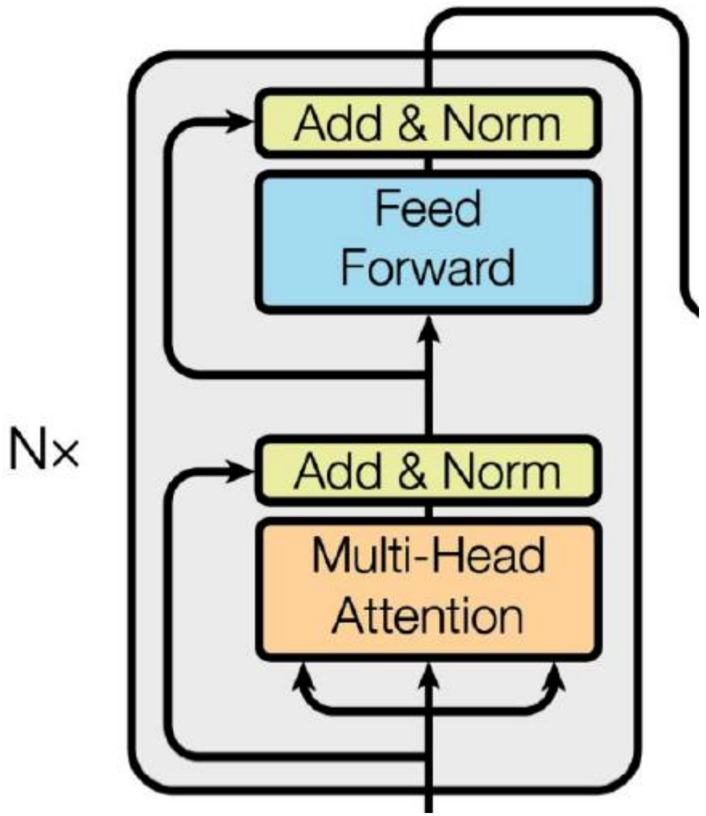
5.1 Attention Is All You Need

- The **encoder's role** is to gradually transform the inputs until each word's representation captures the meaning of the word, in the context of the sentence.
- The **decoder's role** is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation.
- After going through the decoder, each word representation goes through a final **Dense** layer with a **softmax** activation function.



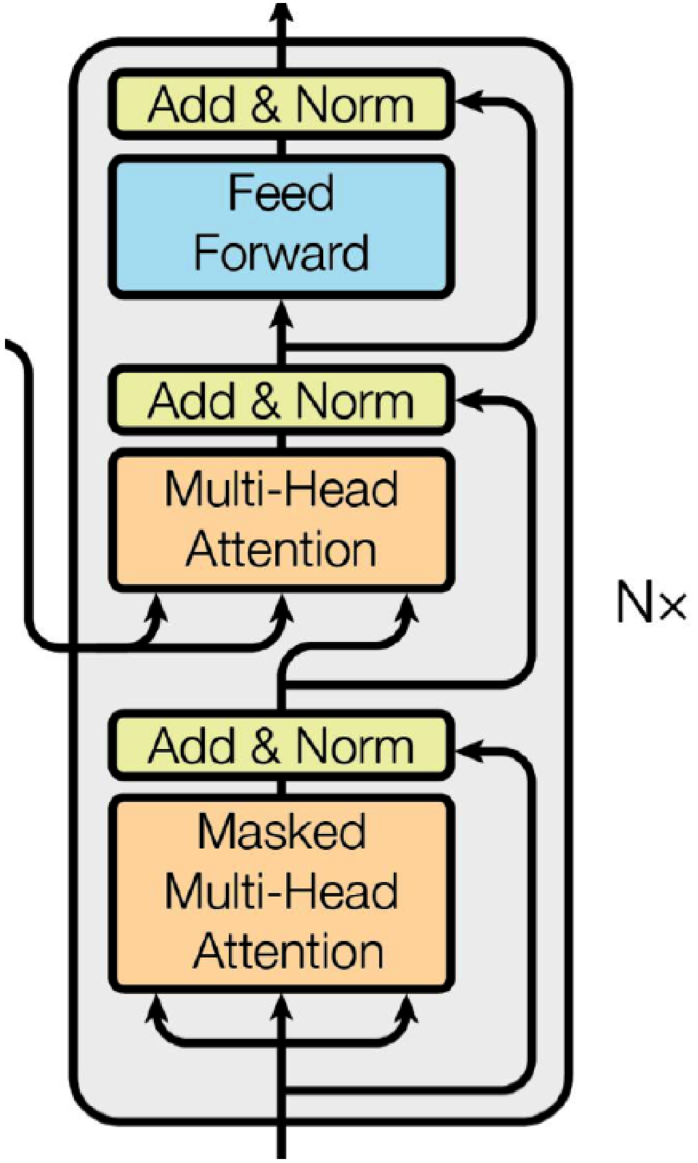
Encoder Modules

- **Skip connections**
- The **multi-head attention** layer updates each word representation by attending to (i.e., paying attention to) all other words in the same sentence.
- **Normalization** layers
- **Feedforward** modules with two dense layers each (the first with ReLU activation, the second with no activation)



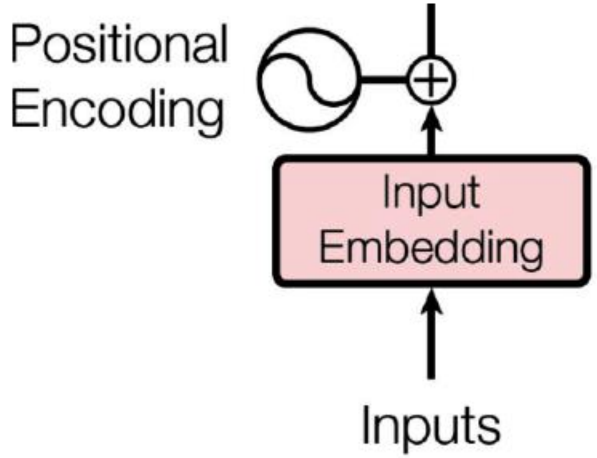
Decoder Modules

- **Skip connections**
- The **masked multi-head attention** layer doesn't attend to words located after it: it's a causal layer.
- **Multi-head attention** layers
- **Normalization** layers
- The upper multi-head attention layer does **cross-attention**, not **self-attention**.
- **Feedforward** modules with two dense layers each (the first with ReLU activation, the second with no activation)



Positional Encoding

- **Dense 3D vectors** that represent the position of each word in the sentence. The n^{th} positional encoding is added to the word embedding of the n^{th} word in each sentence.
- **Same shape** as the output of the embedding layer.
- The authors of the transformer paper used **fixed positional encodings**, based on the **sine** and **cosine** functions at different frequencies
- Each word in the sentence has a **unique positional encoding**.
- The oscillating functions allows the model to learn **relative positions**.



$$P_{p,\,i} = \begin{cases} \sin\left(p/10000^{i/d}\right) & \text{if } i \text{ is even} \\ \cos\left(p/10000^{(i-1)/d}\right) & \text{if } i \text{ is odd} \end{cases}$$

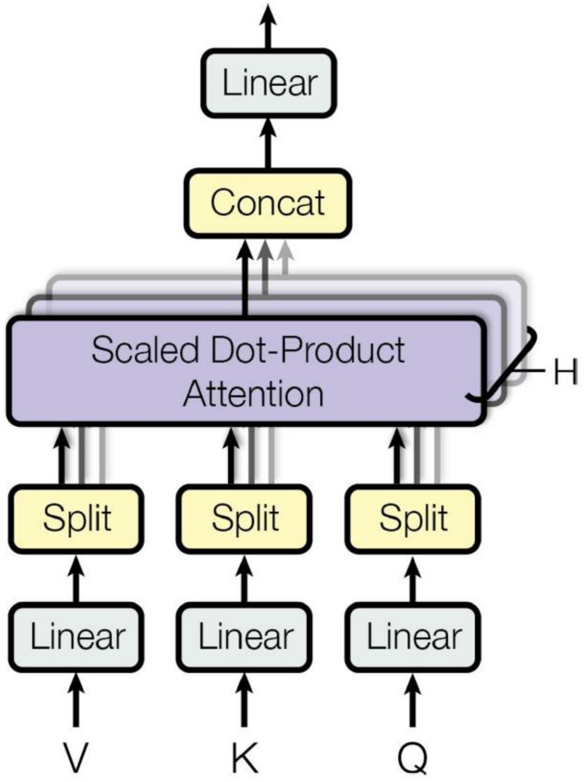
Transformers Video

- YouTube Video: **But what is a GPT? Visual intro to transformers** from 3Blue1Brown

https://youtu.be/wjZofJX0v4M

Multi-head Attention

- Based on the **scaled dot-product attention layer**; queries **Q**, keys **K**, and values **V**.
- **Found efficiently** using matrix multiplications.
- The multi-head attention layer uses **H splits** of the values, keys, and queries: this allows the model to apply multiple projections of the word representation into different subspaces, each focusing on a subset of the word's characteristics.



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

Attention Video

- YouTube Video: **Attention in transformers, visually explained** from 3Blue1Brown

https://youtu.be/eMlx5fFNoYc

5.2 An Avalanche of Transformer Models

- Introduction
- Generative Pre-trained Transformers (GPT)
- Bidirectional Encoder Representations from Transformers (BERT)
- Text-to-Text Transfer Transformer (T5)
- Large Language Model Meta AI (LLaMA)

Introduction

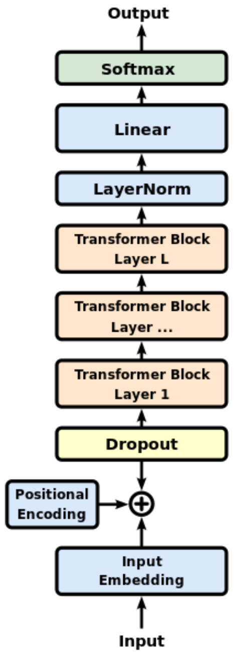
- In 2016, Google Translate gradually replaced the older statistical machine translation approach with the newer neural-networks-based approach that included a **seq2seq model combined by LSTM** and the "**additive**" kind of attention mechanism.
- In 2017, the original (100M-sized) **encoder-decoder transformer** model with a faster (parallelizable or decomposable) attention mechanism was proposed in the "Attention is all you need" paper. The intent of the transformer model is to take a seq2seq model and remove its recurrent neural networks, but preserve its additive attention mechanism.

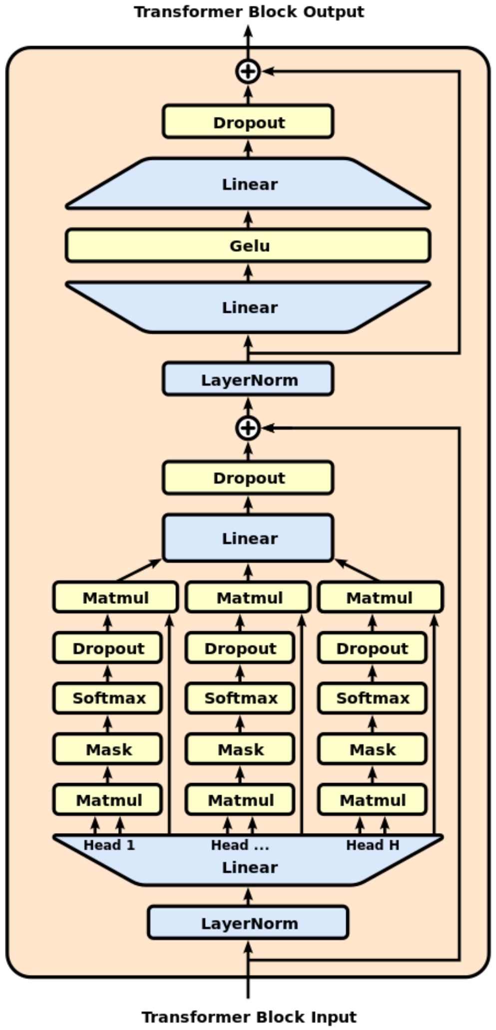
Introduction

- In 2018, an **encoder-only transformer** was used in the (more than 1B-sized) BERT model.
- In 2020, vision transformer and speech-processing convolution-augmented transformer outperformed recurrent neural networks, previously used for vision and speech.
- In 2020, difficulties with converging the original transformer were solved by normalizing layers *before* (instead of after) multiheaded attention by Xiong et al. This is called **pre-LN Transformer**.
- In 2023, **unidirectional** ("autoregressive") transformers were being used in the (more than 100B-sized) GPT-3 and other OpenAI GPT models.

Generative Pre-trained Transformers (GPT)

- In 2018, **OpenAI GPT**, “Improving Language Understanding by Generative Pre-Training.”
- Used **self-supervised pretraining** (predict the next token)
- A transformer of a stack of 12 modules (117 M)
- Then they **fine-tuned** it on various language tasks, using only minor adaptations for each task.
  - **Text classification**
  - **Entailment** (whether sentence A imposes, involves, or implies sentence B as a necessary consequence)
  - **Similarity** (e.g., “Nice weather today” is very similar to “It is sunny”)
  - **Question answering** (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions)





Generative Pre-trained Transformers (GPT)

- In February 2019, **GPT-2** with over 1.5B parameters.
- **Zero-shot learning** (ZSL), achieves good performance on many tasks without any fine-tuning.

- In May 2020, **GPT-3**, 175B parameters, 96 attention layers, each layer contains 96 attention heads.

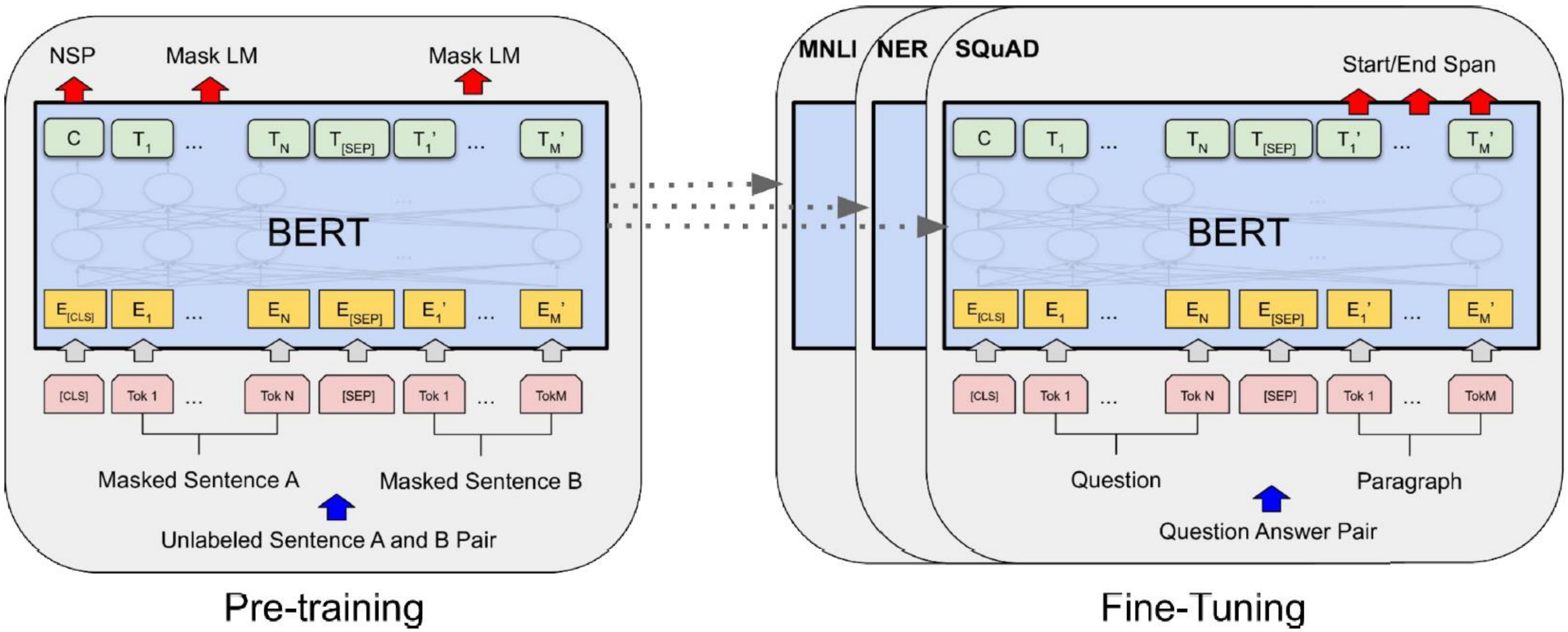
Model	Architecture	Parameter count	Training data	Release date	Training cost
GPT-1	12-level, 12-headed Transformer decoder (no encoder), followed by linear-softmax.	117 M	BookCorpus:[34] 4.5 GB of text, from 7000 unpublished books of various genres.	Jun 11, 2018	30 days on 8 P600 GPUs, or 1 peta FLOP/s-day
GPT-2	GPT-1, but with modified normalization	1.5 M	WebText: 40 GB of text, 8 million documents, from 45 million webpages upvoted on Reddit.	Feb 14, 2019 (initial)	Tens of petaflop/s-day
GPT-3	GPT-2, but with modification to allow larger scaling	175 B	499 billion tokens consisting of CommonCrawl (570 GB), WebText, English Wikipedia, and two book corpora	May 28, 2020	3640 petaflop/s-day
GPT-3.5	Undisclosed	175 B	Undisclosed	Mar 15, 2022	Undisclosed
GPT-4	Also trained with both text prediction and RLHF; accepts both text and images as input	Undisclosed. Estimated 1.7 T	Undisclosed	Mar 14, 2023	Undisclosed. Estimated 2.1e25 FLOP

Bidirectional Encoder Representations from Transformers (BERT)

- In 2018, **Google BERT**, "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding."
- **BERT**$_{\textbf{BASE}}$: 12 encoders with 12 bidirectional self-attention heads totaling 110M parameters
- **BERT**$_{\textbf{LARGE}}$: 24 encoders with 16 bidirectional self-attention heads totaling 340M parameters.
- **Pre-trained** on the Toronto BookCorpus (800M words) and English Wikipedia (2,500M words).

Bidirectional Encoder Representations from Transformers (BERT)

- Used self-supervised pretraining (**masked language model** and **next sentence prediction**)



Bidirectional Encoder Representations from Transformers (BERT)

- The **DistilBERT** model is a small and fast transformer model based on BERT.
- Trained using **distillation**: Transferring knowledge from a **teacher** model to a **student** one, which is usually much smaller than the teacher model.
- This is typically done by using the **teacher's predicted probabilities** for each training instance as targets for the student.
- Distillation often **works better** than training the student from scratch on the same dataset as the teacher!

Text-to-Text Transfer Transformer (T5)

- In 2019, **Google T5**, “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.”
- **Frames all NLP tasks as text-to-text**, using an encoder–decoder transformer.
  - “**Translate English to Spanish**: I like soccer”
  - “**Summarize**:” followed by the paragraph
  - “**Classify**:” followed by the sequence
- **Sizes**
  - **T5-Small**: 60 M, 8 layers, 6 heads
  - **T5-Base**: 220 M, 12 layers, 12 heads
  - **T5-Large**: 770 M, 24 layers, 156 heads
  - **T5-3B**: 3 B, 24 layers, 32 heads
  - **T5-11B**: 11 B, 24 layers, 96 heads

Text-to-Text Transfer Transformer (T5)

- In 2022, **Google ByT5**, "ByT5: Towards a Token-Free Future with Pre-trained Byte-to-Byte Models."
- A variant of the original T5 model, specifically designed to handle raw bytes of text, **no need for subword tokenization** methods like SentencePiece.
- Tokenization Approach: ByT5 processes text at the byte level, **UTF-8** sequences. This allows it to handle an extremely wide range of human languages and other data types (like emojis and special characters) seamlessly.
- ByT5 is trained on a similar mix of tasks as T5, unsupervised and supervised tasks, derived from a dataset called "**Colossal Clean Crawled Corpus**" (**C4**).

Pathways Language Model (PaLM)

- In 2022, **Google PaLM**, "PaLM: Scaling Language Modeling with Pathways."
- Has 540 billion parameters, using over 6,000 TPUs.
- Is a standard transformer, using decoders only.
- This model achieved incredible performance on all sorts of NLP tasks, particularly in natural language understanding (NLU).
- It's capable of impressive feats, such as explaining jokes, giving detailed step-by-step answers to questions, and even coding.
- This is in part due to the model's size, but also thanks to a technique called **Chain of thought prompting**.

Large Language Model Meta AI (LLaMA)

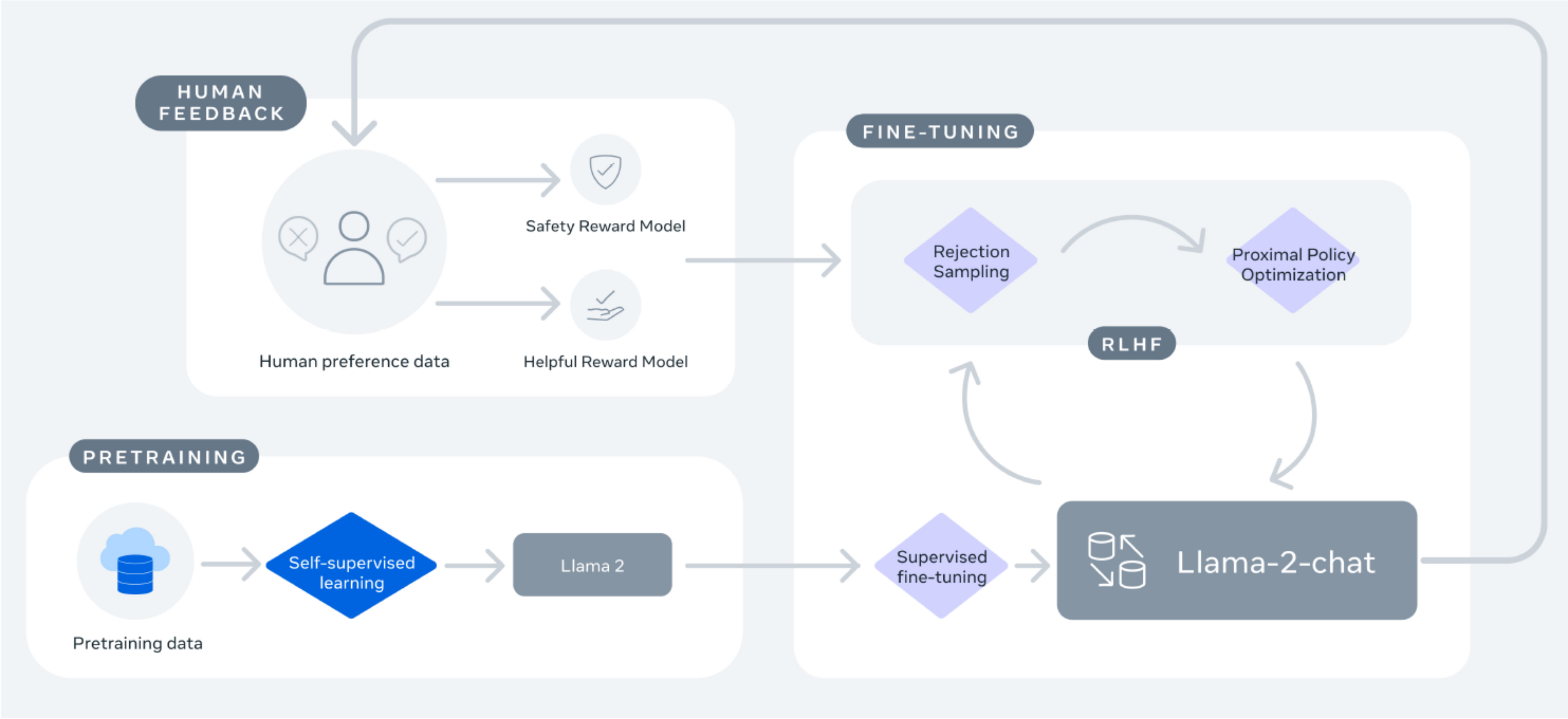
- **LLaMA** is a family of large language models (LLMs) by Meta AI.
- Open Source
- LLaMA 1 released in February 2023.
- LLaMA 2 was release on July 18, 2023.
- LLaMA 3 is expected in May 2024.
- LLaMA uses the transformer architecture, the standard architecture for language modeling since 2018, with some changes.

Model Sizes and Training

	Training Data	Params	Context Length	GQA	Tokens	LR
Llama 1	*See Touvron et al. (2023)*	7B	2k	✗	1.0T	3.0×10^{-4}
		13B	2k	✗	1.0T	3.0×10^{-4}
		33B	2k	✗	1.4T	1.5×10^{-4}
		65B	2k	✗	1.4T	1.5×10^{-4}
Llama 2	*A new mix of publicly available online data*	7B	4k	✗	2.0T	3.0×10^{-4}
		13B	4k	✗	2.0T	3.0×10^{-4}
		34B	4k	✓	2.0T	1.5×10^{-4}
		70B	4k	✓	2.0T	1.5×10^{-4}

Table 1: Llama 2 family of models. Token counts refer to pretraining data only. All models are trained with a global batch-size of 4M tokens. Bigger models — 34B and 70B — use Grouped-Query Attention (GQA) for improved inference scalability.

Training LLaMA 2 Chat - Helpfulness and Safety



Training Time and Cost

		Time (GPU hours)	Power Consumption (W)	Carbon Emitted (tCO_2eq)
LLAMA 2	7B	184320	400	31.22
	13B	368640	400	62.44
	34B	1038336	350	153.90
	70B	1720320	400	291.42
Total		3311616		539.00

Cost Estimate $5M

Performance – Open-source LLMs

Model	Size	Code	Commonsense Reasoning	World Knowledge	Reading Comprehension	Math	MMLU	BBH	AGI Eval
MPT	7B	20.5	57.4	41.0	57.5	4.9	26.8	31.0	23.5
	30B	28.9	64.9	50.0	64.7	9.1	46.9	38.0	33.8
Falcon	7B	5.6	56.1	42.8	36.0	4.6	26.2	28.0	21.2
	40B	15.2	69.2	56.7	65.7	12.6	55.4	37.1	37.0
Llama 1	7B	14.1	60.8	46.2	58.5	6.95	35.1	30.3	23.9
	13B	18.9	66.1	52.6	62.3	10.9	46.9	37.0	33.9
	33B	26.0	70.0	58.4	67.6	21.4	57.8	39.8	41.7
	65B	30.7	70.7	60.5	68.6	30.8	63.4	43.5	47.6
Llama 2	7B	16.8	63.9	48.9	61.3	14.6	45.3	32.6	29.3
	13B	24.5	66.9	55.4	65.8	28.7	54.8	39.4	39.1
	34B	27.8	69.9	58.7	68.0	24.2	62.6	44.1	43.4
	70B	**37.5**	**71.9**	**63.6**	**69.4**	**35.2**	**68.9**	**51.2**	**54.2**

Table 3: Overall performance on grouped academic benchmarks compared to open-source base models.

Performance – Closed-source LLMs

Benchmark (shots)	GPT-3.5	GPT-4	PaLM	PaLM-2-L	Llama 2
MMLU (5-shot)	70.0	**86.4**	69.3	78.3	68.9
TriviaQA (1-shot)	–	–	81.4	**86.1**	85.0
Natural Questions (1-shot)	–	–	29.3	**37.5**	33.0
GSM8K (8-shot)	57.1	**92.0**	56.5	80.7	56.8
HumanEval (0-shot)	48.1	**67.0**	26.2	–	29.9
BIG-Bench Hard (3-shot)	–	–	52.3	**65.7**	51.2

Table 4: Comparison to closed-source models on academic benchmarks. Results for GPT-3.5 and GPT-4 are from OpenAI (2023). Results for the PaLM model are from Chowdhery et al. (2022). Results for the PaLM-2-L are from Anil et al. (2023).

5.3 Vision Transformers

- In 2015, **Visual Attention** used a **convolutional neural network** and a **decoder RNN** with **attention mechanism** to generate captions.
- The decoder uses the attention model to focus on just the right part of the image, *e.g.*, “A woman is throwing a <u>frisbee</u> in a park.

5.3 Vision Transformers

- In 2020, Facebook researchers proposed a **hybrid CNN–transformer** architecture for object detection.
- In Oct 2020, Google researchers introduced a **fully transformer-based** vision model, called **vision transformer** (ViT). Chops the image into little 16 × 16 squares and treats the squares as word representations.
- In Mar 2021, DeepMind researchers introduced the **Perceiver architecture**. It is a **multimodal transformer**, meaning you can feed it text, images, audio, or virtually any other modality.
- In 2021, OpenAI announced **DALL·E**, capable of generating images based on text prompts.

5.4 Hugging Face's Transformers Library

- **Hugging Face** is an AI company that has built a whole ecosystem of easy-to-use, open-source tools for NLP, vision, and beyond.
- Their **Transformers library** allows you to easily download a pretrained model, including its corresponding tokenizer, and then fine-tune it on your own dataset, if needed.
- **Supports** TensorFlow, PyTorch, and JAX.
- The simplest way to use the Transformers library is to use the **`transformers.pipeline()`** function. You just specify which task you want, such as sentiment analysis, and it downloads a default pretrained model, ready to be used.

Hugging Face Pipeline

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
                # many other tasks are available

# Default: distilbert-base-uncased-finetuned-sst-2-english

classifier("The actors were very convincing".)
# Gives a list containing one dictionary per input text:
[{'label': 'POSITIVE', 'score': 0.9998071789741516}]

# Note the bias:
classifier(["I am from India.", "I am from Iraq."])
[{'label': 'POSITIVE', 'score': 0.9896161556243896},
 {'label': 'NEGATIVE', 'score': 0.9811071157455444}]
```

Hugging Face Pipeline

```
# To classify two sentences into:
#   contradiction, neutral, or entailment
model_name = "huggingface/distilbert-base-uncased-
finetuned-mnli"

classifier_mnli = pipeline("text-classification",
                           model=model_name)
classifier_mnli("She loves me. [SEP] She loves me not.")
[{'label': 'contradiction', 'score': 0.9790192246437073}]
```

Manual Usage

```
from transformers import AutoTokenizer,
                         TFAutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForSequenceClassification.from_pretrained(
                                                              model_name)

ids = tokenizer(["I like soccer. [SEP] We all love soccer!",
                 "Joe lived for a very long time. [SEP] Joe is old."],
                padding=True, return_tensors="tf")

outputs = model(ids)
Y_probas = tf.keras.activations.softmax(outputs.logits)
Y_pred = tf.argmax(Y_probas, axis=1)
Y_pred   # 0 = contradiction, 1 = entailment, 2 = neutral
< tf.Tensor: shape = (2,), dtype = int64, numpy = array([2, 1]) >
```

Hugging Face’s Important Links

- Available models: https://huggingface.co/models
- List of tasks: https://huggingface.co/tasks
- Datasets: https://huggingface.co/datasets
- Documentation: https://huggingface.co/docs

Summary

1. Generating Shakespearean Text Using a Character RNN
2. Sentiment Analysis
3. An Encoder–Decoder Network for Neural Machine Translation
4. Attention Mechanisms
5. Transformer Models

Exercises

11. Use the Hugging Face Transformers library to download a pretrained language model capable of generating text (*e.g.*, **GPT**), and try generating more convincing Shakespearean text. You will need to use the model's **`generate()`** method—see Hugging Face's documentation for more details.