Embeddings Recurrent Neural Networks, and Sequences (Part 2)

November 10, 2022

ddebarr@uw.edu

http://cross-entropy.net/ML530/Deep_Learning_5.pdf

Agenda

- Homework Review
- [DLP] Deep Learning for Time Series
- [DLP] Deep Learning for Text

Deep Learning for Time Series

10.1 Different kinds of timeseries tasks 280

10.2 A temperature-forecasting example 281

Preparing the data 285 ▪ A common-sense, non-machine learning baseline 288 ▪ Let's try a basic machine learning model 289 Let's try a 1D convolutional model 290 ▪ A first recurrent baseline 292

10.3 Understanding recurrent neural networks 293

A recurrent layer in Keras 296

10.4 Advanced use of recurrent neural networks 300

Using recurrent dropout to fight overfitting 300 ▪ Stacking recurrent layers 303 ▪ Using bidirectional RNNs 304 Going even further 307

Time Series Applications

- Forecasting: predict future values
- Classification: bot detection
- Event detection: hotword detection (e.g. “Alexa”)
- Anomaly detection: unusual observation
- Change Detection: change of trend

Temperature Forecasting

- wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
  - Weather data
  - Max Planck Institute for Biogeochemistry
  - Jena ("yee nuh"), a city in the Saale ("zah lay") river valley, in the eastern part of Germany
- unzip jena_climate_2009_2016.csv.zip
- Distribution of (inter-arrival gap in seconds, frequency):

  [(600, 420443), (1200, 2), (1800, 1), (8400, 1), (57600, 1), (60600, 1), (63600, 1)]

  https://cross-entropy.net/ML530/missing_data.txt

Inspecting the Jena Weather Dataset

```
import os
fname = os.path.join("jena_climate_2009_2016.csv")

with open(fname) as f:
    data = f.read()

lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
print(header)
print(len(lines))
```

Jena Weather Dataset Columns

1. Date Time: datetime.datetime.strptime(value[0], '%d.%m.%Y %H:%M:%S’) [2009 – 2016 (includes 2 leap years: 2012, 2016)]
2. p (mbar): atmospheric pressure, measured in millibars
3. T (degC): temperature, measured in degrees Celsius
4. Tpot (K): potential temperature (for reference pressure), measured in Kelvin
5. Tdew (degC): dewpoint temperature (saturated with water vapor), measured in degrees Celsius
6. rh (%): relative humidity, measured as water vapor compared to possible water vapor
7. VPmax (mbar): maximum vapor pressure, measured in millibars
8. VPact (mbar): actual vapor pressure, measured in millibars
9. VPdef (mbar): deficit vapor pressure, measured in millibars
10. sh (g/kg): specific humidity, measured as grams of water vapor per kilogram of air
11. H2OC (mmol/mol): dihydrogen oxide (water vapor) concentration, measured as millimoles of water vapor to moles of air
12. rho (g/m**3): water density, measured as mass in grams divided by volume in cubic meters
13. wv (m/s): wind velocity, measured in meters per second
14. max. wv (m/s): maximum wind velocity, measured in meters per second
15. wd (deg): wind direction, measured in degrees

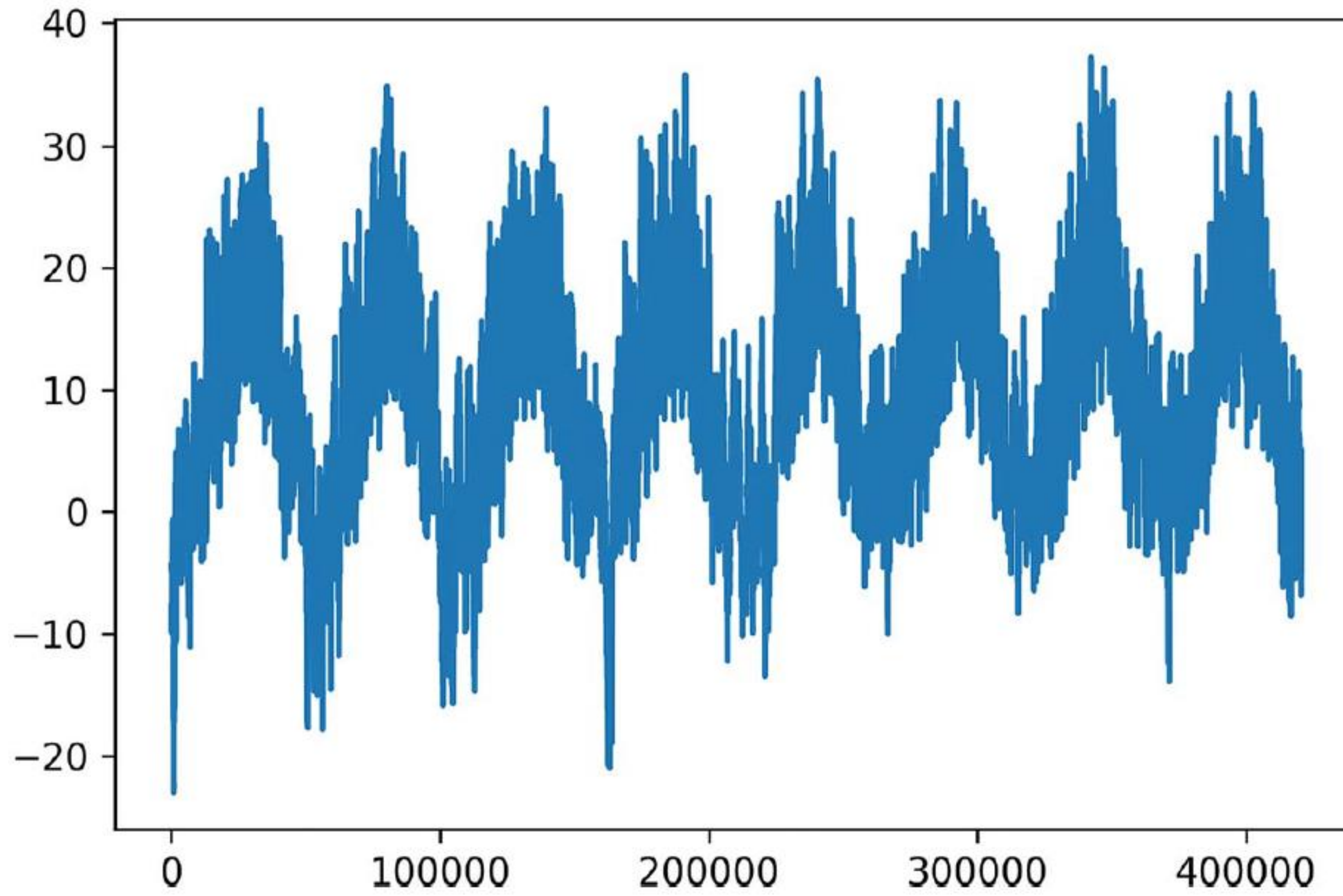
Parsing the Data

```
import numpy as np
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i, :] = values[:]
```

We store column 1 in the "temperature" array.

We store all columns (including the temperature) in the "raw_data" array.

Plotting the Temperature Timeseries (8 years)



Plotting the Temperature for the First 10 Days

0
−5
−10
−15
−20

0
200
400
600
800
1000
1200
1400

Always Look for Periodicity in Your Data

- Periodicity over multiple timescales is an important and very common property of timeseries data
- Whether you’re looking at the weather, mall parking occupancy, traffic to a website, sales of a grocery store, or steps logged in a fitness tracker, you’ll see daily cycles and yearly cycles (human-generated data also tends to feature weekly cycles)

Number of Samples for Each Data Split

```
>>> num_train_samples = int(0.5 * len(raw_data))
>>> num_val_samples = int(0.25 * len(raw_data))
>>> num_test_samples = len(raw_data) - num_train_samples - num_val_samples
>>> print("num_train_samples:", num_train_samples)
>>> print("num_val_samples:", num_val_samples)
>>> print("num_test_samples:", num_test_samples)
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

Normalizing the Data

```
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /= std
```

timeseries_dataset_from_array()

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)

for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

The sequences will be batched in batches of size 2.

Creating a Dataset

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples)
```

Only start_index and end_index change for val_dataset and test_dataset

Inspecting Shapes for the Dataset

```
>>> for samples, targets in train_dataset:
>>>     print("samples shape:", samples.shape)
>>>     print("targets shape:", targets.shape)
>>>     break
samples shape: (256, 120, 14)
targets shape: (256,)
```

Temperature from 24 Hours Ago as Baseline

```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"Validation MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"Test MAE: {evaluate_naive_method(test_dataset):.2f}")
```

The temperature feature is at column 1, so samples[:, -1, 1] is the last temperature measurement in the input sequence. Recall that we normalized our features, so to retrieve a temperature in degrees Celsius, we need to un-normalize it by multiplying it by the standard deviation and adding back the mean.

Val MAE = 2.44; Tst MAE = 2.62

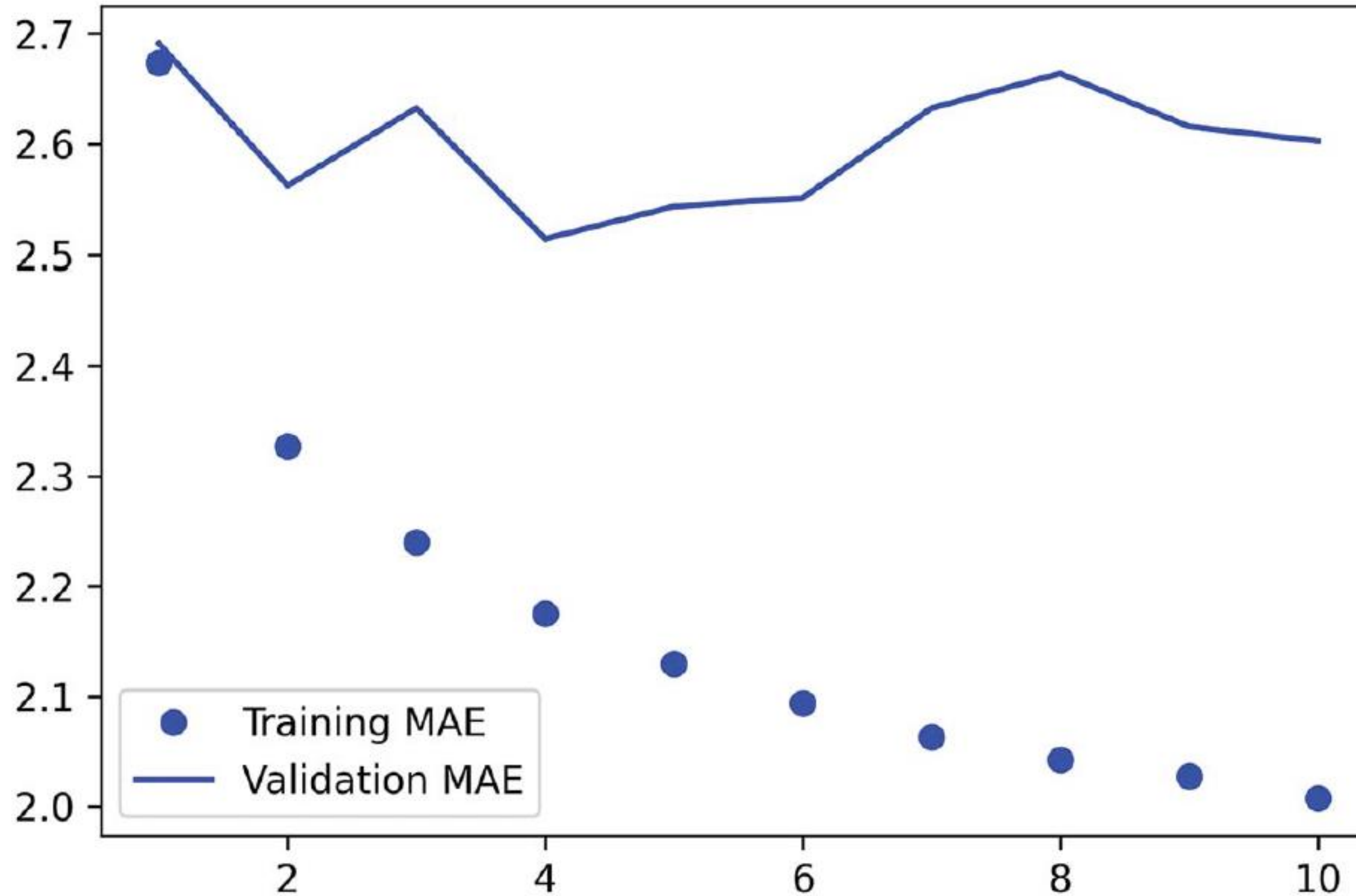
Multi-Layer Perceptron (MLP)

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

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.

MLP Result

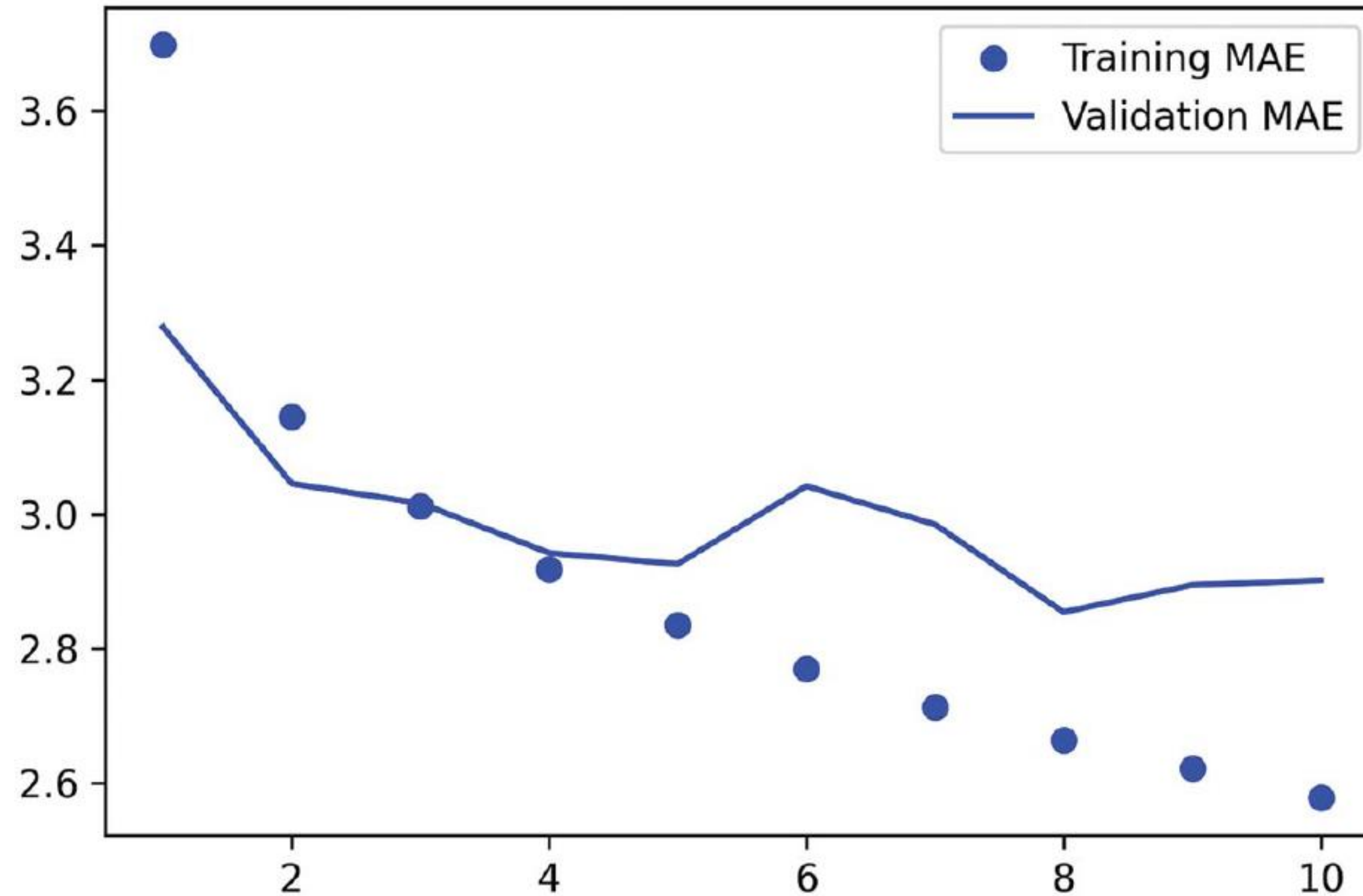


MLP does not beat “use last temperature” baseline, which had Val MAE = 2.44

ConvNet

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

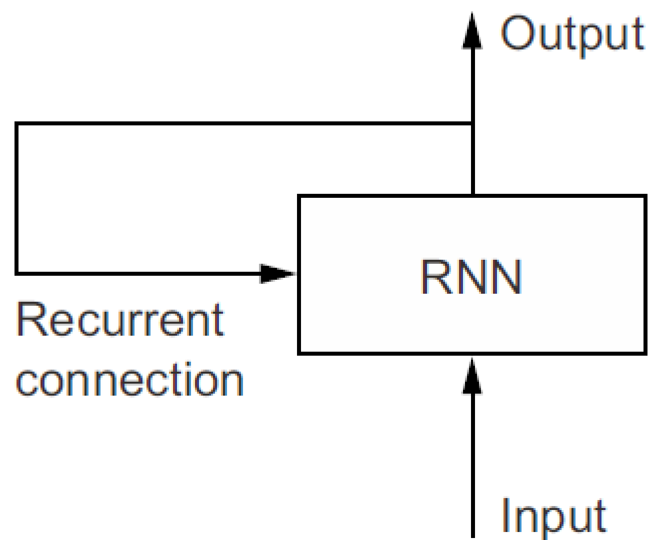
ConvNet Result



Does *not* improve upon either the “use last” basline or the MLP

Pseudocode Recurrent Neural Network (RNN)

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```



NumPy Implementation for an RNN Cell

```
import numpy as np
timesteps = 100
input_features = 32
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.stack(successive_outputs, axis=0)
```

Input data: random noise for the sake of the example

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Initial state: an all-zero vector

Creates random weight matrices

Stores this output in a list

Combines the input with the current state (the previous output) to obtain the current output. We use tanh to add non-linearity (we could use any other activation function).

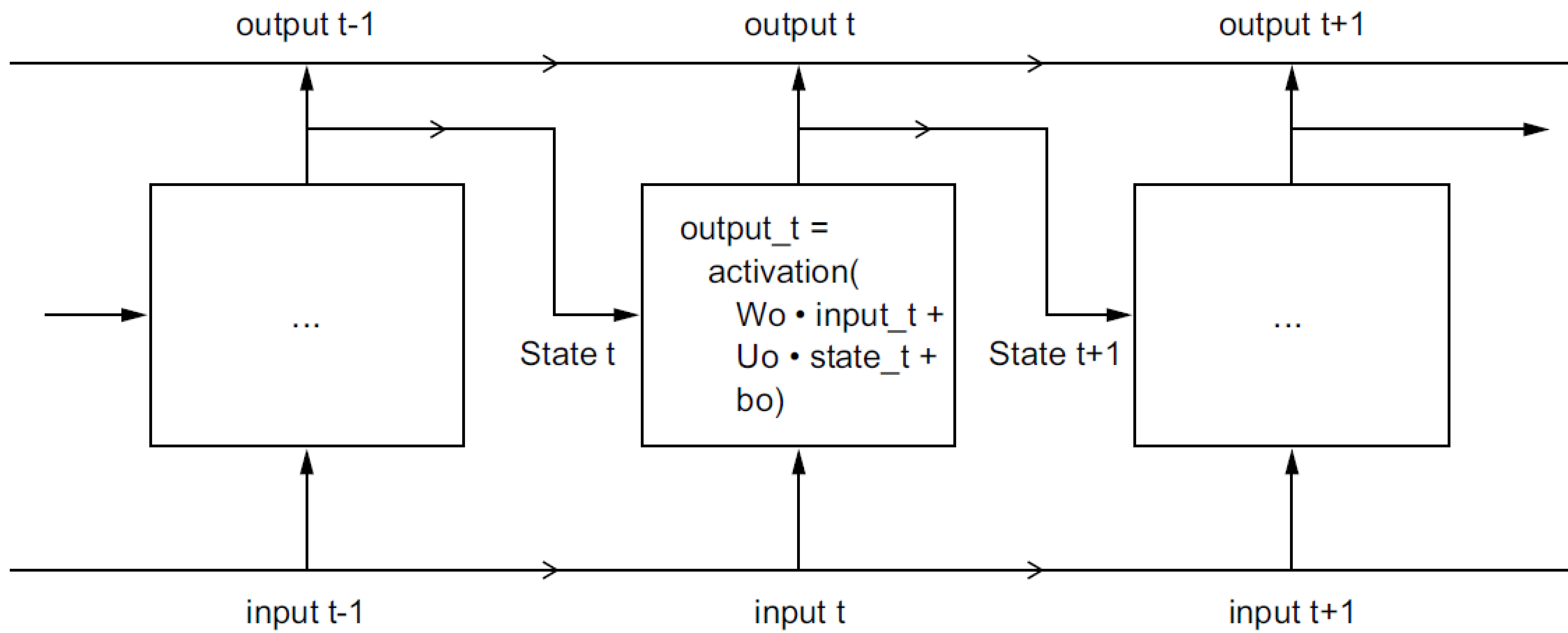
The final output is a rank-2 tensor of shape (timesteps, output_features).

input_t is a vector of shape (input_features,).

Updates the state of the network for the next timestep

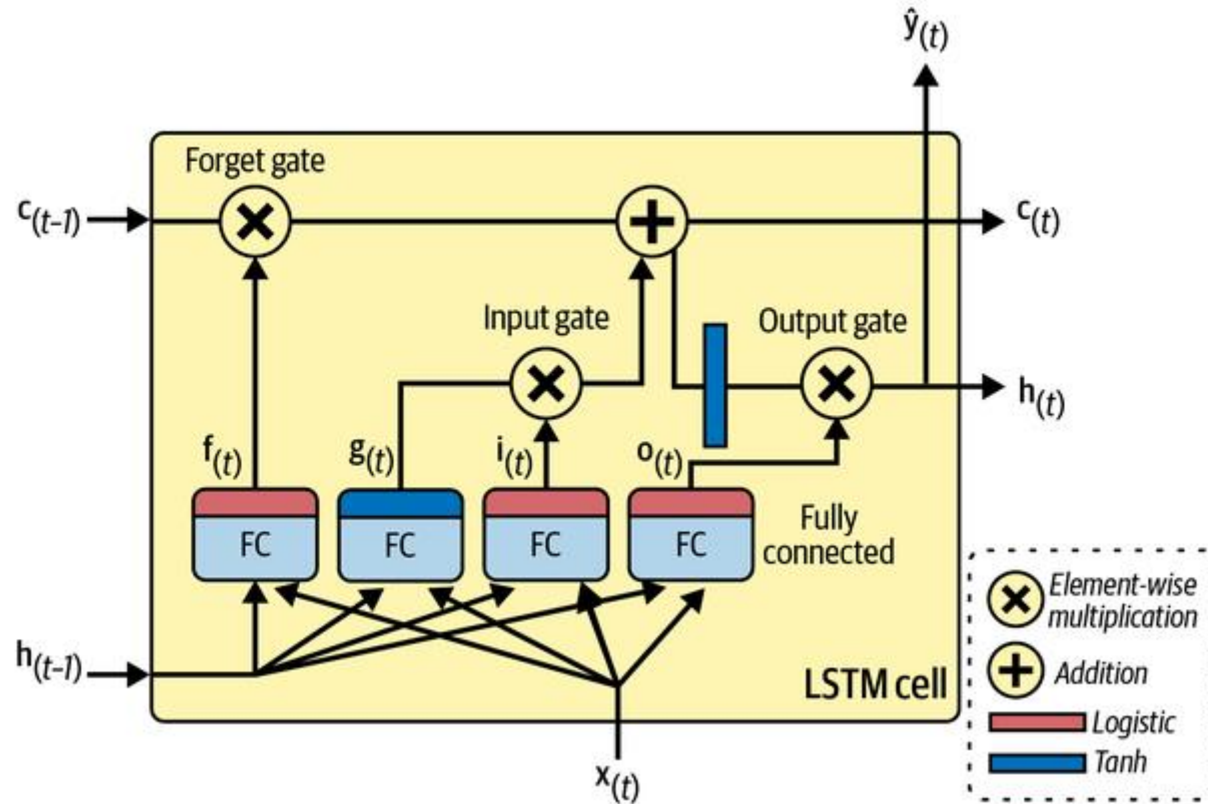
Simple RNN Cell, Unrolled Over Time

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



https://learning.oreilly.com/library/view/hands-on-machine-learning/9781098125967/

Long Short-Term Memory (LSTM) Cell

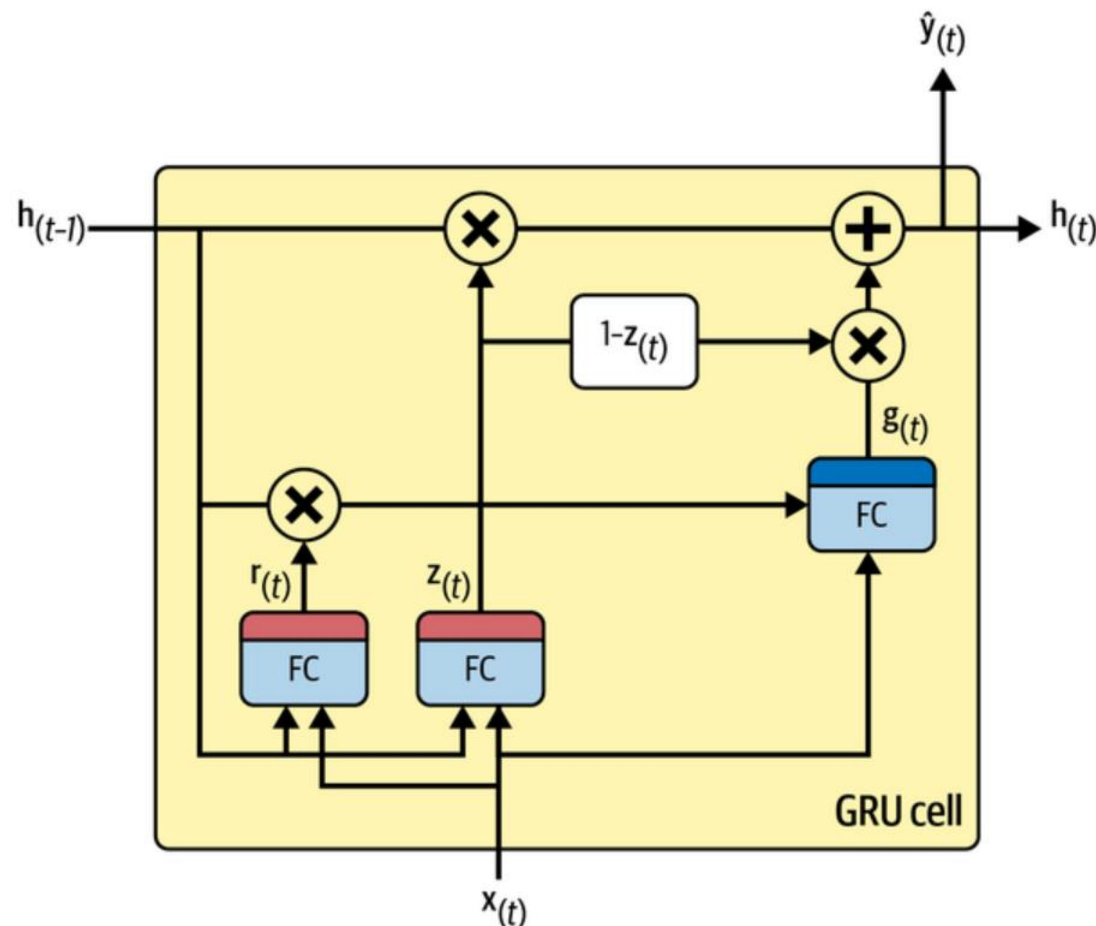


$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

An LSTM cell adds features to its memory.
3 gates; values in [0, 1] …
i: input
f: forget
o: output

https://learning.oreilly.com/library/view/hands-on-machine-learning/9781098125967/

Gated Recurrent Unit (GRU) Cell

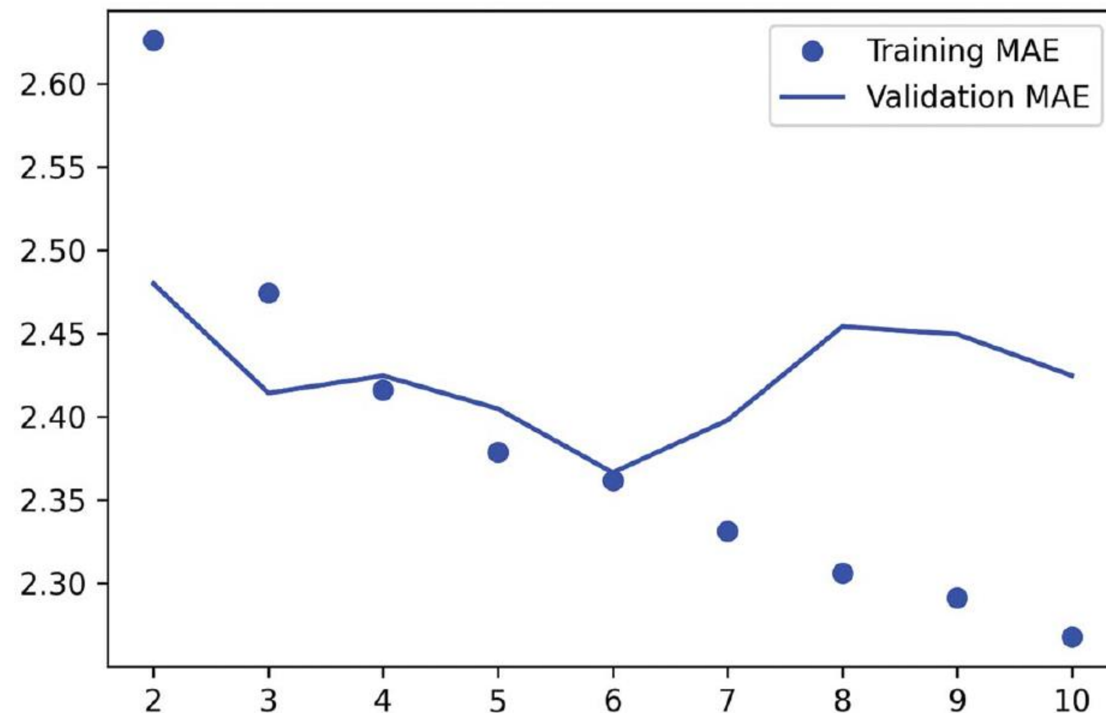


$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\top}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}
\end{aligned}
$$

A GRU cell adds features to its memory.
2 gates; values in [0, 1] …
r: reset
z: update

LSTM Model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```



Beats all previous predictors: Test MAE = 2.55

RNN Layer Can Process Any Sequence Length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

RNN Layer That Returns Only Its Last Output

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

Note that return_sequences=False is the default.

RNN Layer That Returns All Outputs

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(120, 16)
```

Stacking RNN Layers

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

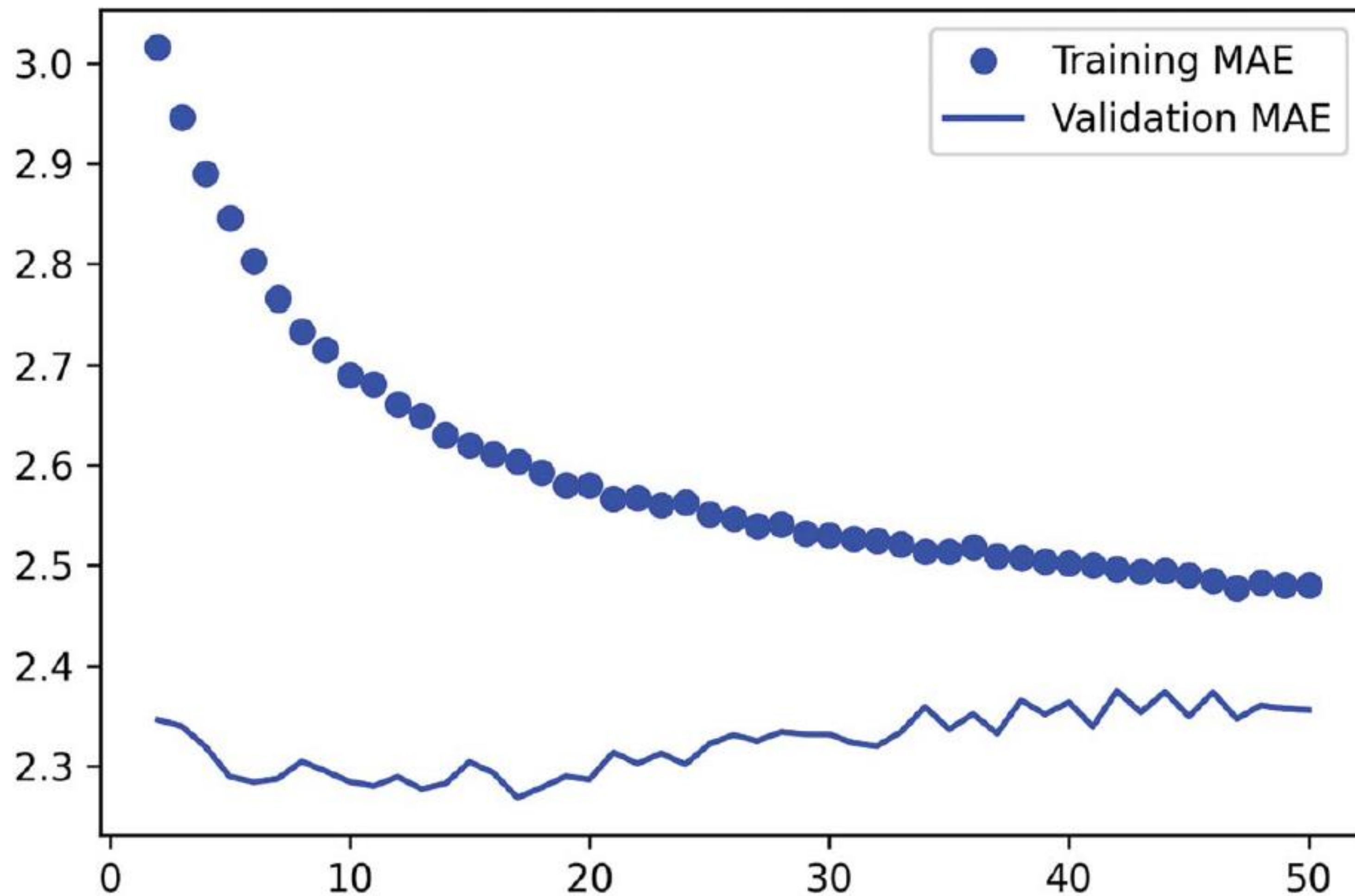
Recurrent Dropout

- same dropout mask used for every position
- dropout argument: the dropout rate for inputs from previous layer [same effect as SpatialDropout1D]
- recurrent_dropout: the dropout rate for inputs from previous position

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

To regularize the Dense layer, we also add a Dropout layer after the LSTM.

Recurrent Dropout Result



Woo! Val MAE = 2.27; Tst MAE = 2.45

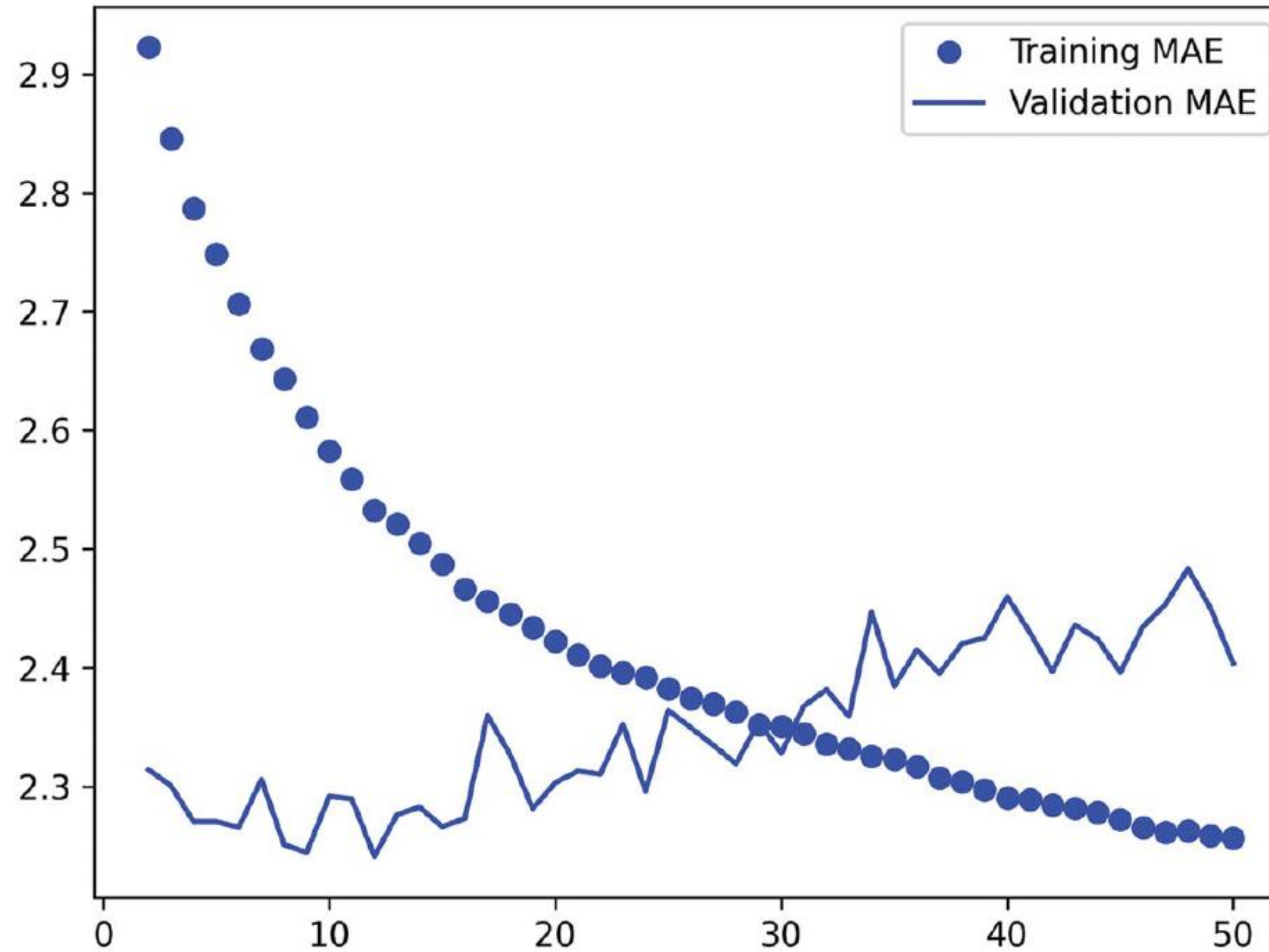
Restrictions for cuDNN Implementation

- Recurrent dropout isn’t supported by the LSTM and GRU cuDNN kernels, so adding it to your layers forces the runtime to fall back to the regular TensorFlow implementation, which is generally two to five times slower on GPU (even though its computational cost is the same)
- See “requirements to use the cuDNN implementation”: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM
- Unrolling can be used to speed up the RNN, but it may also consume more memory

Stacked GRU Cells with recurrent_dropout

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

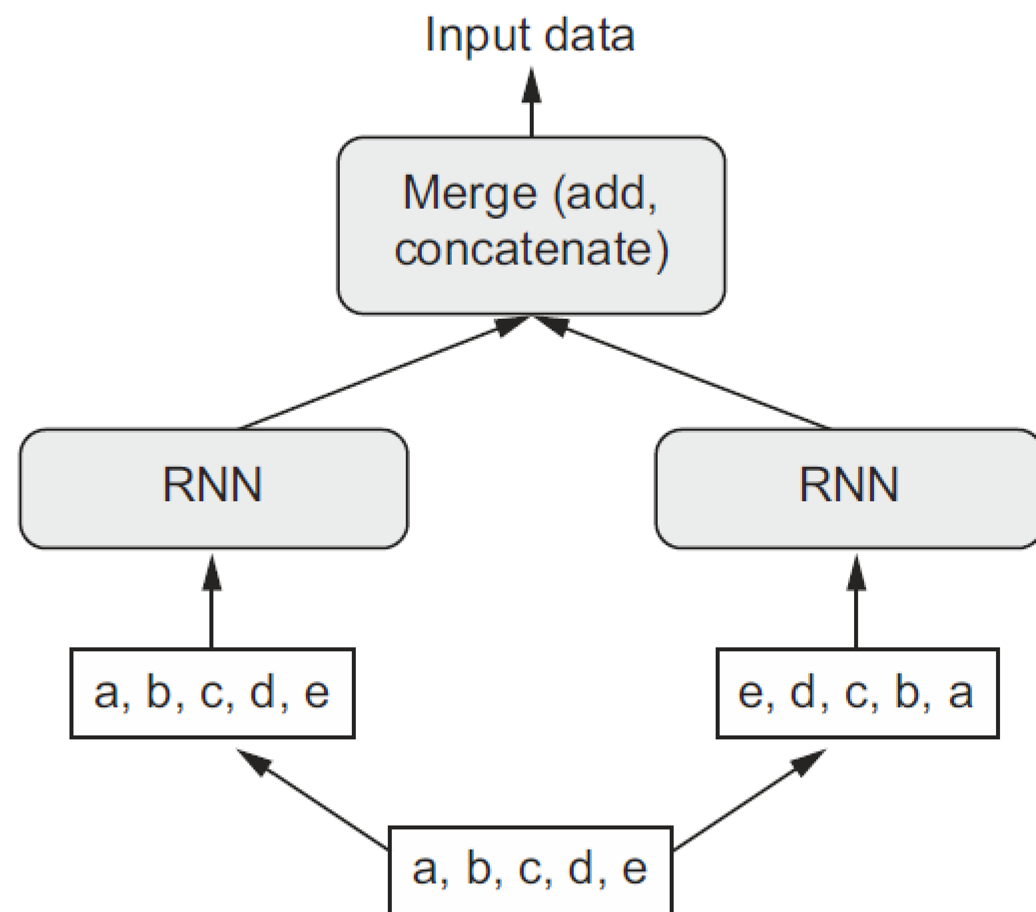
Stacked GRU Cells Results



Our best score: Tst MAE = 2.39

Bidirectional RNN

- There are 2 distinct cells: one for the forward direction, the other for the reverse direction
- The two cell outputs are concatenated

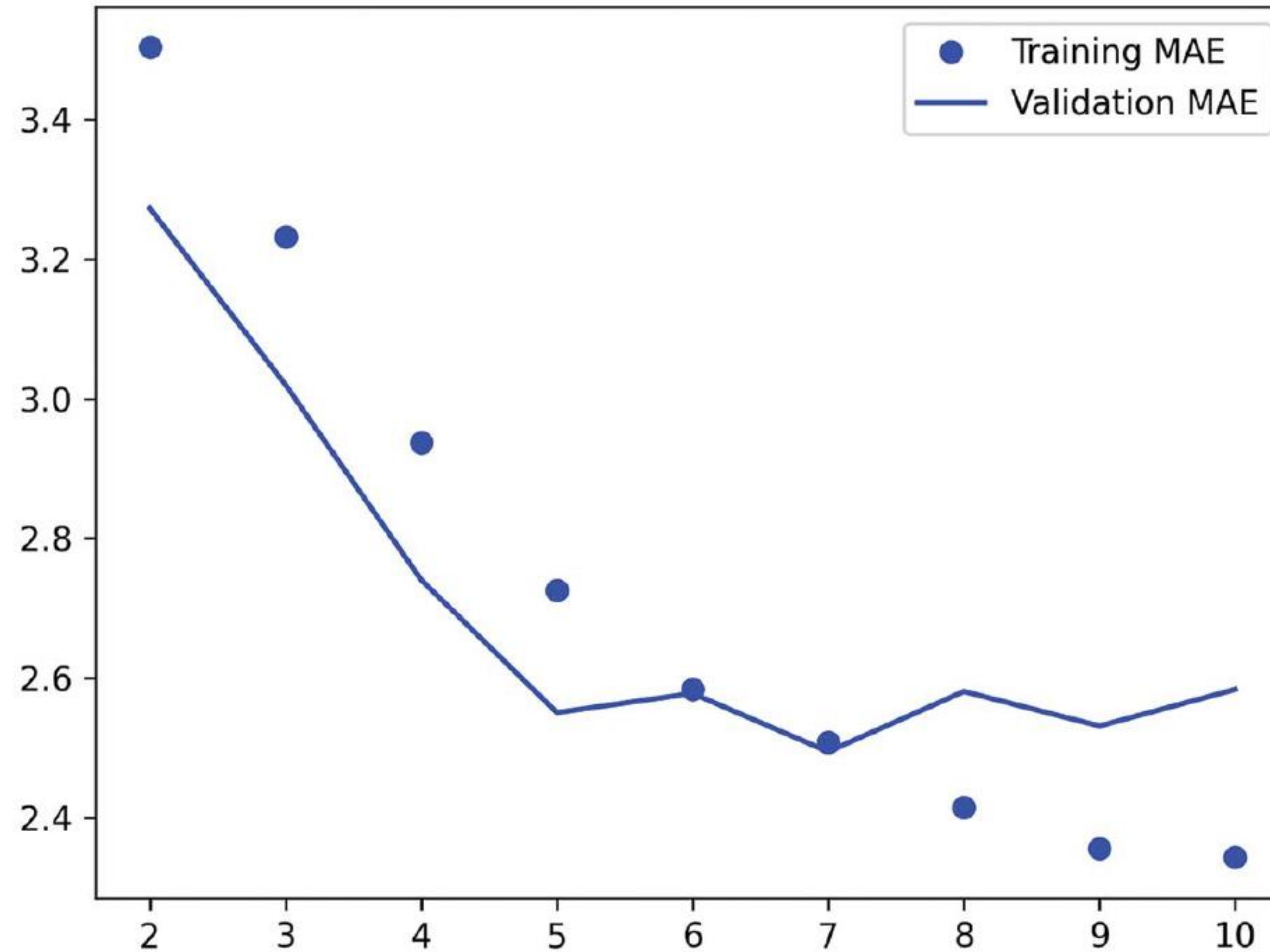


Bidirectional RNN Model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

… does not perform as well as the plain LSTM layer

LSTM with samples[:, ::-1, :] (reversed inputs)



… did worse than the “use last” baseline

Going Even Further

- Adjust the number of units in each recurrent layer in the stacked setup, as well as the amount of dropout. The current choices are largely arbitrary and thus probably suboptimal.
- Adjust the learning rate used by the RMSprop optimizer, or try a different optimizer.
- Try using a stack of Dense layers as the regressor on top of the recurrent layer, instead of a single Dense layer.
- Improve the input to the model: try using longer or shorter sequences or a different sampling rate, or start doing feature engineering.

Markets and Machine Learning

- Always remember that all trading is fundamentally information arbitrage: gaining an advantage by leveraging data or insights that other market participants are missing
- Trying to use well-known machine learning techniques and publicly available data to beat the markets is effectively a dead end, since you won’t have any information advantage compared to everyone else
- You’re likely to waste your time and resources with nothing to show for it

Summary

- As you first learned in chapter 5, when approaching a new problem, it’s good to first establish common-sense baselines for your metric of choice
- Try simple models before expensive ones, to make sure the additional expense is justified
- When you have data where ordering matters, and in particular for timeseries data, recurrent networks are a great fit and easily outperform models that first flatten the temporal data
- To use dropout with recurrent networks, you should use a time-constant dropout mask and recurrent dropout mask
- Stacked RNNs provide more representational power than a single RNN layer

Deep Learning for Text

11.1 Natural language processing: The bird's eye view 309

11.2 Preparing text data 311

Text standardization 312 ▪ Text splitting (tokenization) 313 Vocabulary indexing 314 ▪ Using the TextVectorization layer 316

11.3 Two approaches for representing groups of words: Sets and sequences 319

Preparing the IMDB movie reviews data 320 ▪ Processing words as a set: The bag-of-words approach 322 ▪ Processing words as a sequence: The sequence model approach 327

Deep Learning for Text

11.4 The Transformer architecture 336

Understanding self-attention 337 ▪ Multi-head attention 341 The Transformer encoder 342 ▪ When to use sequence models over bag-of-words models 349

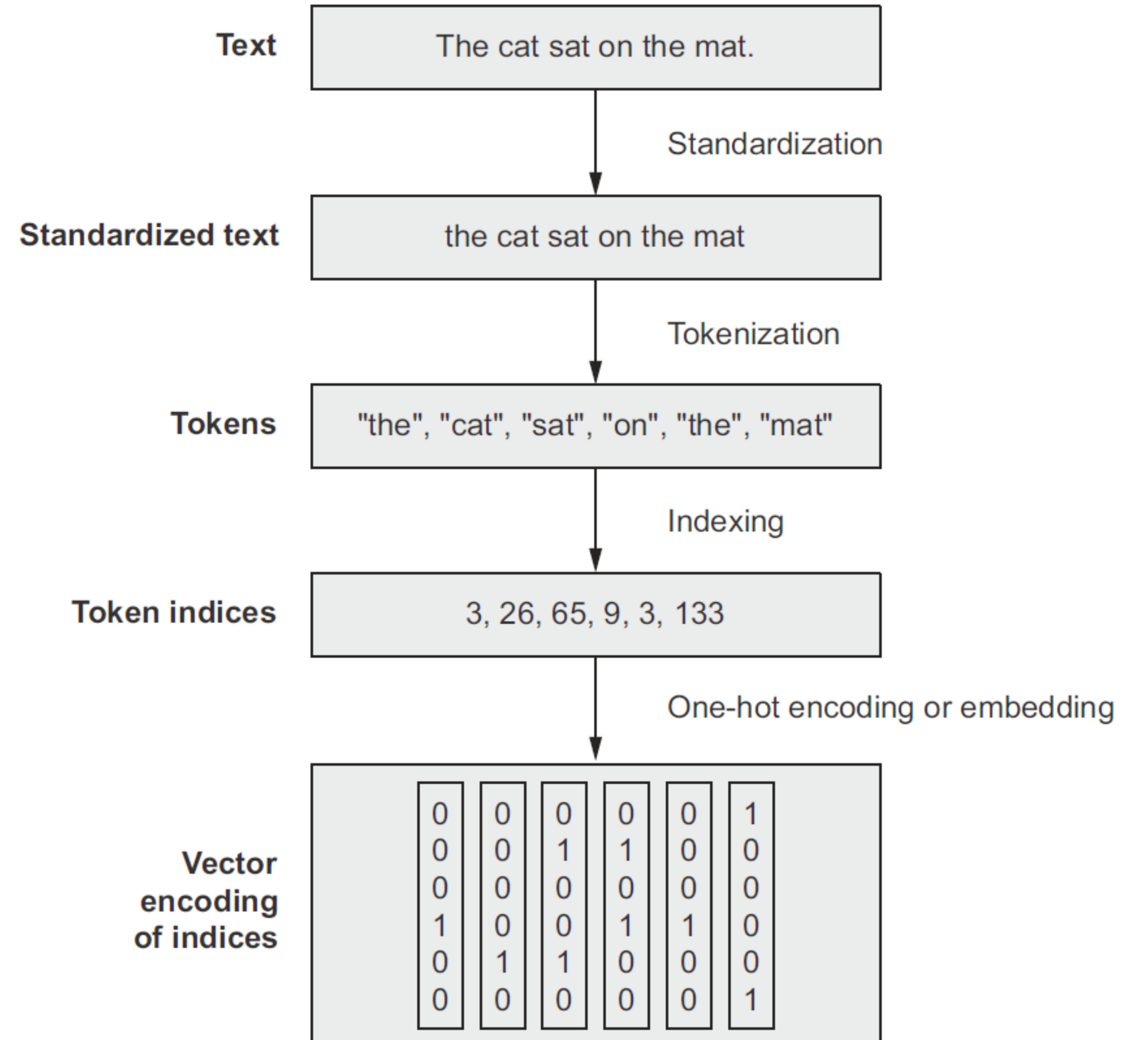
11.5 Beyond text classification: Sequence-to-sequence learning 350

A machine translation example 351 ▪ Sequence-to-sequence learning with RNNs 354 ▪ Sequence-to-sequence learning with Transformer 358

Example Applications

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

From Raw Text to Vectors



Tokenization

- *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.

Understanding n-grams and bag-of-words

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

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

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

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

Indexing

```
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
```

Vectorizer

```python
class Vectorizer:
    def standardize(self, text):
        text = text.lower()
        return "".join(char for char in text
                       if char not in string.punctuation)

    def tokenize(self, text):
        text = self.standardize(text)
        return text.split()

    def make_vocabulary(self, dataset):
        self.vocabulary = {"": 0, "[UNK]": 1}
        for text in dataset:
            text = self.standardize(text)
            tokens = self.tokenize(text)
            for token in tokens:
                if token not in self.vocabulary:
                    self.vocabulary[token] = len(self.vocabulary)
        self.inverse_vocabulary = dict(
            (v, k) for k, v in self.vocabulary.items())

    def encode(self, text):
        text = self.standardize(text)
        tokens = self.tokenize(text)
        return [self.vocabulary.get(token, 1) for token in tokens]

    def decode(self, int_sequence):
        return " ".join(
            self.inverse_vocabulary.get(i, "[UNK]") for i in int_sequence)
```

```python
vectorizer = Vectorizer()
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
vectorizer.make_vocabulary(dataset)
```

Haiku by poet Hokushi

```python
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = vectorizer.encode(test_sentence)
>>> print(encoded_sentence)
[2, 3, 5, 7, 1, 5, 6]
>>> decoded_sentence = vectorizer.decode(encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

TextVectorization

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

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

TextVectorization Declaration

```
import re
import string
import tensorflow as tf

def custom_standardization_fn(string_tensor):
    lowercase_string = tf.strings.lower(string_tensor)
    return tf.strings.regex_replace(
        lowercase_string, f"[{re.escape(string.punctuation)}]", "")

def custom_split_fn(string_tensor):
    return tf.strings.split(string_tensor)

text_vectorization = TextVectorization(
    output_mode="int",
    standardize=custom_standardization_fn,
    split=custom_split_fn,
)
```

Convert strings to lowercase.

Replace punctuation characters with the empty string.

Split strings on whitespace.

Text Vectorization Demo

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

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

```
>>> 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"
```

In a tf.data pipeline or as part of a model

```
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.

```
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.

Preparing the IMDB Movie Reviews Data

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

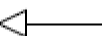
```
aclImdb/
...train/
......pos/
......neg/
...test/
......pos/
......neg/
```

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

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

Creating a Validation Partition

```
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)
```



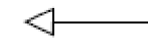
Shuffle the list of training files using a seed, to ensure we get the same validation set every time we run the code.

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

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

Creating Trn/Val/Tst Partitions

```
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.

Shapes and Data Types

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

Multi-Hot Encoding Example

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

Encode the output tokens as multi-hot binary vectors.

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

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

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

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

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

Multi-Hot Encoding Example

```
>>> 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.

IMDB Model

```python
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
```

IMDB Model fit() and evaluate()

```
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.

Tst Accuracy = 89.2%!

n-grams example

- “the cat sat on the mat”
- {"cat", "mat", "on", "sat", "the"}: uni-grams, aka 1-grams
- {"the cat", "cat sat", "sat on", "on the", "the mat"}: bi-grams, aka 2-grams

- Notes:
  - These are sets (duplicates not allowed), instead of bags (duplicates allowed)
  - For the ngrams argument of the TextVectorizer(): passing an integer will create ngrams up to that integer

Multi-Hot Bi-Grams

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

Tst Accuracy = 90.4%!

Term Frequency

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

Term Frequency * Inverse Document Frequency (TF*IDF)

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

https://github.com/keras-team/keras/blob/v2.9.0/keras/layers/preprocessing/index_lookup.py#L801:

return tf.math.log(1 + num_documents / (1 + token_document_counts))

TextVectorization(ngrams = 2)

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

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

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

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

Tst Accuracy = 89.8%!

Exporting Model that Processes Raw Strings

```
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.

```
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")
```

TextVectorization: output_model = ‘int’

```
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.

Sequence Model

```
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.

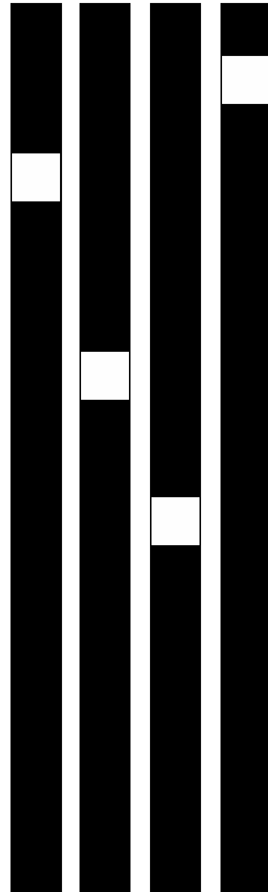
Add a bidirectional LSTM.

Finally, add a classification layer.

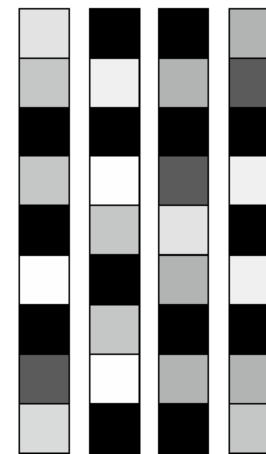
Tst Accuracy = 87%

One-Hot Word Vectors vs Word Embeddings

One-Hot Word Vectors:
distance(queen, king)
==
distance(queen, pancake)



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

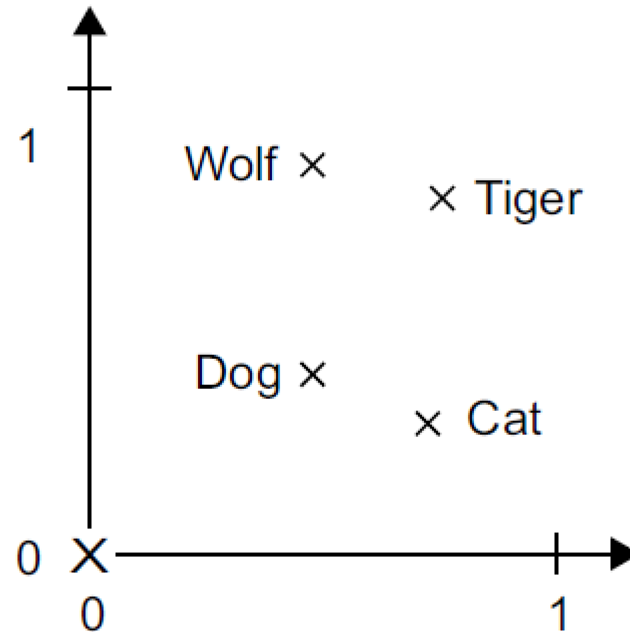


Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Word Embeddings:
queen – woman + man = king

Word Embeddings Example

- Vertical axis: ranges from domesticated to wild
- Horizontal axis: ranges from canine to feline



Embeddings: Trained as Part of Network vs Pretrained

- 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*.

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).

Word index → Embedding layer → Corresponding word vector

Model with 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()
```

Tst Accuracy = 87%

Masking: Zeros are Skipped

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

Embedding Layer with Masking Enabled

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

Tst Accuracy = 88%

Parsing the Global Vectors (GloVe) Word Embeddings File

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

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

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

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

Preparing the GloVe Embeddings Matrix

```
embedding_dim = 100

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

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

Retrieve the vocabulary indexed by our previous TextVectorization layer.

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

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

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

Creating the Embeddings Layer

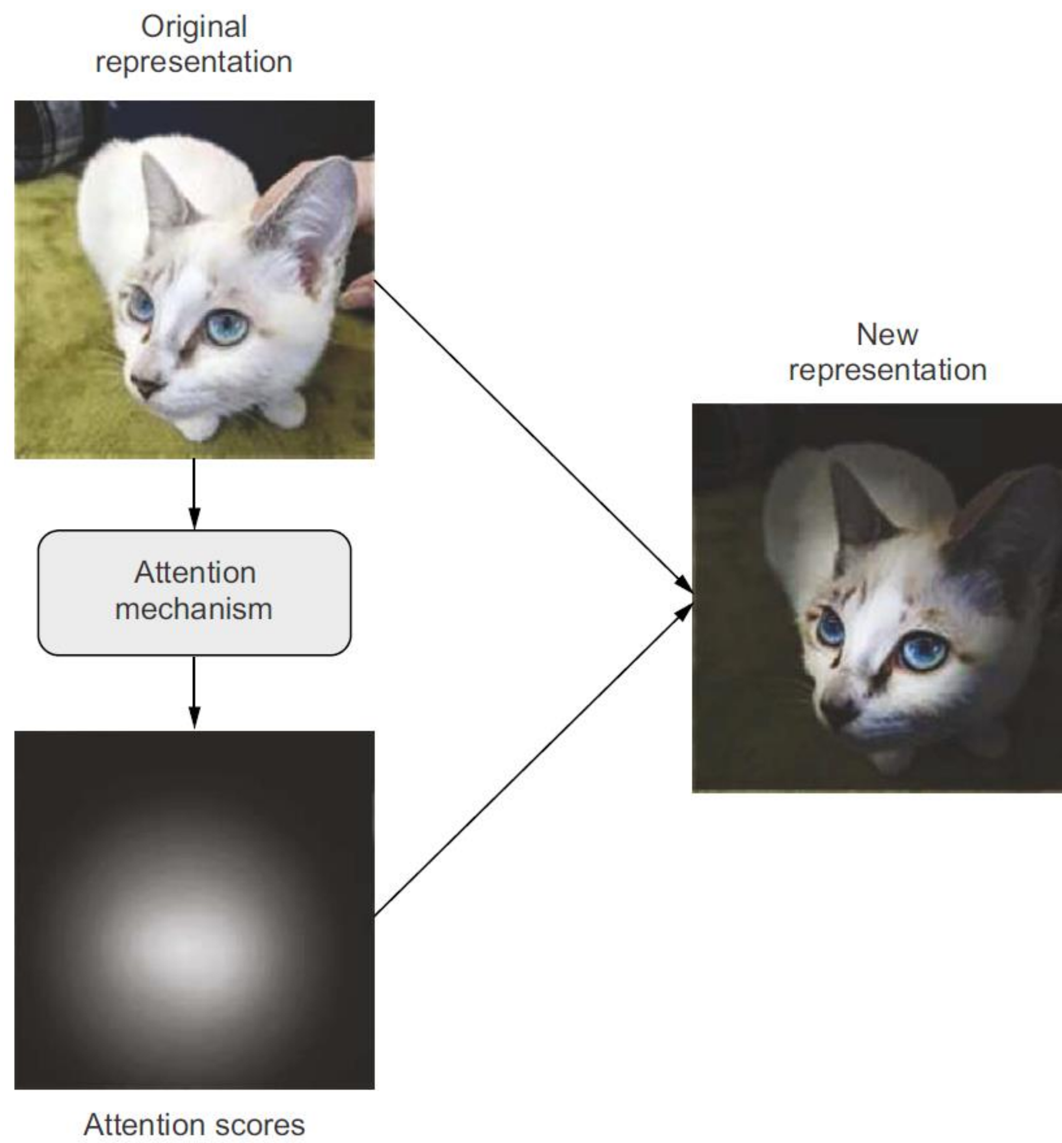
```
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)
```

Model that Uses a Pretrained Embedding

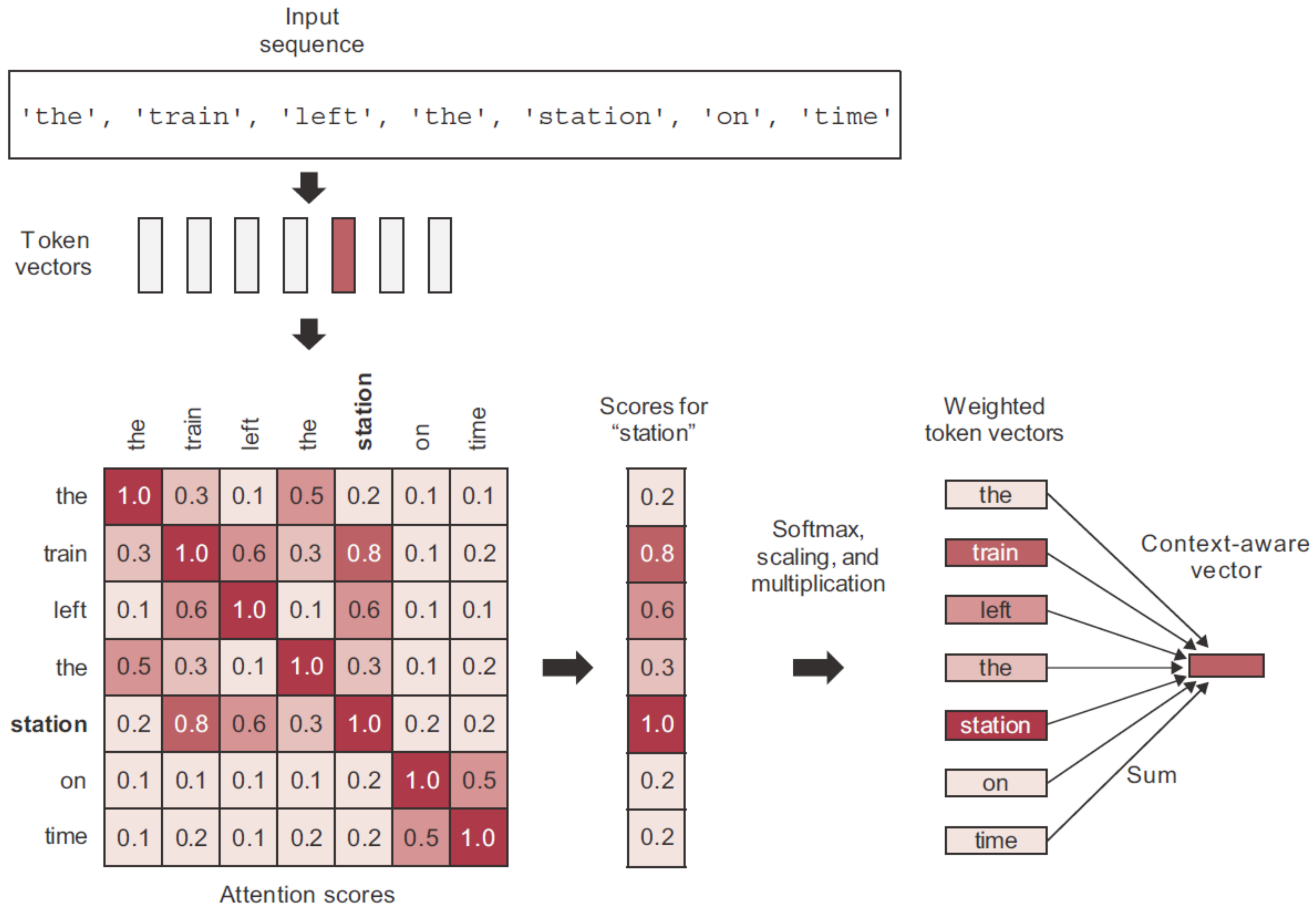
```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
        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

“You’ll find that on this particular task, pretrained embeddings aren’t very helpful”

Attention Scores Applied to an Image

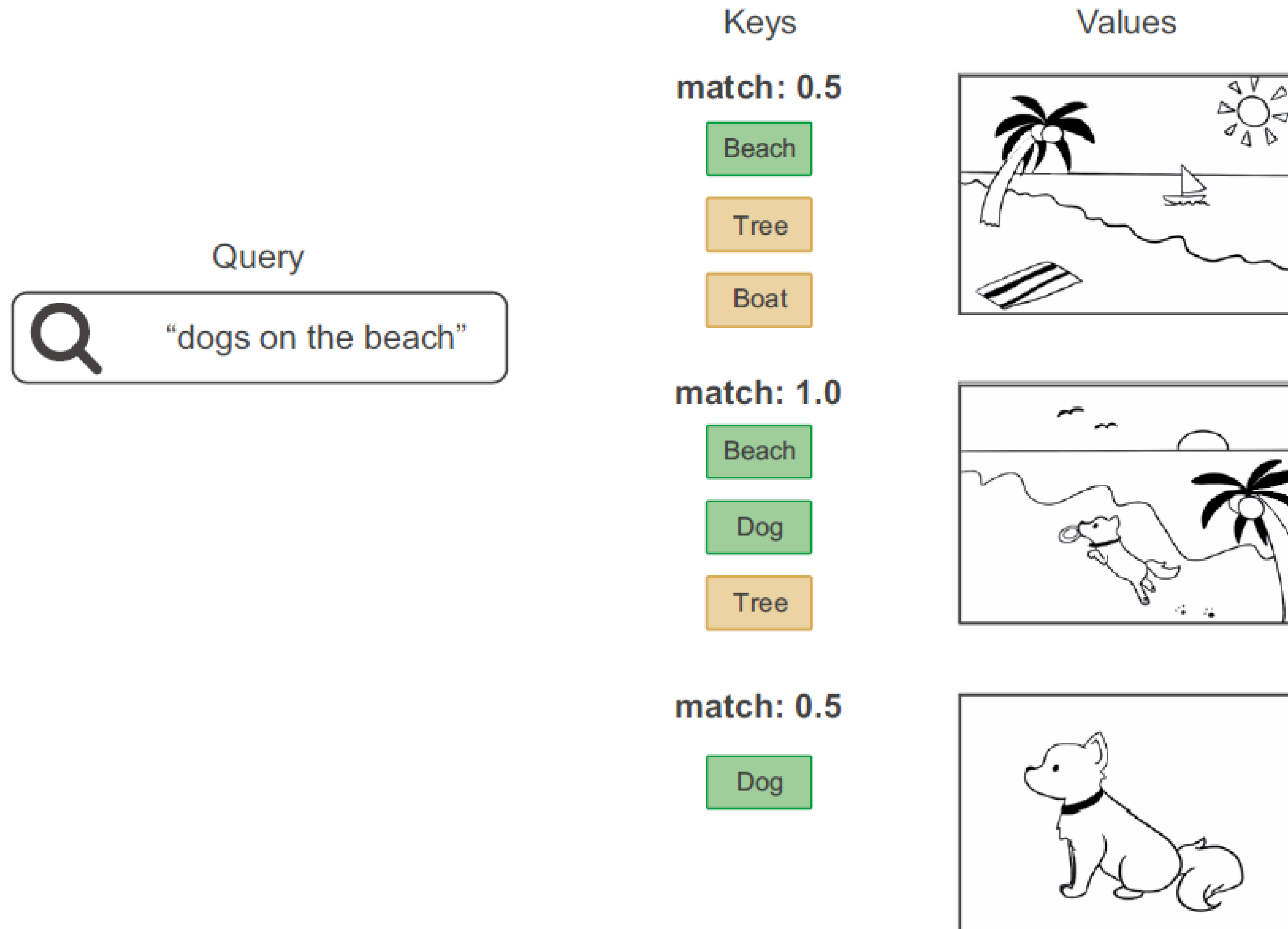


Self-Attention



Queries, Keys, and Values

In search, we match a query against keys to retrieve values …



Self-Attention

- for i in range(num_heads):

  $Q_i = X * Wq_i$ # (512, 768) x (768, 64) = (512, 64); the Query matrix

  $K_i = X * Wk_i$ # (512, 768) x (768, 64) = (512, 64); the Key matrix

  $V_i = X * Wv_i$ # (512, 768) x (768, 64) = (512, 64); the Value matrix

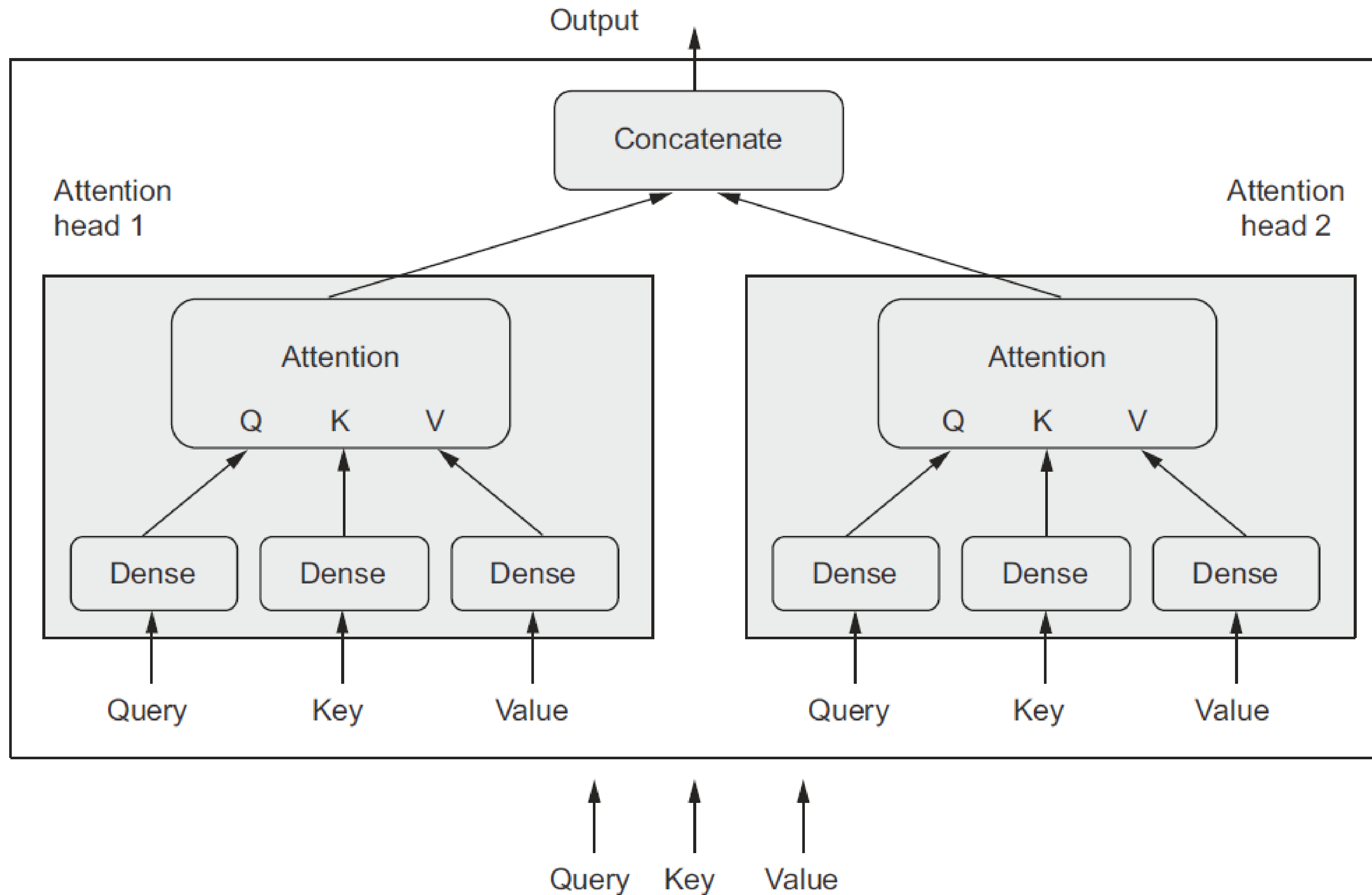
  $A_i = \text{SoftmaxRows}(\frac{Q_i * K_i^T}{\sqrt{64}}) * V_i$ # (512, 512) x (512, 64) = (512, 64)

- Concatenate the 12 Ai matrices horizontally then project

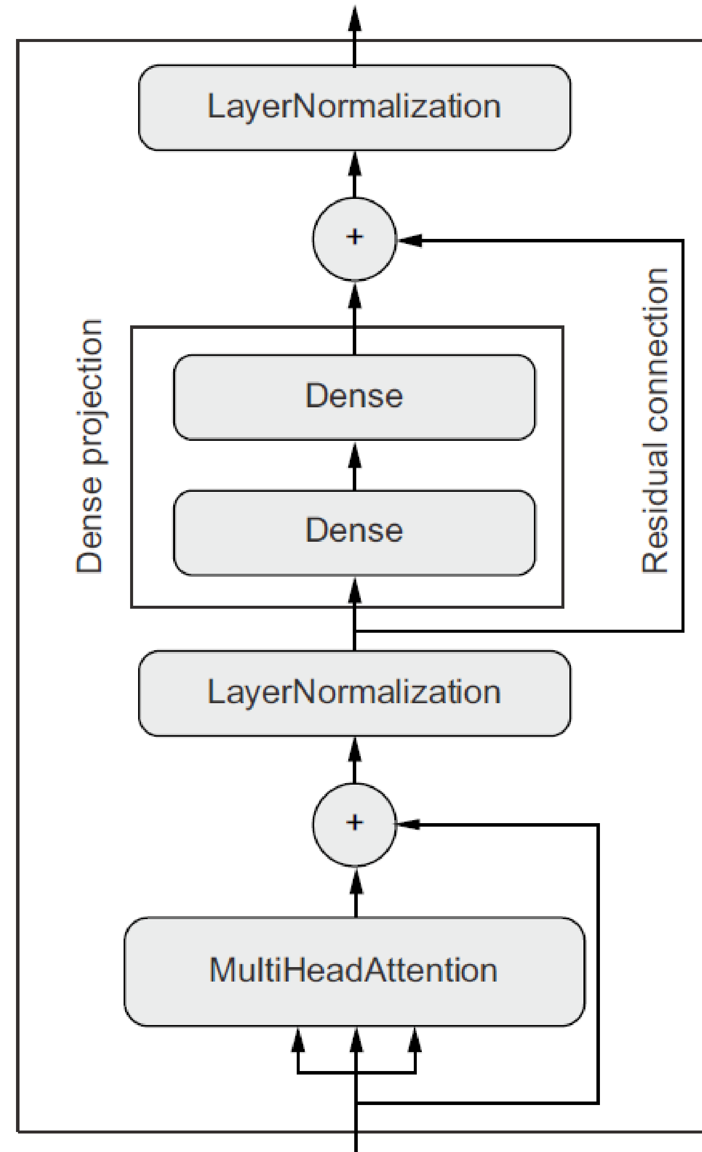
  A * P # (512, 768) x (768, 768) = (512, 768)

Dimensions above are consistent with Bidirectional Encoder Representations from Transformers (BERT) “Base”

Multi-Head Attention



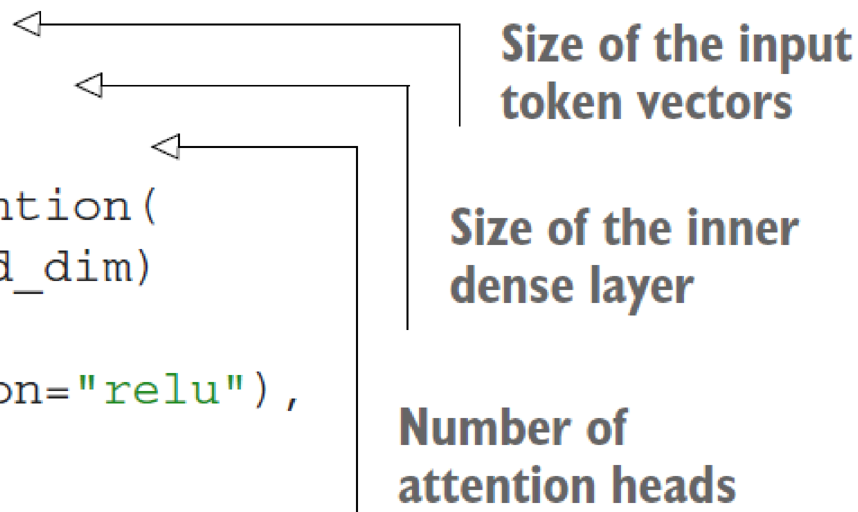
The Transformer Encoder



Transformer Encoder: __init__()

https://arxiv.org/abs/1706.03762
Last sentence of section 3.2.2: d_k = d_model / h
key_dim = embed_dim // num_heads

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

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



TransformerEncoder: call() and get_config()

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

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

Computation goes in call().

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

Implement serialization so we can save the model.

get_config(): returns values of the constructor arguments; used to create the layer

Layer Normalization vs Batch Normalization

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

Input shape: (batch_size, sequence_length, embedding_dim)

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

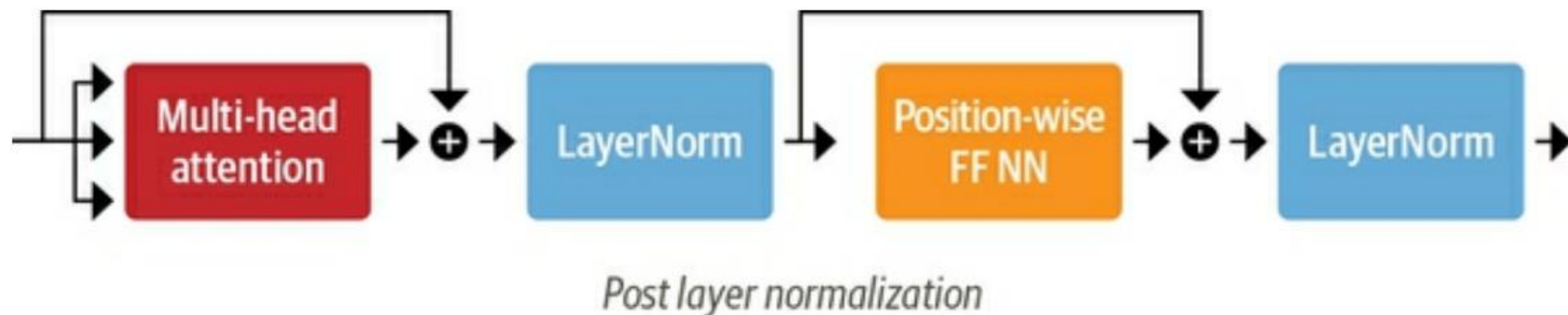
```
def batch_normalization(batch_of_images):
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

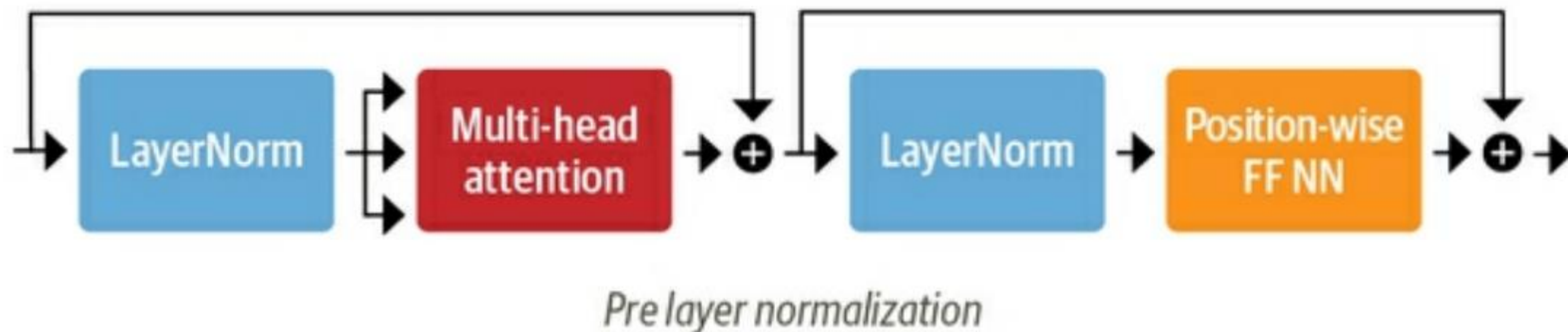
Input shape: (batch_size, height, width, channels)

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

https://learning.oreilly.com/library/view/natural-language-processing/9781098136789/

Post-Layer Normalization vs Pre-Layer Normalization





Using the TranformerEncoder

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

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

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

Tst Accuracy = 87.5%

Features of NLP Approaches

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

Positional Embeddings

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

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

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

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

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

Prepare an Embedding layer for the token indices.

And another one for the token positions

Add both embedding vectors together.

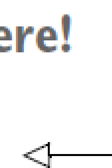
Like the Embedding layer, this layer should be able to generate a mask so we can ignore padding 0s in the inputs. The compute_mask method will called automatically by the framework, and the mask will get propagated to the next layer.

Implement serialization so we can save the model.

Combining PositionalEmbedding and TransformerEncoder

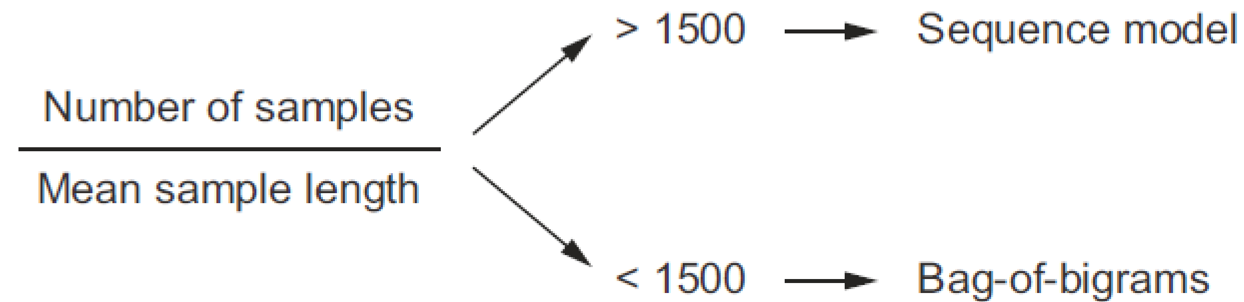
```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

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

Look here!



Tst Accuracy = 88.3%

Proposed Heuristic for Model Selection



What about leveraging a pretrained transformer?

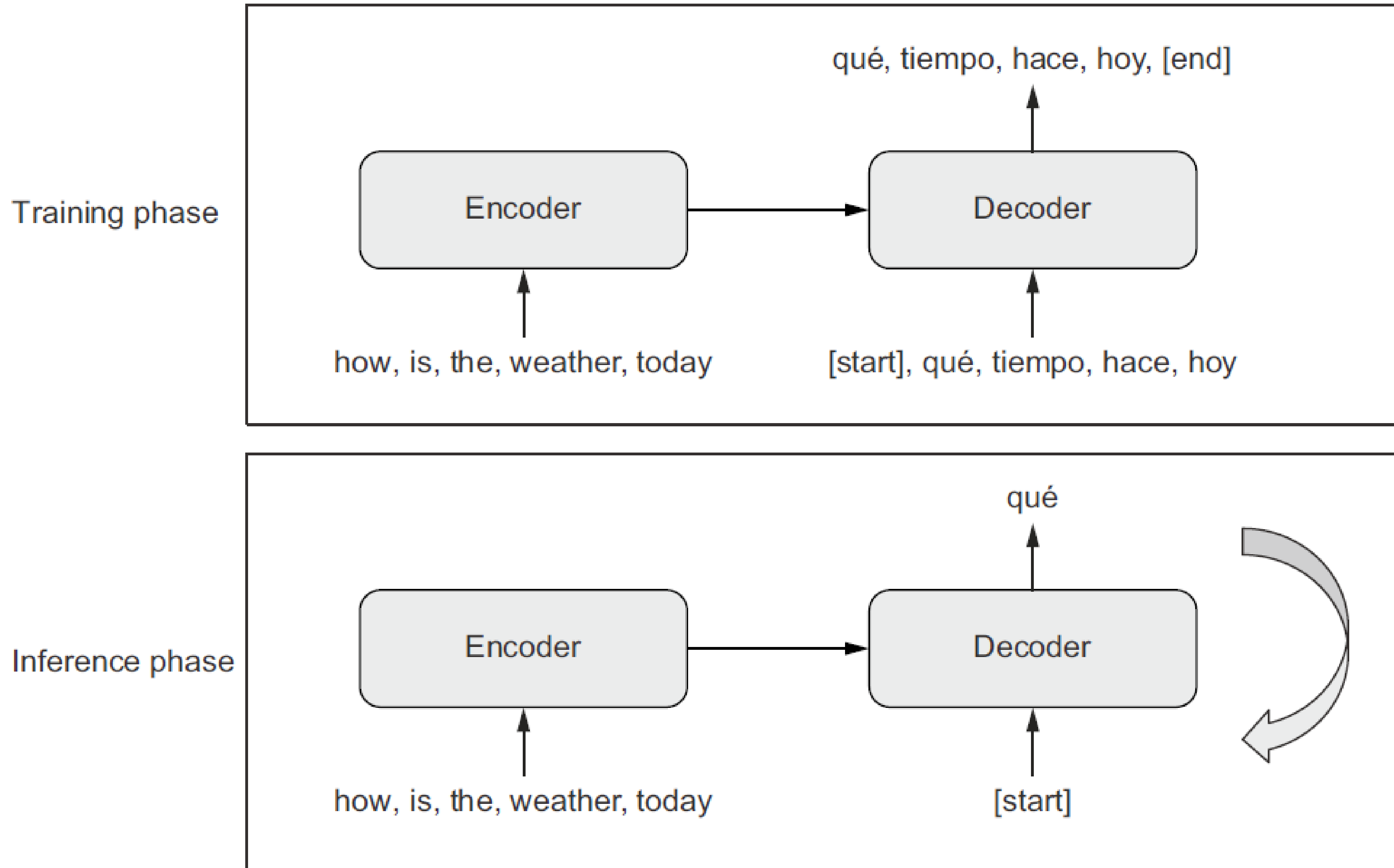
Chapter 11 IMDB Recap

- multi-hot unigrams with MLP: 89.2%
- multi-hot bigrams with MLP: 90.4%
- tf-idf bigrams with MLP: 89.8%
- one-hot encoding with bi-directional LSTM: 87%
- embedding with bi-directional LSTM without masking: 87%
- embedding with bi-directional LSTM with masking: 88%
- frozen, pre-trained embedding with bi-directional LSTM with masking: "not very helpful"
- transformer block from scratch without positional embedding: 87.5%
- transformer block from scratch with positional embedding: 88.3%
- [homework] fine-tuned hugging face microsoft/deberta-v3-large: 97.2%

Sequence-to-Sequence Learning Examples

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

Sequence-to-Sequence Processing



Example English-to-Spanish Translation Data

```
!wget http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip
!unzip -q spa-eng.zip
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

Iterate over the lines in the file.

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

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

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

Splitting Data into Trn, Val, and Tst

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

Vectorizing the English and Spanish Text Pairs

```
import tensorflow as tf
import string
import re

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

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

vocab_size = 15000
sequence_length = 20

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

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

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

The English layer

The Spanish layer

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

Learn the vocabulary of each language.

Preparing Datasets for the Translation Task

```
batch_size = 64

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

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

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

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

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

Use in-memory caching to speed up preprocessing.

Naïve Way to Use an RNN for Seq-to-Seq

```
inputs = keras.Input(shape=(sequence_length,), dtype="int64")
x = layers.Embedding(input_dim=vocab_size, output_dim=128)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

- The target sequence must always be the same length as the source sequence. In practice, this is rarely the case. Technically, this isn't critical, as you could always pad either the source sequence or the target sequence to make their lengths match.
- Due to the step-by-step nature of RNNs, the model will only be looking at tokens $0 \ldots N$ in the source sequence in order to predict token N in the target sequence. This constraint makes this setup unsuitable for most tasks, and particularly translation. Consider translating "The weather is nice today" to French—that would be "Il fait beau aujourd'hui." You'd need to be able to predict "Il" from just "The," "Il fait" from just "The weather," etc., which is simply impossible.

GRU-based Decoder and End-to-End Model

The Spanish target sentence goes here.

Don't forget masking.

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

Predicts the next token

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

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

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

Accuracy = 64%

[better to use the Bi-Lingual Evaluation Understudy (BLEU) score]

Sequence-to-Sequence RNN

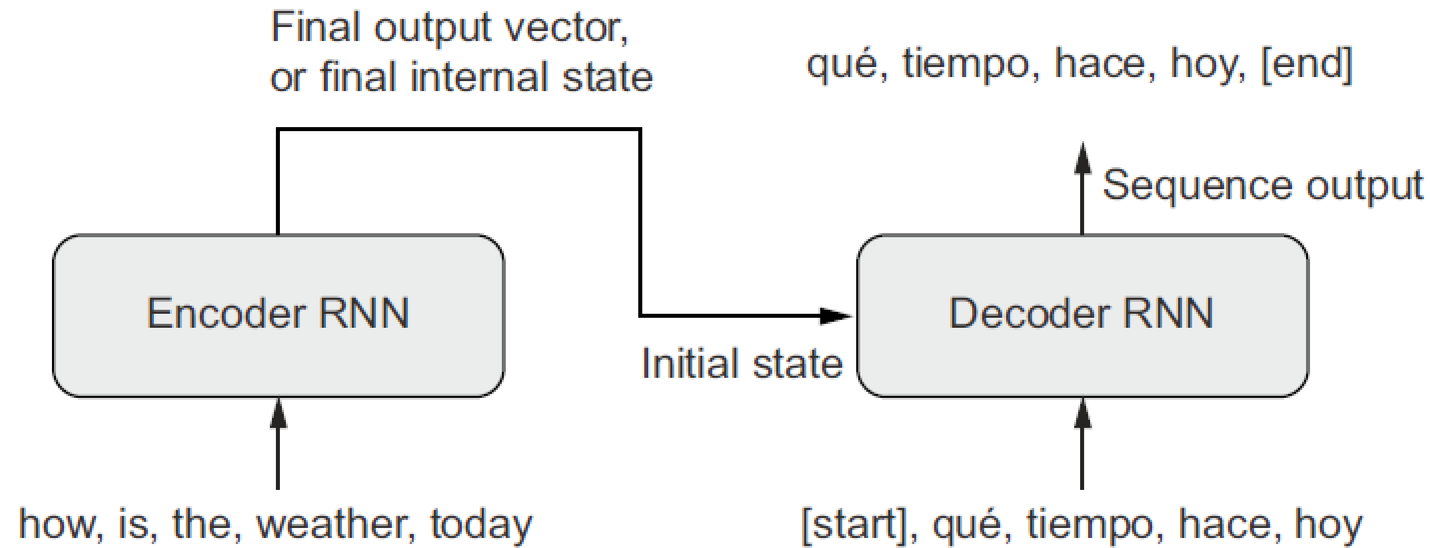


Figure 11.13 A sequence-to-sequence RNN: an RNN encoder is used to produce a vector that encodes the entire source sequence, which is used as the initial state for an RNN decoder.

GRU-based Encoder

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

embed_dim = 256
latent_dim = 1024

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

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

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

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

Seq-to-Seq Inference

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

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

        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

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

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

Seed token

Sample the next token.

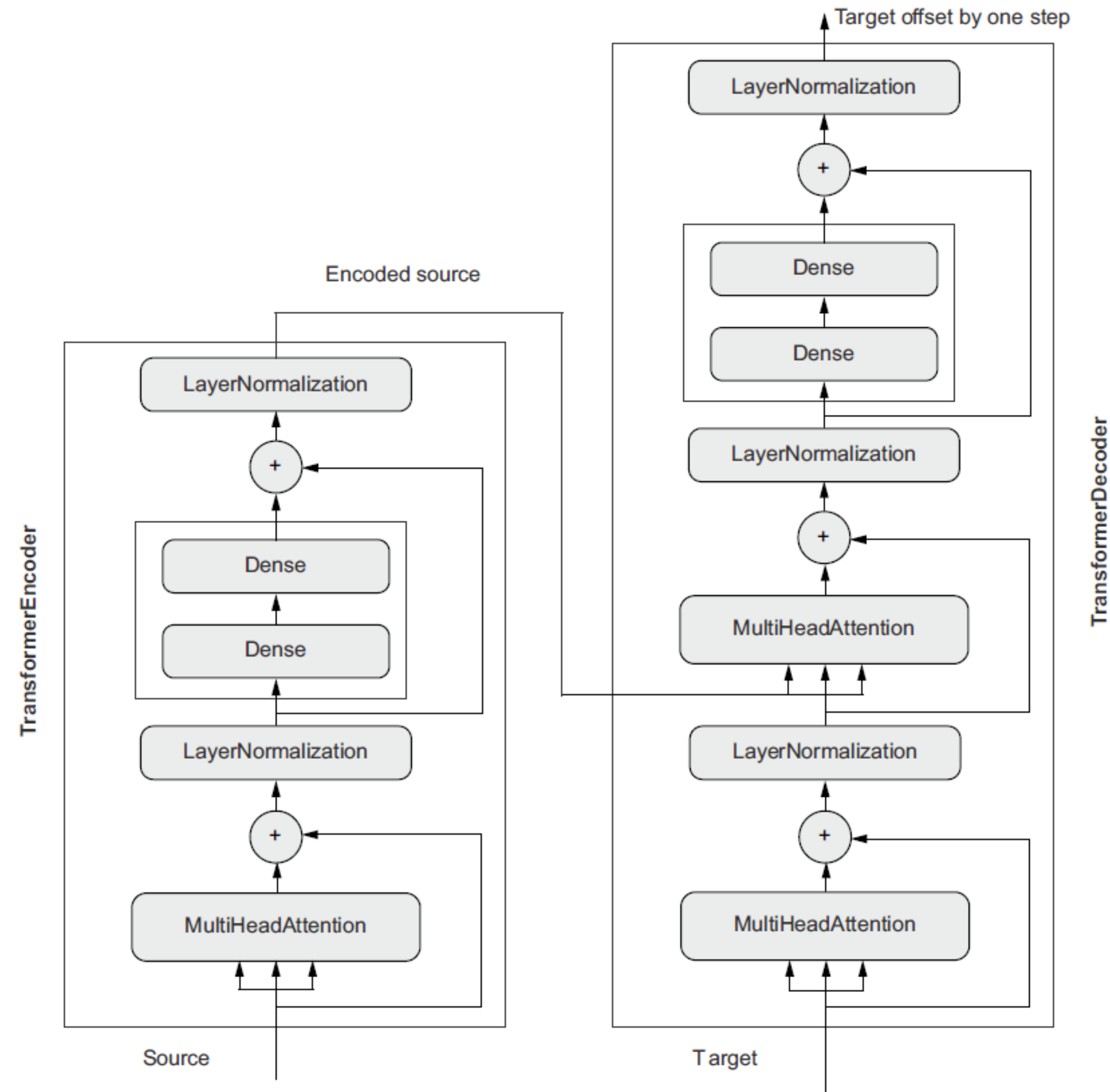
Convert the next token prediction to a string and append it to the generated sentence.

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

Example Output

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

Seq-to-Seq with Transformer



Transformer Decoder

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

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

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

TransformerDecoder::get_causal_attention_mask

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

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

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

TransformerDecoder::call

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

Retrieve the causal mask.

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

Merge the two masks together.

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

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

End-to-End Transformer

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

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

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

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

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

Encode the source sentence.

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

Predict a word for each output position.

Seq-to-Seq Inference

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

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

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

Sample the next token.

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

Exit condition

Example Output

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

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

https://en.wikipedia.org/wiki/BLEU

BiLingual Evaluation Understudy (BLEU)
[used for machine translation evaluation]

There is not a single definition of BLEU, but a whole family of them, parametrized by the weighting vector $w := (w_1, w_2, \cdots)$. It is a probability distribution on $\{1, 2, 3, \cdots\}$, that is, $\sum_{i=1}^{\infty} w_i = 1$, and $\forall i \in \{1, 2, 3, \cdots\}, w_i \in [0, 1]$.

With a choice of w, the BLEU score is

$$BLEU_w(\hat{S}; S) := BP(\hat{S}; S) \cdot \exp\left(\sum_{n=1}^{\infty} w_n \ln p_n(\hat{S}; S)\right)$$

In words, it is a weighted geometric mean of all the modified n-gram precisions, multiplied by the brevity penalty. We use the weighted geometric mean, rather than the weighted arithmetic mean, to strongly favor candidate corpuses that are simultaneously good according to multiple n-gram precisions.

The most typical choice, the one recommended in the original paper, is $w_1 = \cdots = w_4 = \frac{1}{4}$

https://web.stanford.edu/~jurafsky/slp3/10.pdf

Search: Greedy vs Beam vs Monte Carlo

$p(t_3|\text{source}, t_1, t_2)$

$p(t_2|\text{source}, t_1)$

$p(t_1|\text{source})$

start → .4 → ok → .7 → ok → 1.0 → </s>
ok → .2 → yes → 1.0 → </s>
ok → .1 → </s>
start → .5 → yes → .3 → ok → 1.0 → </s>
yes → .4 → yes → 1.0 → </s>
yes → .2 → </s>
start → .1 → </s>

t_1 t_2 t_3

https://web.stanford.edu/~jurafsky/slp3/10.pdf

Beam Decoding with Beam Width = 2

log P ("the green witch arrived"|x)
= log P (the|x) + log P(green|the,x)
+ log P(witch | the, green,x)
+logP(arrived|the,green,witch,x)
+log P(END|the,green,witch,arrived,x)

-2.7

log P (arrived the|x)
= -2.3
the

log P(arrived|x)
=-1.6
arrived

-.69

log P(arrived witch|x)
= -3.9
-2.3
witch

-1.6
start

-.92
log P(the|x)
=-.92
the

-.69
log P(the green|x)
= -1.6
green

-1.6
-3.2
mage
-2.1

-.51
witch

-2.5
-.36
arrived
-.22
END
-2.3
-4.8
at

-3.7
-1.6
came

-1.2
log P(the witch|x)
= -2.1
witch

-.11
-2.2
arrived

-2.7
-.51
END

-1.61
-3.8
by

-2.3
-4.4
who

log $P(y_1|x)$ log $P(y_2|y_1,x)$ log $P(y_3|y_2,y_1,x)$ log $P(y_4|y_3,y_2,y_1,x)$ log $P(y_5|y_4,y_3,y_2,y_1,x)$

y_1 y_2 y_3 y_4 y_5

Length Normalization

Without normalization

$$\begin{aligned} score(y) &= \log P(y|x) \\ &= \log(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)...P(y_t|y_1,...,y_{t-1},x)) \\ &= \sum_{i=1}^{t} \log P(y_i|y_1,...,y_{i-1},x) \end{aligned}$$

With normalization, where ‘T’ = target sequence length

$$score(y) = -\log P(y|x) = \frac{1}{T}\sum_{i=1}^{t} -\log P(y_i|y_1,...,y_{i-1},x)$$

 https://web.stanford.edu/~jurafsky/slp3/10.pdf

Beam Search Pseudocode

```
function BEAMDECODE(c, beam_width) returns best paths

  y0, h0 ← 0
  path ← ()
  complete_paths ← ()
  state ← (c, y0, h0, path)        ;initial state
  frontier ← ⟨state⟩        ;initial frontier

  while frontier contains incomplete paths and beamwidth > 0
     extended_frontier ← ⟨⟩
     for each state ∈ frontier do
        y ← DECODE(state)
        for each word i ∈ Vocabulary do
           successor ← NEWSTATE(state, i, y_i)
           new_agenda ← ADDTOBEAM(successor, extended_frontier, beam_width)

     for each state in extended_frontier do
        if state is complete do
           complete_paths ← APPEND(complete_paths, state)
           extended_frontier ← REMOVE(extended_frontier, state)
           beam_width ← beam_width - 1
     frontier ← extended_frontier

  return completed_paths
```

```
function NEWSTATE(state, word, word_prob) returns new state

function ADDTOBEAM(state, frontier, width) returns updated frontier

  if LENGTH(frontier) < width then
     frontier ← INSERT(state, frontier)
  else if SCORE(state) > SCORE(WORSTOF(frontier))
     frontier ← REMOVE(WORSTOF(frontier))
     frontier ← INSERT(state, frontier)
  return frontier
```

 https://web.stanford.edu/~jurafsky/slp3/2.pdf

Tokenization: Byte Pair Encoding (BPE)

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Summary

- There are two kinds of NLP models: *bag-of-words models* that process sets of words or N-grams without taking into account their order, and *sequence models* that process word order. A bag-of-words model is made of `Dense` layers, while a sequence model could be an RNN, a 1D convnet, or a Transformer.
- When it comes to text classification, the ratio between the number of samples in your training data and the mean number of words per sample can help you determine whether you should use a bag-of-words model or a sequence model.
- *Word embeddings* are vector spaces where semantic relationships between words are modeled as distance relationships between vectors that represent those words.
- *Sequence-to-sequence learning* is a generic, powerful learning framework that can be applied to solve many NLP problems, including machine translation. A sequence-to-sequence model is made of an encoder, which processes a source sequence, and a decoder, which tries to predict future tokens in target sequence by looking at past tokens, with the help of the encoder-processed source sequence.
- *Neural attention* is a way to create context-aware word representations. It's the basis for the Transformer architecture.
- The *Transformer* architecture, which consists of a `TransformerEncoder` and a `TransformerDecoder`, yields excellent results on sequence-to-sequence tasks. The first half, the `TransformerEncoder`, can also be used for text classification or any sort of single-input NLP task.

Bonus https://learning.oreilly.com/library/view/natural-language-processing/9781098136789/

Visualizing Attention

from bertviz import head_view

Layer: 0 Attention: All

[CLS] time flies like an arrow [SEP] fruit flies like a banana [SEP]

Layer: 0 Attention: All

[CLS] time flies like an arrow [SEP] fruit flies like a banana [SEP]

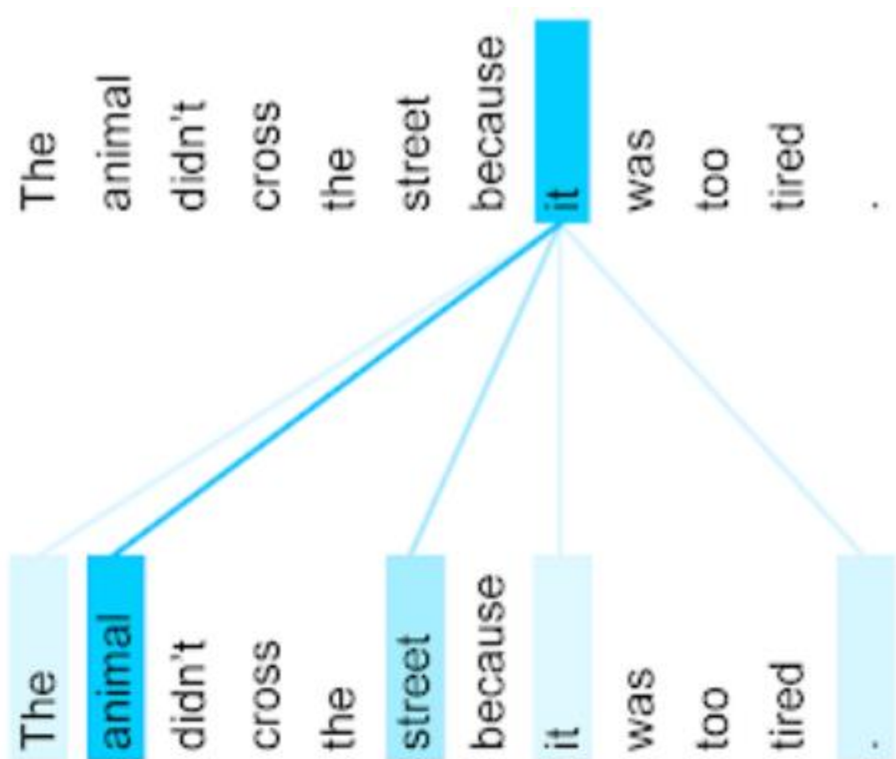
Layer: 0 Attention: All

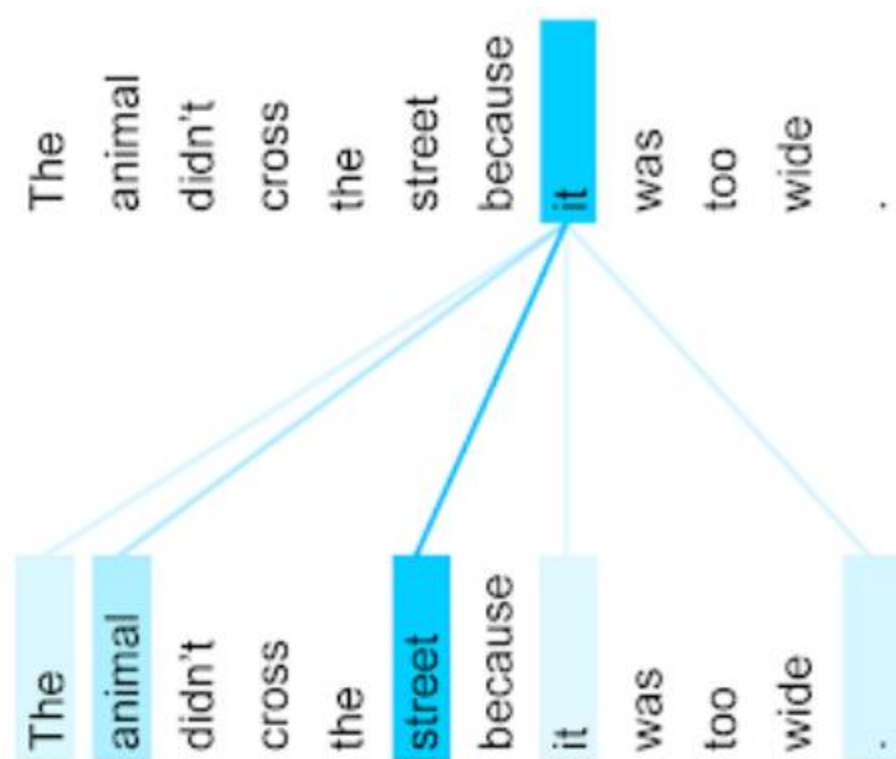
[CLS] time flies like an arrow [SEP] fruit flies like a banana [SEP]

Bonus

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

Example of Attention





Bonus https://learning.oreilly.com/library/view/natural-language-processing/9781098136789/

https://ai.googleblog.com/2020/02/exploring-transfer-learning-with-t5.html

The Transformers Timeline

ULMFiT BERT RoBERTa XLM-R DeBERTa GPT-Neo

Transformer GPT GPT-2 DistilBERT GPT-3 T5 GPT-J

2017 2018 2019 2020 2021

ULMFiT: Universal Language Modeling Fine-tuning
GPT: Generative Pre-trained Transformer
BERT: Bidirectional Encoder Representations from Transformers
RoBERTa: Robust BERT approach
DistilBERT: Distilled (made smaller) BERT
XLM-R: cross-lingual Language Model – RoBERTa
DeBERTa: Decoding-enhanced BERT with disentangled attention
T5: Text-To-Text Transfer Transformer

BlenderBot: Chatbot Example

- Blog post
  https://ai.facebook.com/blog/state-of-the-art-open-source-chatbot/
- Install and run

```
conda install pytorch torchvision cudatoolkit=10.1 -c pytorch
git clone https://github.com/facebookresearch/ParlAI.git
cd ParlAI
python setup.py develop --user
pip install 'git+https://github.com/rsennrich/subword-nmt.git#egg=subword-nmt'
python parlai/scripts/safe_interactive.py -t blended_skill_talk -mf zoo:blender/blender_90M/model
```

[context]: your persona: i love to dance.

your persona: i'm a teacher.

Enter Your Message: Have you ever tried an interpretive dance class?

[TransformerGenerator]: no i haven ' t , but i ' d love to try it . what do you do for fun ?

Enter Your Message: I love to do the twist. How about you?

[TransformerGenerator]: i like to watch hockey . it ' s my favorite . what ' s your favorite sport to watch ?