Natural Language Processing with RNNs and Attention

Deep Learning

Stijn Lievens and Sabine Devreese

2023-2024

Natural Language Processing with RNNs and Attention

Natural Language Processing

- The *Turing test* (from 1950) essentially is a test of natural language processing.
- **Natural Language Processing (NLP)** is about making machines master written and spoken language. Subtasks include:
  - text classification
  - translation
  - summarization
  - question answering
  - etc.

Common Approach to NLP Tasks

- Recurrent neural networks (RNNs) are commonly used for NLP tasks.
  - These will be explored further in this chapter.
- **Attention mechanisms** are neural network components that learn to select that part of the input that the rest of the model should focus on (at each time step).
- **Transformers** are very successful attention-only architectures that are used in many NLP tasks.
  - Transformers are *not* RNNs!

16.1 Generating Shakespearean Text Using a Character RNN

The Task

- In the 2015 blog post **The Unreasonable Effectiveness of Recurrent Neural Networks** Andrej Karpathy showed how to train an RNN to predict the next character in a sentence.
- An example of text generated by the network after it was trained on all of Shakespeare's works:

```
PANDARUS:

Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.
```

The Task

- Our task will be to train such a character-level language model on the works of Shakespeare.
  - Even though the text itself doesn’t make much sense, the model will learn to spell (most) words correctly and even to use correct punctuation.

16.1.1 Creating the Training Dataset

Downloading the Data

Use `tf.keras.utils.get_file()` to download the data.

```
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() # shakespeare_text is now a string
```

Looking at the Data

- It is always good to look at the data before proceeding.

```
print(shakespeare_text[:80])
```

yields

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

- OK. The text looks like Shakespeare.

Encoding the Data

- We use the `TextVectorization` layer (from Chapter 13) to encode the text.
  - We will encode each lower-case character as an integer.

```
text_vec_layer = tf.keras.layers.TextVectorization(
  split="character", standardize="lower")
# shakespeare_text is a string and adapt expects a dataset or list
text_vec_layer.adapt([shakespeare_text])
encoded = text_vec_layer([shakespeare_text])[0]
```

- Now, `encoded` is a 1D tensor of integers.

Removing Padding and Unknown Tokens

- The `TextVectorization` layer adds tokens for ‘padding’ and ‘unknown’.
  - These have integer indices 0 and 1, respectively.
  - We are not going to use these tokens, so we remove them.

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

Creating Inputs and Targets

- We will train a sequence-to-sequence RNN.
  - E.g. when the input is “to be or not to b” (without the final “e”), the target will be “o be or not to be”.

```
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)
```

Creating Inputs and Targets

- The code on the previous slide is quite similar to the `to_windows` method from the previous chapter.
  - It optionally shuffles the windows.
  - It batches the windows
  - On the last line it creates the inputs and targets.

```
list(to_dataset(text_vec_layer(["To be"])[0], length=4))
```

yields encoded versions of “to b” as input and “o be” as target:

```
[(<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[ 4,  5,  2, 23]])>,
<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[ 5,  2, 23,  3]])>)]
```

Create the Training/Val/Test Sets

- We will use approximately 90% of the data for training, 5% for validation, and 5% for testing.
  - Remember: total length of encoded is 1,115,394.
- We will use sequences of length 100.

```
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)
```

16.1.2 Building and Training the Char-RNN Model

The Model

- Our dataset is reasonably large, and language modelling is not an easy task.
  - We will need more than a simple RNN.
  - We will use a GRU layer with 128 units.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

The Model (Ctd.)

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    ... ])
```

- The model starts with an `Embedding` layer.
  - This way, it can learn to map similar characters close together.
  - The `input_dim` equals the number of tokens. The `output_dim` is a hyperparameter (16 in this case).
  - Inputs to the embedding will be `(batch_size, window_length)`, while the outputs are `(batch_size, window_length, output_dim)`.

The Model (Ctd.)

```
model = tf.keras.Sequential([
    ...
    tf.keras.layers.Dense(n_tokens, activation="softmax")])
```

- The output layer is a `Dense` layer with 39 (`n_tokens`) units and `softmax` activation function:
  - We want to predict the probability of each possible (next) character.
    - This also fixes the loss function we will use: categorical cross-entropy.

The Model (Extra code)

We check the shape of the inputs and outputs of the model:

```
for features, labels in train_set.take(1):
  print(f"Input to the model has shape: {features.shape}")
  output = model(features)
  print(f"Output of the model has shape: {output.shape}")
  print(f"Labels have shape: {labels.shape}")
```

yields

```
Input to the model has shape: (32, 100)
Output of the model has shape: (32, 100, 39)
Labels have shape: (32, 100)
```

Compiling and Training the Model

- Compiling and training the model is done with the following (familiar) code:

```
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])
```

Final Model

- The model we trained doesn't include text preprocessing.
  - We can include the `TextVectorization` layer in the model.
  - We also need to subtract 2 from the character IDs.

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

Using the Final Model

- Let's predict the next character after "To be or not to b":

```
# Index 0 because we only have one sample
# Index -1 because we only want the last (predicted) character
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
# Do not forget to add 2 back to the character ID
text_vec_layer.get_vocabulary()[y_pred + 2]
```

yields

```
'e'
```

16.1.3 Generating Fake Shakespearean Text

Greedy Decoding

- To generate new text, we can feed the model some text:
  - make the model predict the *most likely* next character
  - add this character to the text and repeat
- This is called **greedy decoding** because we always choose the most likely next character.
  - In practice this leads to repetitive text.

Sampling the Next Character

- A better approach is to sample the next character from the probability distribution predicted by the model.
  - We can use `tf.random.categorical()` for this.
  - The `categorical` function samples random class indices given class *log probabilities*. (*logits*)

Sampling the Next Character

```
log_probas = tf.math.log([[0.5, 0.4, 0.1]])  # probas = 50%, 40%, and
          10%
tf.random.set_seed(42)
tf.random.categorical(log_probas, num_samples=8)  # draw 8 samples
```

yields

```
<tf.Tensor: shape=(1, 8), dtype=int64, numpy=array([[0, 1, 0, 2, 1, 0,
0, 1]])>
```

Note: `tf.random.categorical()` always works on batches of data, hence `[[0.5, 0.4, 0.1]]` and not `[0.5, 0.4, 0.1]`.

Using a Temperature

- The **temperature** is a number that we divide the logits by.
  - A higher temperature leads to a more uniform probability distribution.
  - A lower temperature makes high-probability characters even more likely.
- We can use the temperature to control the randomness of the generated text.
  - A high temperature leads to more diverse text.
  - A low temperature leads to more rigid and precise text.

Using a Temperature (Extra)

- Suppose we have 3 classes with probabilities `[0.7, 0.2, 0.1]`.
- Using a temperature of 2, the probabilities become

```
probas = np.array([0.7,0.2,0.1])
logits = np.log(probas) # array([-0.35667494, -1.60943791, -2.30258509])
logits_t2 = logits / 2 # Divide logits by 2
# Compute probabilities from rescaled logits (using softmax)
probas_t2 = np.exp(logits_t2) / np.sum(np.exp(logits_t2))
probas_t2  # array([0.52287938, 0.27949079, 0.19762983])
```

You can see that the probabilities became more uniform.

Using a Temperature (Extra)

- Using a temperature of 0.5, the probabilities become

```
probas = np.array([0.7,0.2,0.1])
logits = np.log(probas) # array([-0.35667494, -1.60943791, -2.30258509])
logits_t05 = logits / 0.5
probas_t05 = np.exp(logits_t05) / np.sum(np.exp(logits_t05))
probas_t05 # array([0.90740741, 0.07407407, 0.01851852])
```

You see that the most likely value became even more likely when using the low temperature.

Sampling the Next Character

The `next_char` helper function picks the next character to add to the input text. It uses a temperature of 1.0 by default.

```
def next_char(text, temperature=1):
    # shakepeare_model is the model we trained earlier
    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]
    # text_vec_layer is the layer we adapted earlier
    return text_vec_layer.get_vocabulary()[char_id + 2]
```

Generating Text

With `extend_text` we can generate text of arbitrary length.

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

Examples of Generated Text

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

yields

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

while

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

yields

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

Other Sampling Strategies

- Instead of sampling from all characters, we could sample from the top k characters.
- Or, from the set of top characters whose total probability is at least p (this is called *nucleus sampling*).
- Or use *beam search* (as discussed later in this chapter)

16.1.4 Stateful RNN

16.2 Sentiment Analysis

IMDb Movie Reviews

- The IMDb Dataset is a dataset of 50000 movie reviews in English.
- Each movie review is classified as either “positive” or “negative”.
- *Sentiment classification* on the IMDb dataset is the “hello world” of NLP.

IMDb Movie Reviews

You already used this dataset in the exercises of chapters 10 and 13. The models you developed there weren't sequence models and didn't take the order of the words into account.

Loading the Dataset

- This time, we will load the dataset using `tensorflow_datasets` (see Chapter 13).

```
import tensorflow_datasets as tfds

raw_train_set, raw_valid_set, raw_test_set = tfds.load(
    name="imdb_reviews",
    # Note: we will use 10% of the train data for validation.
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
tf.random.set_seed(42)
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)
```

Inspecting the Data

As always, let's inspect the data before proceeding.

```
for review, label in raw_train_set.take(4): #Use non-batched version
    print(review.numpy().decode("utf-8")[:75], "...")
    print("Label:", label.numpy())
```

yields

```
This was an absolutely terrible movie. Don't be lured in by Christopher
Wal ...
Label: 0
I have been known to fall asleep during films, but this is usually due
to a ...
Label: 0
Mann photographs the Alberta Rocky Mountains in a superb fashion, and
Jimmy ...
Label: 0
This is the kind of film for a snowy Sunday afternoon when the rest of
the  ...
Label: 1
```

Tokenizing the Reviews

- Obviously, we can't feed strings into the neural network.
- We will use `TextVectorization` to tokenize the reviews bases on words.
  - Spaces are used to separate words. This may not work well for languages like Chinese (no spaces between words) or German (long compound words).
  - Even in English, spaces are not perfect, e.g. "San Francisco" and "#ILoveDeepLearning".

Other Ways of Tokenizing

- **Byte Pair Encoding** (BPE) is a popular tokenization algorithm.
- BPE starts by splitting the whole training set into individual characters (including spaces).
- It then repeatedly merges the most frequent pairs (of tokens) until the vocabulary reaches the desired size.
  - The algorithm remembers the merges, so they can be applied to tokenize new data.
- Many other tokenization algorithms exist, which can be found in **TensorFlow Text** or in **Hugging Face** library.

Using the `TextVectorization` Layer

- For this (simple) example, tokenizing on spaces is fine.
  - We will limit the vocabulary to 1000 tokens. Thus “padding”, “unknown” and the 998 most frequent words.

```
vocab_size = 1000
text_vec_layer =
          tf.keras.layers.TextVectorization(max_tokens=vocab_size)
# Adapt only on the reviews, not the labels
text_vec_layer.adapt(train_set.map(lambda reviews, labels: reviews))
```

Building and Compiling the Model

- We will create a model that starts with the `TextVectorization` layer.
- Next, we will use an `Embedding` layer to learn a representation of the words.
- The `GRU` layer will process the sequence of word embeddings.
- Finally, we will use a `Dense` layer with a single unit to predict the sentiment.
  - We will use the sigmoid activation function, since we are doing binary classification.

Building and Compiling the Model (Ctd.)

```
embed_size = 128
tf.random.set_seed(42)
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)
```

Model not Learning Anything

- However, this model doesn't seem to be learning anything.
  - It is not better than random guessing.
- The problem is with the padding of the reviews.

```
# Code not in book
text_vec_layer([
  ["Good movie!"],
  ["This really was a terrible movie. All the characters were very
          boring."]])
```

yields

```
<tf.Tensor: shape=(2, 12), dtype=int64, numpy=
array([[ 50,  18,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [ 11,  63,  14,   4, 383,  18,  32,   2, 102,  67,  51, 355]])>
```

Shorter Reviews are Padded

```
<tf.Tensor: shape=(2, 12), dtype=int64, numpy=
array([[ 50,  18,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [ 11,  63,  14,   4, 383,  18,  32,   2, 102,  67,  51, 355]])>
```

- As you can see from the (repeated) result above, the short review is padded with zeros until it reaches the length of the longest review (in the batch).
- The GRU layer, however, will process all the zeros, updating its state each time.
  - In the end, the GRU layer ends up *forgetting* what the actual review was about!

16.2.1 Masking

Using Masking on the Embedding Layer

- Basically, we want the GRU layer to stop updating its state when it encounters a padding token.
  - I.e. the padding tokens should be ignored.
- This is easy! Simply add `mask_zero=True` to the `Embedding` layer.
  - The padding tokens (with ID 0) will be ignored by downstream layers.

```
# Only change is the mask_zero=True
tf.keras.layers.Embedding(vocab_size, embed_size, mask_zero=True)
```

Mask Example (Extra)

```
# Code not in book to show mask
embedding_layer = tf.keras.layers.Embedding(
  vocab_size, embed_size, mask_zero=True)
ids = text_vec_layer(
  [["Good movie!"],
  ["This really was a terrible movie. All the characters were very
          boring."]])
# Remember. ids =
# array([[ 50,  18,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
#        [ 11,  63,  14,   4, 383,  18,  32,   2, 102,  67,  51, 355]])
embedding_layer.compute_mask(ids)
```

yields

```
<tf.Tensor: shape=(2, 12), dtype=bool, numpy=
array([[ True,  True, False, False, False, False, False, False, False,
        False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True]])>
```

Mask Example (Ctd.)

- Note, how each padding token (zero) yields a `False` in the mask.
  - So `False` means “ignore this token/input”.

Mask Propagation

- The embedding layer produces the mask, which is a tensor.
- This mask tensor is then automatically propagated by the model to the next layer.
  - If that layer's `call` method has a `mask` argument, it will receive the mask.
- Each layer may handle the mask differently, but in general inputs for which the corresponding position in the mask is `False` will be ignored.
  - When a recurrent layer encounters a masked time step, it simply copies the output from the previous time step.

Mask Propagation (Ctd.)

- Masks are only propagated by layers who have their `supports_masking=True`.
  - A recurrent layer has `supports_masking=True`, only when `return_sequences=True`.
  - When `return_sequences=False`, the recurrent layer only returns the last output, so there is no need to propagate the mask.
- Many layers supports masking:
  - `SimpleRNN`, `GRU`, `LSTM`, `Dense`, ...
  - However, convolutional layers do *not* support masking.

Masks on the Output Layer

- If the mask propagates to the output layer, it gets applied to the losses as well.
  - Thus, masked time steps will not contribute to the loss (their loss will be 0).
  - In our case, the model does not output sequences, so this is not relevant.

Masking and the Functional API

- It is possible to compute the mask ‘manually’ and pass it to the next layer using the functional API.
  - This is useful when you want to use a layer (e.g. a convolutional layer) that does not support masking.

Ragged Tensors

- Instead of using masking, we could (in this case) also use **ragged** tensors.
  - Keras's recurrent layers have built-in support for ragged tensors.

```
text_vec_layer_ragged = tf.keras.layers.TextVectorization(
    max_tokens=vocab_size, ragged=True) # Note: ragged=True
text_vec_layer_ragged.adapt(train_set.map(lambda reviews, labels:
          reviews))
text_vec_layer_ragged(["Great movie!", "This is DiCaprio's best role."])
```

yields

```
<tf.RaggedTensor [[86, 18], [11, 7, 1, 116, 217]]>
```

Note how the padding tokens are not included in the output.

16.2.2 Reusing Pretrained Embeddings and Language Models

Embeddings

- Embeddings tend to “cluster” similar words together.
  - E.g. “awesome” and “amazing” will be close together, while “awful” and “terrible” will be close together (on the other side of the embedding space)
- These embeddings are only trained on 25000 reviews.

Reusing Existing Embeddings

- It is possible to reuse existing embeddings.
  - After all, the word “amazing” has the same meaning in the context of movie reviews as in the context of general language.
- Existing word embeddings are:
  - Google’s **Word2Vec** embeddings
  - Stanford’s **GloVe** embeddings
  - Facebook’s (Meta’s) **FastText** embeddings

Reusing Pretrained Models

- In 2018, Jeremy Howard and Sebastian Ruder showed the effectiveness of unsupervised pretraining for NLP tasks.
- Before **their paper** “Universal Language Model Fine-tuning for Text Classification”, using pretrained models was only common in computer vision.
  - How, reusing pretrained language models is the norm in NLP.

Classifier based On Pretrained Model

- We can very easily create a sentiment classifier based on a pretrained model.
  - We will use the **Universal Sentence Encoder** from TensorFlow Hub.
  - This model encodes sentences into an embedding space of dimension 512.
- We will then add two `Dense` layers on top of this model to create a sentiment classifier.

Classifier based On Pretrained Model

```
import os
import tensorflow_hub as hub

# Avoid downloading the model if it is already in the cache
os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
model = tf.keras.Sequential([
  # Note: trainable=True, thus the weights will be fine-tuned
  hub.KerasLayer("https://tfhub.dev/google/universal-sentence-
          encoder/4",
      trainable=True, dtype=tf.string, input_shape=[]),
  tf.keras.layers.Dense(64, activation="relu"),
  tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.fit(train_set, validation_data=valid_set, epochs=10)
```

Results of this Model

- After training, this model should reach a validation accuracy of over 90%.
  - This is already very good. Some reviews are inherently ambiguous.

Note on Tensorflow Hub (Extra)

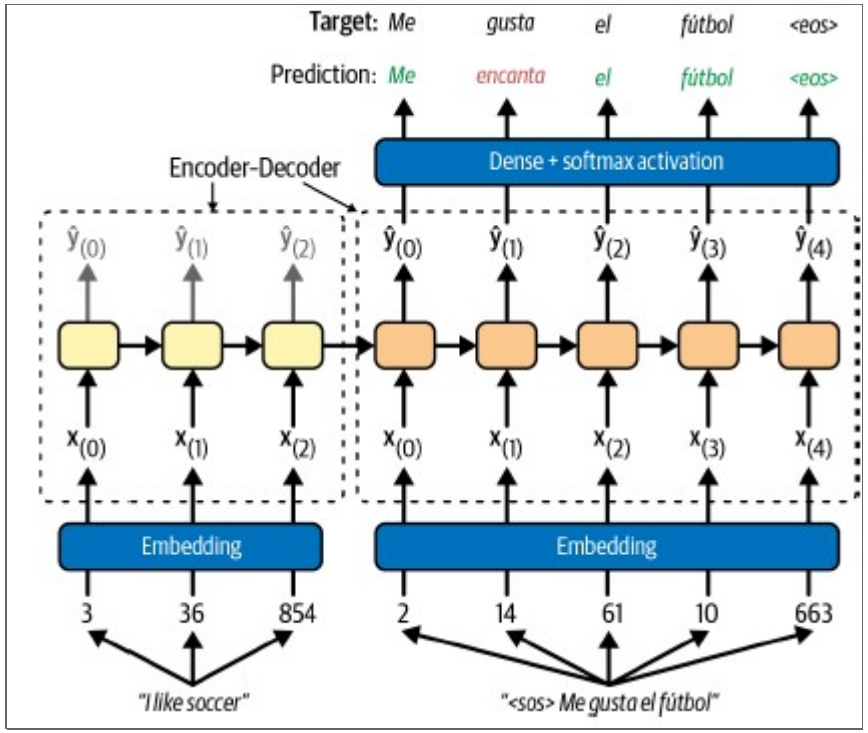
- Tensorflow Hub has been moved to **“Kaggle Models”**.
  - The link in the example should probably be changed later.

16.3 An Encoder-Decoder Network for Neural Machine Translation

The Task

- We will develop a model that will translate (short) English sentences into Spanish.

Model Overview



Encoder-Decoder model for machine translation

Model Overview

- English sentences are fed as inputs to the **encoder**.
- The **decoder** outputs the Spanish translations.
- During training, **teacher forcing** is used.
  - This means that the decoder is fed the correct Spanish translation, but shifted by one word. The first input is <sos> (start-of-sequence).
  - Stated otherwise, during training the decoder is fed the word it *should* have predicted, regardless of its actual prediction.
  - Teacher forcing significantly speeds up training.

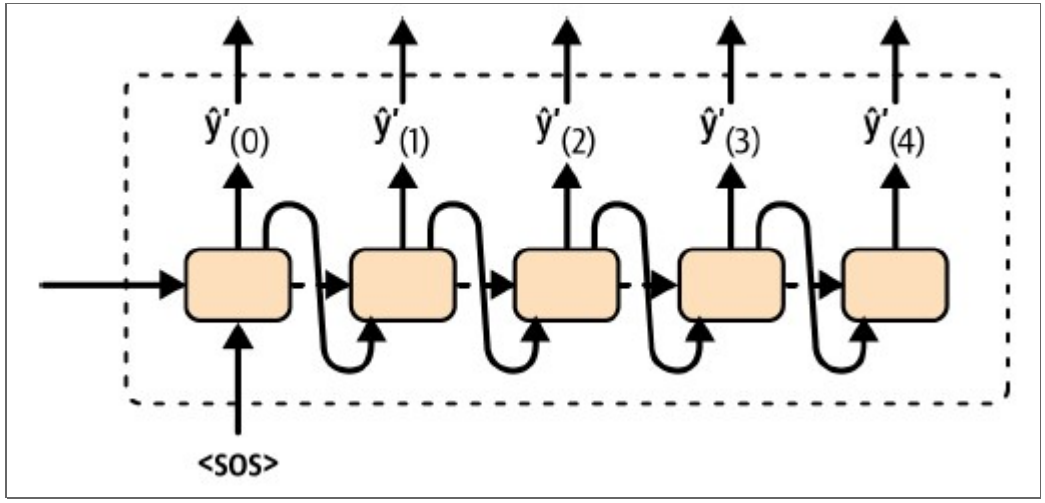
Embeddings and Outputs

- We use two embedding layers, one for English and one for Spanish, to embed the token IDs into an embedding space.
- As expected, at each step, the decoder outputs a probability distribution over all possible (Spanish) tokens (words).
  - We use the `softmax` activation function for this.
- Since this is a regular classification task, we use the `sparse_categorical_crossentropy` loss function.

Making Inferences

- During inference, i.e. when actually translating English sentences, we do *not* have the correct Spanish translation.
- Instead, at inference time, we will feed the decoder the word that it predicted at the previous time step. (I.e. we feed it the most likely word from the previous time step.)
  - Obviously, we will also need to look up the embedding of this word.

Making Inferences



Decoder inference

The Dataset

- The dataset is composed of about 120000 sentence pairs.

```
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)
# convert to Path, change name, add name of text file and read text
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text()
# text is now a (long) string with all the sentence pairs
```

The Dataset

- Remove Spanish characters “¡” and “¿” from the dataset as `TextVectorization` layer doesn’t handle them.
- English and Spanish sentences are separated by a tab.

```
import numpy as np

text = text.replace("¡", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
# pairs is a list of lists, each sublist has length 2 and contains an
# English sentence and a Spanish sentence
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs)  # separates the pairs into 2
          lists
```

`*` and `zip` (Extra)

- In the previous example, the `*` operator is used to unpack a list into its individual elements.
- An example of using the `*` operator to unpack values is given below.

```
def sum_of_3(a, b, c):
  return a + b + c

my_list = [1,2,3]
print(sum_of_3(my_list)) # Error !!
print(sum_of_3(*my_list)) # Will print 6
```

`*` and `zip` (Extra Ctd.)

- The `zip` function takes any number of iterables and returns an iterables of tuples.
- The first tuple contains the first element of each of the iterables.
- The second tuple contains the second element of each of the iterables.
- Etc.
- The number of tuples returned is equal to the length of the shortest iterable.

`*` and `zip` (Extra Ctd.)

```
list1 = [1,2,3]
list2 = [4,5,6,7] # Note length mismatch
for x, y in zip(list1, list2):
  print(x, y)
```

yields

```
1 4
2 5
3 6
```

`*` and `zip` (Extra Ctd.)

An example with four iterables, each of length 2.

```
lowercase, uppercase = zip(["a", "A"], ["b", "B"], ["c", "C"], ["d",
          "D"])
print(lowercase)
print(uppercase)
```

yields

```
('a', 'b', 'c', 'd')
('A', 'B', 'C', 'D')
```

`*` and `zip` (Extra Ctd.)

- An example illustrating what happens in `zip(*pairs)`

```
pairs = [["Eng1", "Spa1"], ["Eng2", "Spa2"], ["Eng3", "Spa3"]]
eng, spa = zip(*pairs)
print(eng)
print(spa)
```

yields

```
('Eng1', 'Eng2', 'Eng3')
('Spa1', 'Spa2', 'Spa3')
```

The Dataset

- Checking a couple of translations:

```
for i in range(3):
  print(sentences_en[i], "=>", sentences_es[i])
```

yields

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

`TextVectorization` Layers

- We create one `TextVectorization` layer per language.
  - We limit the vocabulary to 1000 tokens (which is small).
  - All sentences have a maximum length of 50, so we limit the length of the sequences to 50.
  - For the Spanish sentences, we add `startofseq` and `endofseq` tokens.

```
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])
```

Check Some Tokens

```
text_vec_layer_en.get_vocabulary()[:10]
```

yields

```
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
```

and

```
text_vec_layer_es.get_vocabulary()[:10]
```

gives

```
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom',
'la']
```

Training and Validation Sets

- We will use the first 100000 sentence pairs for training and the remaining sentence pairs for validation.

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
# Inputs for decoder are the Spanish sentences starting with
          `startofseq`
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:]])
# Targets for decoder are the Spanish sentences ending with `endofseq`
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:]])
```

Creating the Model

- An encoder-decoder model is *not* sequential, hence we will use the functional API.
- We start with the two text inputs, one for the encoder and one for the decoder:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

- Next, we encode the inputs using the `TextVectorization` layers we created earlier:

```
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
```

Creating the Model (Ctd.)

- Next, we create two embedding layers, and embed the inputs.
  - We set `mask_zero=True` for both embedding layers so that masking is handled automatically.
  - We choose an embedding size of 128, but this is a hyperparameter.

```
embed_size = 128
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```

Creating the Model (Ctd.)

- Next, we create the encoder, which is an LSTM.
  - We use `return_state=True` to get the final state of the LSTM as this will be used as the initial state of the decoder.
  - The LSTM layer returns two states separately: a short-term state and a long-term state. We gather these into a list using the `*` operator.

```
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
```

Creating the Model (Ctd.)

- Next, we create the decoder, which is also an LSTM.
  - We use `return_sequences=True` because we want to output a probability distribution at each time step.
  - We use `initial_state=encoder_state` to set the initial state of the decoder to the final state of the encoder.

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings,
          initial_state=encoder_state)
```

Creating the Model (Ctd.)

- We pass the `decoder_outputs` through a `Dense` layer with a softmax activation function to get the probability distribution over all possible Spanish words at each time step.

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

Creating the Model (Ctd.)

- Finally, create the model and compile and fit it.

```
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))
```

Translating Sentences

- Translating a sentence is not as simple as calling `model.predict()`.
  - This is because we need to feed the decoder the word it predicted at the previous time step.
- We implement a simple solution where we call the model multiple times.
  - We keep track of the translation so far and use that as input to the model.
  - We keep looping until the model predicts `endofseq`.

Translating Sentences

```
def translate(sentence_en):
  translation = ""
  for word_idx in range(max_length):
    X = np.array([sentence_en])  # encoder input
    X_dec = np.array(["startofseq " + translation])  # decoder input
    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
    translation += " " + predicted_word
  return translation.strip()
```

Example Translations

```
translate("I like soccer")
```

gives

```
'me gusta el fútbol'
```

while

```
translate("I like soccer and also going to the beach")
```

gives

```
'me gusta el fútbol y a veces mismo al bus'
```

which means “I like soccer and sometimes even the bus”.

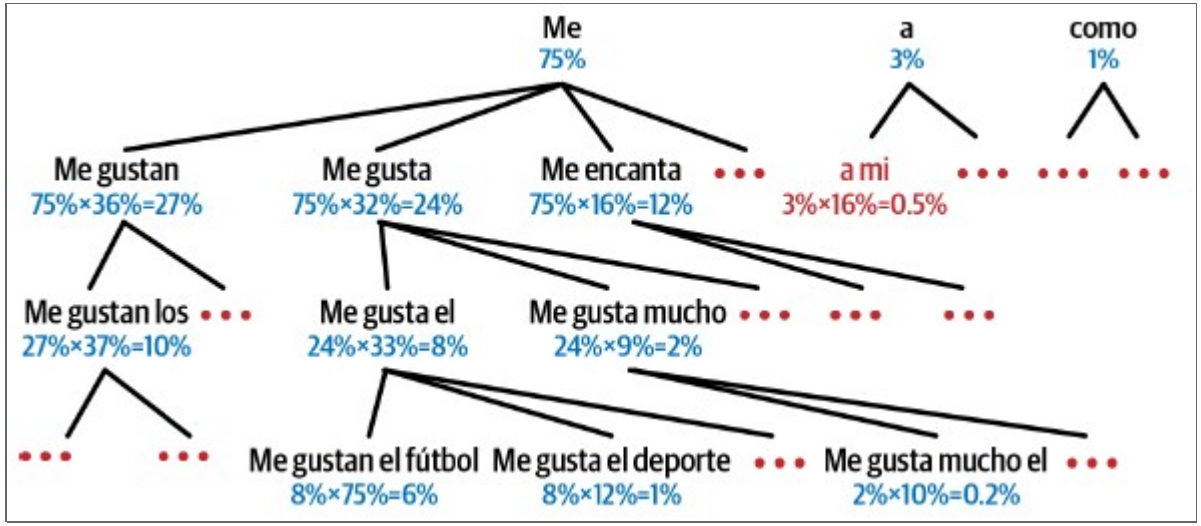
16.3.1 Bidirectional RNNs

16.3.2 Beam Search

Improving Translations without Additional Training

- The current `translate` method has no way of correcting a previous mistake.
  - Once a word has been output, it cannot be changed.
- **Beam search** keeps track of a short list of the k (the *beam width*) most promising sentences.
  - At each decoder step, it tries to extend each of these sentences, again only keeping the k most promising ones.

Example Beam Search



Beam search

Example Beam Search

- On the previous slide, the beam width is 3.
- We try to translate the sentence “I like soccer”
- At the first step, the decoder outputs 1000 probabilities.
  - We keep the 3 most likely words: “me” (75%), “a” (3%) and “como” (1%).
- In the second step, we use the model to predict the next word(s) for each of the 3 sentences.
  - E.g. for the sentence starting with “me”, we get “gustan” (36%), “gusta” (32%) and “encanta” (12%).
  - For the sentence starting with “a”, the most likely word is “mi” (16%), etc.

Example Beam Search

- The three most promising translations so far are
  - “Me gustan” (Probability $0.75 \times 0.36 = 0.27$)
  - “Me gusta” (Probability $0.75 \times 0.32 = 0.24$)
  - “Me encanta” (Probability $0.75 \times 0.16 = 0.12$)
- We use the model to complete each of these sentences. These are (in principle) 3000 possible completions.
  - We only keep the 3 most promising ones.

16.4 Attention Mechanisms

General Attention Idea

- In a “traditional” encoder-decoder architecture, the decoder only has access to the final state of the encoder.
- Now, we will allow the decoder to look at all the states of the encoder.
- These states need to be aggregated into a single vector.
- Be giving different weights to the different outputs at each timestep, the decoder can focus on different parts of the encoder outputs.
- This is called an **attention mechanism**.

Luong Attention

- There are different ways of computing the attention weights.
- **Luong attention** or **multiplicative** attention uses the dot product to measure the similarity between two vectors.
  - This is very similar to the way attention is computed in transformers and will be explained there in more detail.

16.4.1 Attention is All You Need: The Original Transformer Architecture

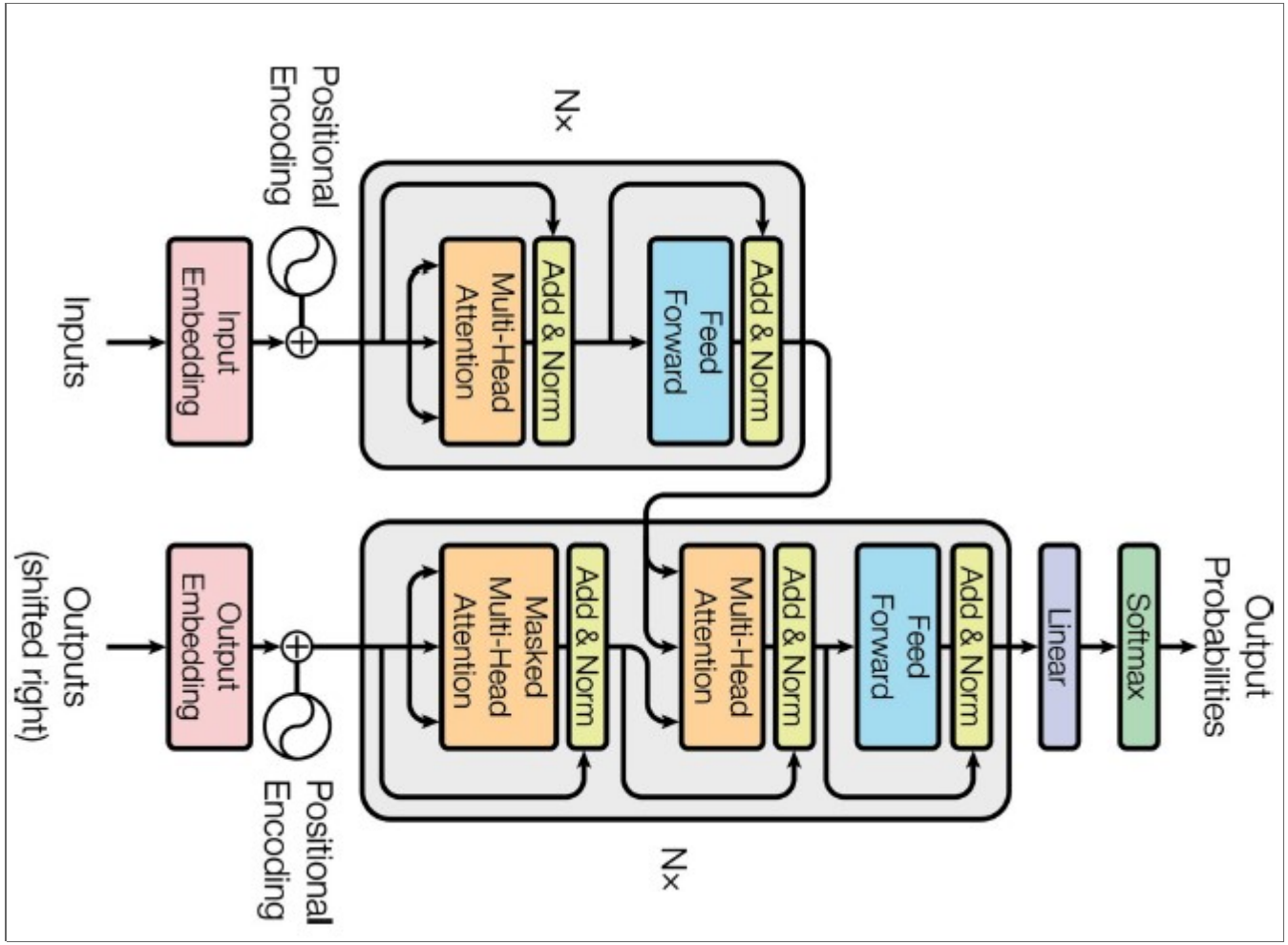
Attention is All You Need

- The **Attention is all you need** paper introduced the **Transformer** architecture.
- The transformer architecture does *not* use recurrent layers nor does it use convolutional layers. It uses
  - Attention mechanisms
  - Embedding layers
  - Dense layers
  - Normalization layers
  - Skip connections

The Transformer Architecture

- A transformer does **not** process the sequence elements sequentially. This makes it
  - easier to train
  - easier to parallelize (across multiple GPUs)
  - easier to capture long-range patterns

The Transformer Architecture



The original transformer architecture

The Transformer Architecture

- The original transformer was developed for language translation.
- It uses an **encoder** (left-hand side in the figure) and a **decoder** (right-hand side).
- The output of the embedding layer is a tensor of shape `(batch_size, sequence_length, embedding_size)`.
  - The tensors are gradually transformed as they flow through the transformer but their *shape remains the same*.

Encoder Role

- Suppose we use the transformer to translate English to Spanish.
- The inputs to the encoder are the English sentences.
- The role of the encoder is to compute a word representation for each word in the input sentence that captures the meaning of this word **in the context of the sentence**.

Decoder Role

- The decoder is used to generate the output sentence, i.e. the Spanish translation.
- Its role is to transform each word representation in the translated sentence into a word representation of the next word in the translated sentence.
- On top of the decoder sits a fully connected layer with a softmax activation function that outputs a probability distribution over all possible Spanish words.
  - After training the probability for the correct Spanish word is hopefully close to 1.

Encoder and Decoder Structure

- Both the encoder and the decoder are composed of a stack of **identical layers** that are repeated N times.
  - In the paper $N = 6$.
- All the elements in the sequence are treated in parallel.
  - Thus, the transformer actually doesn't have a notion of the "order" of the elements in the sequence.
  - That's why we use a **positional encoding**.

Positional Encoding

- A **positional encoding** is a dense vector that encodes the position of an element in the sequence.
- This can be implemented by using an `Embedding` layer by making it encode all the positions from 0 to the maximum sequence length.
  - Note: in the original paper, the authors use a fixed positional encoding based on sine and cosine functions.

Multi-Head Attention

- The **multi-head attention** layer is the most important layer in the transformer.
- First, let's consider attention with a *single* head. In the transformer, this is the same as Luong attention (dot product attention), except for a scaling factor.

Scaled Dot-Product Attention

The formula for scaled dot-product attention is

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

- $\mathbf{Q} \in \mathbb{R}^{n_q \times d_k}$, is the query matrix. It contains one row per query. Each query (and key) has dimension d_k.
- $\mathbf{K} \in \mathbb{R}^{n_k \times d_k}$, is the key matrix. It contains one row per key. Each key (and query) has dimension d_k.
- $\mathbf{V} \in \mathbb{R}^{n_k \times d_v}$, is the value matrix. It contains one row per value. Each value has dimension d_v.

Scaled Dot-Product Attention (Ctd.)

- $\mathbf{QK}^T$ is a matrix of shape (n_q, n_k). It contains the dot products between each query and each key.
  - A dot product is a measure of similarity.
  - These dot products are scaled down by $\sqrt{d_k}$ because this works better in practice.
- The softmax is applied to each row of this matrix.
  - The values in each row sum to 1, and hence say how much each value should be weighted.
- Finally, this matrix is multiplied by $\mathbf{V}$ to get the output matrix of shape (n_q, d_v).
  - Each row of the result is the weighted sum of the rows in $\mathbf{V}$, where the weights are given the similarity scores between the query and the keys.

`tf.keras.layers.Attention`

- The `tf.keras.layers.Attention` layer implements scaled dot-product attention.
  - If `use_scale=True` then it will learn how to scale the attention scores. (In the original paper, the authors use a fixed scaling factor.)
- Note: apart from the optional scale parameter, the `Attention` layer does not have any learnable parameters.

Example tf.keras.layers.Attention

```
tf.random.set_seed(42)
# Generate random query, key and value
n_q, n_k, d_k, d_v = 2, 4, 5, 3
query = tf.random.normal(shape=(1,n_q,d_k))
key = tf.random.normal(shape=(1, n_k, d_k ))
value = tf.random.normal(shape=(1, n_k, d_v))

att_layer = tf.keras.layers.Attention()
att_layer([query, value, key]) # note order
```

gives

```
<tf.Tensor: shape=(1, 2, 3), dtype=float32, numpy=
array([[[-0.7952401 ,  1.2932012 , -0.22712055],
        [-0.66913885, -0.13671604,  1.7900223 ]]], dtype=float32)>
```

`Attention` but “Manually”

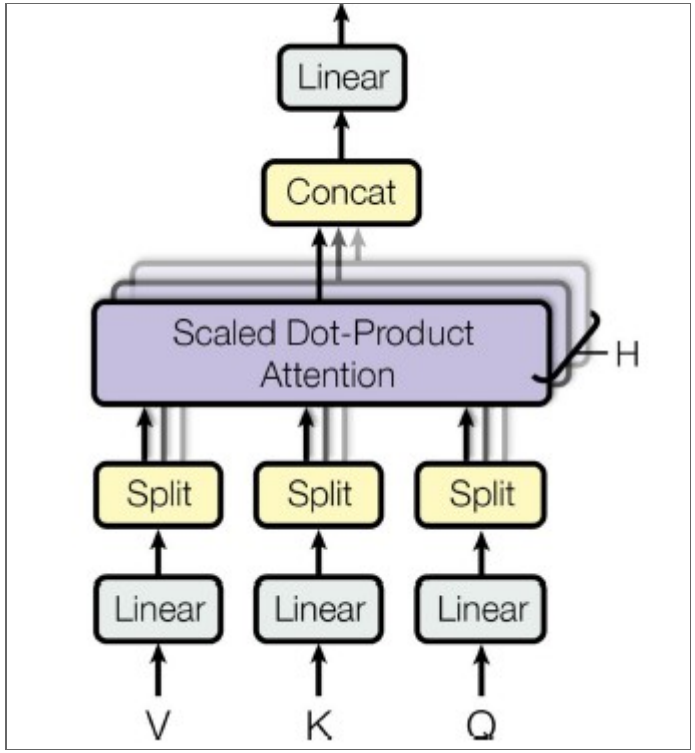
```
# Compute similarity scores between query and key:
similarity_scores = tf.math.softmax(tf.matmul(query, key,
          transpose_b=True))
# Take linear combination of values
tf.matmul(similarity_scores, value)
```

yields

```
<tf.Tensor: shape=(1, 2, 3), dtype=float32, numpy=
array([[[-0.7952401 ,  1.2932012 , -0.22712055],
        [-0.66913885, -0.13671604,  1.7900223 ]]], dtype=float32)>
```

We get the same result as with the `Attention` layer.

Multi-Head Attention



Multi-head attention

Multi-Head Attention

- We want to introduce learnable parameters in the attention mechanism, so that the model can learn which parts of the sequence to focus on.
- We want to give the model the ability to focus on different aspects of the words in the sequence.
  - E.g. “the current word is a verb”, “the current word is in the present tense”, ….

Multi-Head Attention

- Prior to calling the scaled dot-product attention, the query, key and value matrices are (conceptually) transformed using multiple linear transformations.
  - These are (time-distributed) `Dense` layers without an activation function.
- The number of linear transformations is called the **number of heads**.
  - Each linear transformation is applied to *all* elements in the sequence (in a time-distributed manner)
  - These linear tranformations have learnable parameters.
- The results of the scaled-dot product attentions are concatenated.
- Finally, there is a final linear transformation.

Multi-Head Attention (Extra)

- For simplicity of notation, we ignore the batch dimension in this example.
- Suppose the number of heads is 3.
- When $\mathbf{Q} \in \mathbb{R}^{n_q \times d_k}$, then the linear transformations for $\mathbf{Q}$ will compute 3 tensors $\mathbf{PQ}_1, \mathbf{PQ}_2, \mathbf{PQ}_3$ all in $\mathbb{R}^{n_q \times d'_k}$
- Likewise from $\mathbf{K} \in \mathbb{R}^{n_k \times d_k}$ we get $\mathbf{PK}_1, \mathbf{PK}_2, \mathbf{PK}_3$ all in $\mathbb{R}^{n_k \times d'_k}$
- And from $\mathbf{V} \in \mathbb{R}^{n_k \times d_v}$ we get $\mathbf{PV}_1, \mathbf{PV}_2, \mathbf{PV}_3$ all in $\mathbb{R}^{n_k \times d'_v}$

Multi-Head Attention (Extra)

- These tensors are passed separately to the scaled dot-product attention layer and we get three outputs $\mathbf{O}_1, \mathbf{O}_2, \mathbf{O}_3$ all in $\mathbb{R}^{n_q \times d'_v}$
- These outputs are concatenated, yielding a tensor $\mathbf{O}$ in $\mathbb{R}^{n_q \times 3d'_v}$
- Finally, a linear transformation brings this back to $\mathbb{R}^{n_q \times d_k}$ so that the output has the same shape as the query input.

Use of Multi-Head Attention

- In the original paper, the *encoder* uses multi-head attention in each of the blocks.
  - The queries, keys and values are all the same.
  - This is called **self-attention**.

Use of Multi-Head Attention

- In the original paper, the *decoder* uses two different types of attention.
- The first attention layer in the decoder is self-attention, i.e. the queries, keys and values are all the same.
  - However, when training, the decoder input is the complete output sequence shifted by one position.
  - If the decoder could look “into the future”, it would be cheating, and it wouldn’t learn anything.
  - This layer is called **masked self-attention** (see later)

Use of Multi-Head Attention

- The second attention layer in the decoder attends to the outputs of the encoder.
  - The queries come from the decoder.
  - The keys and values come from the encoder.
  - Here, the decoder is allowed to look at all the encoder outputs.
  - This is called **cross-attention**.

Masked Self-Attention (Causal Attention)

- In the decoder, we want to prevent the decoder from looking “into the future”.
- This can be done by making sure that the attention weights for tokens in the future are zero.
- Suppose, we have the following attention weights, prior to applying the softmax, i.e. $\mathbf{QK}^T$ (ignoring any possible scaling factor).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Masked Self-Attention (Causal Attention)

- Replacing the upper triangular part with $-\infty$:

$$\begin{bmatrix} a_{11} & -\infty & -\infty & -\infty \\ a_{21} & a_{22} & -\infty & -\infty \\ a_{31} & a_{32} & a_{33} & -\infty \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

  will, after applying the softmax, make the attention weights for the future tokens zero.

16.5 An Avalanche of Transformer Models

Many Transformer Models

- 2018 is called the “ImageNet moment for NLP”.
- A *lot* of progress has been made.

The GPT Paper

- The **GPT paper** came out in 2018.
- GPT stands for **Generative Pre-Training**.
- The GPT architecture ressembles the *decoder* of the original transformer.
  - It does not use cross-attention, only masked multi-head attention.
- It is trained in a **self-supervised** way.
  - The training objective is simply to predict the *next token*.

The GPT Paper

- Once this large stack of transformers is pretrained using next-token prediction, the model can be fine-tuned for a specific task.
- Some tasks include:
  - text classification
  - similarity
  - question answering

BERT

- BERT stands for **Bidirectional Encoder Representations from Transformers**.
- BERT's architecture is similar to the original transformer encoder.
  - It uses non-masked multi-head attention.
- BERT was trained using two pretraining tasks:
  - **Masked Language Modeling** (MLM)
  - **Next Sentence Predictions** (NSP)

Masked Language Modeling

- Each word in a sentence has a 15% probability of being masked.
  - The model is trained to predict the masked words.
- Example:
  - Original sentence: “She had fun at the birthday party.”
  - Input sentence: “She [MASK] fun at the [MASK] party.”
  - Task: predict the two masked words.
- More in detail, each selected word has
  - a 80% probability of being masked
  - a 10% probability of being replaced by a random word
  - a 10% probability of being left unchanged

Next Sentence Prediction

- The model is trained to predict whether two sentences are consecutive or not.
- E.g. “The dog sleeps” and “It snored loudly” are consectutive, while “The dog sleeps” and “The Earth orbits the Sun” are not.
- It turned out later that NSP was not as important as initially thought.
  - It was dropped in later architectures.

Other Papers

- The **GPT-2 paper**
  - Very similar to original GPT architecture, but even larger (1.5 billion parameters).
  - Can perform **zero shot learning**: can achieve good performance on many tasks without fine-tuning.
- Switch transformers (by Google researchers)
- ....

16.6 Vision Transformers

16.7 Hugging Face’s Transformers Library

Hugging Face

- **Hugging Face** is an AI-company thas has built a whole ecosystem of easy-to-use open-source tools for NLP, vision and more.
- The *Transformer Library* is the central component in their ecosystem.
  - Allows to easily download a pretrained model, and its corresponding tokenizer.
  - The library supports Tensorflow, PyTorch and more.

Example Usage

- The easiest way to perform a task is to use `transformers.pipeline()` function:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")  # many other tasks are
          available
result = classifier("The actors were very convincing.")
print(result)
```

yields

```
[{'label': 'POSITIVE', 'score': 0.9998071789741516}]
```

Example Usage showing Bias

```
classifier(["I am from India.", "I am from Iraq."]) # batch of sentences
```

gives

```
[{'label': 'POSITIVE', 'score': 0.9896161556243896},
 {'label': 'NEGATIVE', 'score': 0.9811071157455444}]
```

Models with Bias

Always think very carefully before deploying a model that has biases that could be harmful to certain groups of people!

Moving beyond the Default Model

- The example above used the “default” model for the `sentiment-analysis` task.
- You can manually specify a different model
- The code below uses a different model for the `text-classification` task.
  - The `text-classification` task classifies two sentences into three categories: “contradiction”, “entailment” and “neutral”.

```
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.")
```

yields

```
[{'label': 'contradiction', 'score': 0.9790192246437073}]
```

Models and Taks

Models and tasks can be found at https://huggingface.co/models and https://huggingface.co/tasks

Moving beyond `pipeline`

- The Transformers library provides many different classes:
  - tokenizers
  - models
  - configurations
  - callbacks
  - ....