Natural Language Processing Demystified | Transformers, Pre-training, and Transfer Learning

https://nlpdemystified.org
https://github.com/nitinpunjabi/nlp-demystified

Course module for this demo: https://www.nlpdemystified.org/course/transformers

IMPORTANT
Enable **GPU acceleration** by going to *Runtime > Change Runtime Type*. Keep in mind that, on certain tiers, you're not guaranteed GPU access depending on usage history and current load.

Also, if you're running this in the cloud rather than a local Jupyter server on your machine, then the notebook will *timeout* after a period of inactivity.

Refer to this link on how to run Colab notebooks locally on your machine to avoid this issue:
https://research.google.com/colaboratory/local-runtimes.html

```
!pip install BPEmb

import math
import numpy as np
import tensorflow as tf

from bpemb import BPEmb
```

Transformers From Scratch

We'll build a transformer from scratch, layer-by-layer. We'll start with the **Multi-Head Self-Attention** layer since that's the most involved bit. Once we have that working, the rest of the model will look familiar if you've been following the course so far.

Multi-Head Self-Attention

Scaled Dot Product Self-Attention

Inside each attention head is a **Scaled Dot Product Self-Attention** operation as we covered in the slides. Given *queries*, *keys*, and *values*, the operation returns a new "mix" of the values.

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The following function implements this and also takes a mask to account for padding and for masking future tokens for decoding (i.e. **look-ahead mask**). We'll cover masking later in the notebook.

```
def scaled_dot_product_attention(query, key, value, mask=None):
  key_dim = tf.cast(tf.shape(key)[-1], tf.float32)
  scaled_scores = tf.matmul(query, key, transpose_b=True) / np.sqrt(key_dim)

  if mask is not None:
    scaled_scores = tf.where(mask==0, -np.inf, scaled_scores)

  softmax = tf.keras.layers.Softmax()
  weights = softmax(scaled_scores)
  return tf.matmul(weights, value), weights
```

Suppose our *queries*, *keys*, and *values* are each a length of 3 with a dimension of 4.

```
seq_len = 3
embed_dim = 4

queries = np.random.rand(seq_len, embed_dim)
```

```
keys = np.random.rand(seq_len, embed_dim)
values = np.random.rand(seq_len, embed_dim)

print("Queries:\n", queries)
```

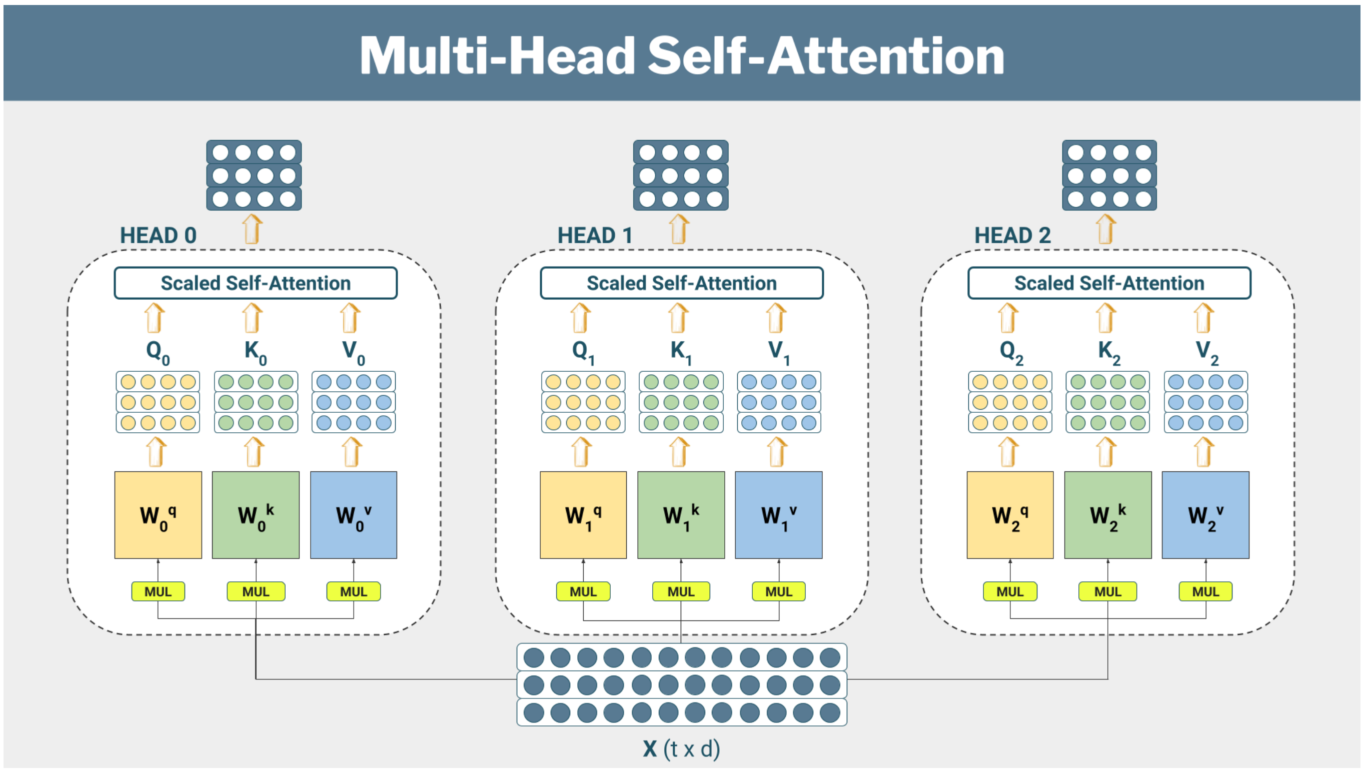
This would be the self-attention output and weights.

```
output, attn_weights = scaled_dot_product_attention(queries, keys, values)

print("Output\n", output, "\n")
print("Weights\n", attn_weights)
```

Generating queries, keys, and values for multiple heads.

Now that we have a way to calculate self-attention, let's actually generate the input *queries*, *keys*, and *values* for multiple heads.

In the slides (and in most references), each attention head had its own separate set of *query*, *key*, and *value* weights. Each weight matrix was of dimension $d \ x \ d/h$ where h was the number of heads.



It's easier to understand things this way and we can certainly code it this way as well. But we can also "simulate" different heads with a single query matrix, single key matrix, and single value matrix.

We'll do both. First we'll create *query*, *key*, and *value* vectors using separate weights per head.

In the slides, we used an example of 12 dimensional embeddings processed by three attentions heads, and we'll do the same here.

```
batch_size = 1
seq_len = 3
embed_dim = 12
num_heads = 3
head_dim = embed_dim // num_heads

print(f"Dimension of each head: {head_dim}")
```

Using separate weight matrices per head

Suppose these are our input embeddings. Here we have a batch of 1 containing a sequence of length 3, with each element being a 12-dimensional embedding.

```
x = np.random.rand(batch_size, seq_len, embed_dim).round(1)
print("Input shape: ", x.shape, "\n")
print("Input:\n", x)
```

We'll declare three sets of *query* weights (one for each head), three sets of *key* weights, and three sets of *value* weights. Remember each weight matrix should have a dimension of $\mathrm{d \times d/h}$.

```
# The query weights for each head.
wq0 = np.random.rand(embed_dim, head_dim).round(1)
wq1 = np.random.rand(embed_dim, head_dim).round(1)
wq2 = np.random.rand(embed_dim, head_dim).round(1)

# The key weights for each head.
wk0 = np.random.rand(embed_dim, head_dim).round(1)
wk1 = np.random.rand(embed_dim, head_dim).round(1)
wk2 = np.random.rand(embed_dim, head_dim).round(1)

# The value weights for each head.
wv0 = np.random.rand(embed_dim, head_dim).round(1)
wv1 = np.random.rand(embed_dim, head_dim).round(1)
wv2 = np.random.rand(embed_dim, head_dim).round(1)
```

```
print("The three sets of query weights (one for each head):")
print("wq0:\n", wq0)
print("wq1:\n", wq1)
print("wq2:\n", wq1)
```

We'll generate our *queries*, *keys*, and *values* for each head by multiplying our input by the weights.

```
# Geneated queries, keys, and values for the first head.
q0 = np.dot(x, wq0)
k0 = np.dot(x, wk0)
v0 = np.dot(x, wv0)

# Geneated queries, keys, and values for the second head.
q1 = np.dot(x, wq1)
k1 = np.dot(x, wk1)
v1 = np.dot(x, wv1)

# Geneated queries, keys, and values for the third head.
q2 = np.dot(x, wq2)
k2 = np.dot(x, wk2)
v2 = np.dot(x, wv2)
```

These are the resulting *query*, *key*, and *value* vectors for the first head.

```
print("Q, K, and V for first head:\n")

print(f"q0 {q0.shape}:\n", q0, "\n")
print(f"k0 {k0.shape}:\n", k0, "\n")
print(f"v0 {v0.shape}:\n", v0)
```

Now that we have our Q, K, V vectors, we can just pass them to our self-attention operation. Here we're calculating the output and attention weights for the first head.

```
out0, attn_weights0 = scaled_dot_product_attention(q0, k0, v0)

print("Output from first attention head: ", out0, "\n")
print("Attention weights from first head: ", attn_weights0)
```

Here are the other two (attention weights are ignored).

```
out1, _ = scaled_dot_product_attention(q1, k1, v1)
out2, _ = scaled_dot_product_attention(q2, k2, v2)

print("Output from second attention head: ", out1, "\n")
print("Output from third attention head: ", out2,)
```

As we covered in the slides, once we have each head's output, we concatenate them and then put them through a linear layer for further processing.

```
combined_out_a = np.concatenate((out0, out1, out2), axis=-1)
print(f"Combined output from all heads {combined_out_a.shape}:")
print(combined_out_a)

# The final step would be to run combined_out_a through a linear/dense layer
# for further processing.
```

So that's a complete run of **multi-head self-attention** using separate sets of weights per head.

Let's now get the same thing done using a single query weight matrix, single key weight matrix, and single value weight matrix.

These were our separate per-head query weights:

```
print("Query weights for first head: \n", wq0, "\n")
print("Query weights for second head: \n", wq1, "\n")
print("Query weights for third head: \n", wq2)
```

Suppose instead of declaring three separate query weight matrices, we had declared one. i.e. a single $d\ x\ d$ matrix. We're concatenating our per-head query weights here instead of declaring a new set of weights so that we get the same results.

```
wq = np.concatenate((wq0, wq1, wq2), axis=1)
print(f"Single query weight matrix {wq.shape}: \n", wq)
```

In the same vein, pretend we declared a single key weight matrix, and single value weight matrix.

```
wk = np.concatenate((wk0, wk1, wk2), axis=1)
wv = np.concatenate((wv0, wv1, wv2), axis=1)

print(f"Single key weight matrix {wk.shape}:\n", wk, "\n")
print(f"Single value weight matrix {wv.shape}:\n", wv)
```

Now we can calculate all our *queries*, *keys*, and *values* with three dot products.

```
q_s = np.dot(x, wq)
k_s = np.dot(x, wk)
v_s = np.dot(x, wv)
```

These are our resulting query vectors (we'll call them "combined queries"). How do we simulate different heads with this?

```
print(f"Query vectors using a single weight matrix {q_s.shape}:\n", q_s)
```

Somehow, we need to separate these vectors such they're treated like three separate sets by the self-attention operation.

```
print(q0, "\n")
print(q1, "\n")
print(q2)
```

Notice how each set of per-head queries looks like we took the combined queries, and chopped them vertically every four dimensions.

We can split our combined queries into $\mathrm{d\ x\ d/h}$ heads using **reshape** and **transpose**.

The first step is to *reshape* our combined queries from a shape of:
(batch_size, seq_len, embed_dim)

into a shape of
(batch_size, seq_len, num_heads, head_dim).

https://www.tensorflow.org/api_docs/python/tf/reshape

```
# Note: we can achieve the same thing by passing -1 instead of seq_len.
q_s_reshaped = tf.reshape(q_s, (batch_size, seq_len, num_heads, head_dim))
print(f"Combined queries: {q_s.shape}\n", q_s, "\n")
print(f"Reshaped into separate heads: {q_s_reshaped.shape}\n", q_s_reshaped)
```

At this point, we have our desired shape. The next step is to *transpose* it such that simulates vertically chopping our combined queries. By transposing, our matrix dimensions become:
(batch_size, num_heads, seq_len, head_dim)

https://www.tensorflow.org/api_docs/python/tf/transpose

```
q_s_transposed = tf.transpose(q_s_reshaped, perm=[0, 2, 1, 3]).numpy()
print(f"Queries transposed into \"separate\" heads {q_s_transposed.shape}:\n",
      q_s_transposed)
```

If we compare this against the separate per-head queries we calculated previously, we see the same result except we now have all our queries in a single matrix.

```
print("The separate per-head query matrices from before: ")
print(q0, "\n")
print(q1, "\n")
print(q2)
```

Let's do the exact same thing with our combined keys and values.

```
k_s_transposed = tf.transpose(tf.reshape(k_s, (batch_size, -1, num_heads, head_dim)), perm=[0, 2, 1, 3]).numpy()
v_s_transposed = tf.transpose(tf.reshape(v_s, (batch_size, -1, num_heads, head_dim)), perm=[0, 2, 1, 3]).numpy()

print(f"Keys for all heads in a single matrix {k_s.shape}: \n", k_s_transposed, "\n")
print(f"Values for all heads in a single matrix {v_s.shape}: \n", v_s_transposed)
```

Set up this way, we can now calculate the outputs from all attention heads with a single call to our self-attention operation.

```
all_heads_output, all_attn_weights = scaled_dot_product_attention(q_s_transposed,
                                                                  k_s_transposed,
                                                                  v_s_transposed)
print("Self attention output:\n", all_heads_output)
```

As a sanity check, we can compare this against the outputs from individual heads we calculated earlier:

```
print("Per head outputs from using separate sets of weights per head:")
print(out0, "\n")
print(out1, "\n")
print(out2)
```

To get the final concatenated result, we need to reverse our **reshape** and **transpose** operation, starting with the **transpose** this time.

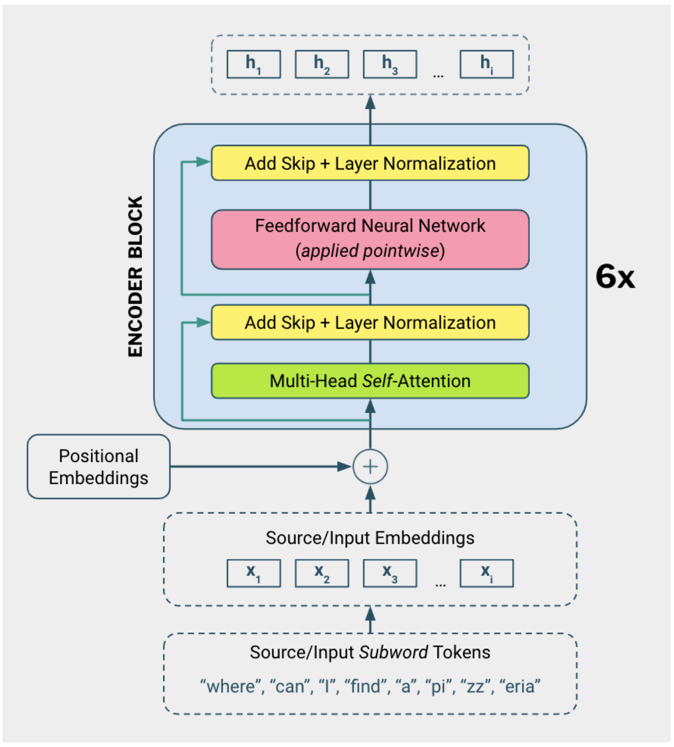
```
combined_out_b = tf.reshape(tf.transpose(all_heads_output, perm=[0, 2, 1, 3]),
                            shape=(batch_size, seq_len, embed_dim))
print("Final output from using single query, key, value matrices:\n",
      combined_out_b, "\n")
print("Final output from using separate query, key, value matrices per head:\n",
      combined_out_a)
```

We can encapsulate everything we just covered in a class.

```
class MultiHeadSelfAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadSelfAttention, self).__init__()
    self.d_model = d_model
    self.num_heads = num_heads

    self.d_head = self.d_model // self.num_heads

    self.wq = tf.keras.layers.Dense(self.d_model)
    self.wk = tf.keras.layers.Dense(self.d_model)
    self.wv = tf.keras.layers.Dense(self.d_model)

    # Linear layer to generate the final output.
    self.dense = tf.keras.layers.Dense(self.d_model)

  def split_heads(self, x):
    batch_size = x.shape[0]

    split_inputs = tf.reshape(x, (batch_size, -1, self.num_heads, self.d_head))
    return tf.transpose(split_inputs, perm=[0, 2, 1, 3])

  def merge_heads(self, x):
    batch_size = x.shape[0]
    merged_inputs = tf.transpose(x, perm=[0, 2, 1, 3])
    return tf.reshape(merged_inputs, (batch_size, -1, self.d_model))

  def call(self, q, k, v, mask):
    qs = self.wq(q)
    ks = self.wk(k)
    vs = self.wv(v)

    qs = self.split_heads(qs)
    ks = self.split_heads(ks)
    vs = self.split_heads(vs)

    output, attn_weights = scaled_dot_product_attention(qs, ks, vs, mask)
    output = self.merge_heads(output)

    return self.dense(output), attn_weights



mhsa = MultiHeadSelfAttention(12, 3)

output, attn_weights = mhsa(x, x, x, None)
print(f"MHSA output{output.shape}:")
print(output)
```

Encoder Block

We can now build our **Encoder Block**. In addition to the **Multi-Head Self Attention** layer, the **Encoder Block** also has **skip connections**, **layer normalization steps**, and a **two-layer feed-forward neural network**. The original **Attention Is All You Need** paper also included some **dropout** applied to the self-attention output which isn't shown in the illustration below (see references for a link to the paper).


h₁ h₂ h₃ ... hᵢ
Add Skip + Layer Normalization
Feedforward Neural Network (applied pointwise)
Add Skip + Layer Normalization
Multi-Head Self-Attention
ENCODER BLOCK
6x
Positional Embeddings
Source/Input Embeddings
x₁ x₂ x₃ ... xᵢ
Source/Input Subword Tokens
"where", "can", "I", "find", "a", "pi", "zz", "eria"

Since a two-layer feed forward neural network is used in multiple places in the transformer, here's a function which creates and returns one.

```
def feed_forward_network(d_model, hidden_dim):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(hidden_dim, activation='relu'),
      tf.keras.layers.Dense(d_model)
  ])
```

This is our encoder block containing all the layers and steps from the preceding illustration (plus dropout).

```
class EncoderBlock(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, hidden_dim, dropout_rate=0.1):
    super(EncoderBlock, self).__init__()

    self.mhsa = MultiHeadSelfAttention(d_model, num_heads)
    self.ffn = feed_forward_network(d_model, hidden_dim)

    self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
    self.dropout2 = tf.keras.layers.Dropout(dropout_rate)

    self.layernorm1 = tf.keras.layers.LayerNormalization()
    self.layernorm2 = tf.keras.layers.LayerNormalization()

  def call(self, x, training, mask):
    mhsa_output, attn_weights = self.mhsa(x, x, x, mask)
    mhsa_output = self.dropout1(mhsa_output, training=training)
    mhsa_output = self.layernorm1(x + mhsa_output)

    ffn_output = self.ffn(mhsa_output)
    ffn_output = self.dropout2(ffn_output, training=training)
    output = self.layernorm2(mhsa_output + ffn_output)

    return output, attn_weights
```

Suppose we have an embedding dimension of 12, and we want 3 attention heads and a feed forward network with a hidden dimension of 48 (4x the embedding dimension). We would declare and use a single encoder block like so:

```
encoder_block = EncoderBlock(12, 3, 48)

block_output,  _ = encoder_block(x, True, None)
print(f"Output from single encoder block {block_output.shape}:")
print(block_output)
```

Word and Positional Embeddings

Let's now deal with the actual input to the **initial** encoder block. The inputs are going to be *positional word embeddings*. That is, word embeddings with some positional information added to them.

Let's start with **subword** tokenization. For demonstration, we'll use a subword tokenizer called **BPEmb**. It uses **Byte-Pair Encoding** and supports over two hundred languages.

https://bpemb.h-its.org/

```
# Load the English tokenizer.
bpemb_en = BPEmb(lang="en")
```

The library comes with embeddings for a number of words.

```
bpemb_vocab_size, bpemb_embed_size = bpemb_en.vectors.shape
print("Vocabulary size:", bpemb_vocab_size)
print("Embedding size:", bpemb_embed_size)
```

```
# Embedding for the word "car".
bpemb_en.vectors[bpemb_en.words.index('car')]
```

We don't need the embeddings since we're going to use our own embedding layer. What we're interested in are the subword tokens and their respective ids. The ids will be used as indexes into our embedding layer.

If this doesn't sound familiar, refer to the module on word vectors:
https://www.nlpdemystified.org/course/word-vectors

These are the subword tokens for our example sentence from the slides. **BPEmb** places underscores in front of any tokens which are whole words or intended to begin words.

Remember that subword tokenizers are trained using count frequencies over a corpus. So these subword tokens are specific to **BPEmb**. Another subword tokenizer may output something different. This is why it's important that when we use a pretrained model, we make sure to use the pretrained model's tokenizer. We'll see this when we use pretrained transformers later in this module.

```
sample_sentence = "Where can I find a pizzeria?"
tokens = bpemb_en.encode(sample_sentence)
print(tokens)
```

We can retrieve each subword token's respective id using the *encode_ids* method.

```
token_seq = np.array(bpemb_en.encode_ids("Where can I find a pizzeria?"))
print(token_seq)
```

Now that we have a way to tokenize and vectorize sentences, we can declare and use an embedding layer with the same vocabulary size as **BPEmb** and a desired embedding size.

```
token_embed = tf.keras.layers.Embedding(bpemb_vocab_size, embed_dim)
token_embeddings = token_embed(token_seq)

# The untrained embeddings for our sample sentence.
print("Embeddings for: ", sample_sentence)
print(token_embeddings)
```

Next, we need to add *positional* information to each token embedding. As we covered in the slides, the original paper used sinusoidals but it's more common these days to just use another set of embeddings. We'll do the latter here.

Here, we're declaring an embedding layer with rows equalling a maximum sequence length and columns equalling our token embedding size. We then generate a vector of position ids.

```
max_seq_len = 256
pos_embed = tf.keras.layers.Embedding(max_seq_len, embed_dim)

# Generate ids for each position of the token sequence.
pos_idx = tf.range(len(token_seq))
print(pos_idx)
```

We'll use these position ids to index into the positional embedding layer.

```
# These are our positon embeddings.
position_embeddings = pos_embed(pos_idx)
print("Position embeddings for the input sequence\n", position_embeddings)
```

The final step is to add our token and position embeddings. The result will be the input to the first encoder block.

```
input = token_embeddings + position_embeddings
print("Input to the initial encoder block:\n", input)
```

Encoder

Now that we have an encoder block and a way to embed our tokens with position information, we can create the **encoder** itself.

Given a batch of vectorized sequences, the encoder creates positional embeddings, runs them through its encoder blocks, and returns contextualized tokens.

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, num_blocks, d_model, num_heads, hidden_dim, src_vocab_size,
               max_seq_len, dropout_rate=0.1):
    super(Encoder, self).__init__()

    self.d_model = d_model
    self.max_seq_len = max_seq_len

    self.token_embed = tf.keras.layers.Embedding(src_vocab_size, self.d_model)
    self.pos_embed = tf.keras.layers.Embedding(max_seq_len, self.d_model)

    # The original Attention Is All You Need paper applied dropout to the
    # input before feeding it to the first encoder block.
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

    # Create encoder blocks.
    self.blocks = [EncoderBlock(self.d_model, num_heads, hidden_dim, dropout_rate)
    for _ in range(num_blocks)]

  def call(self, input, training, mask):
    token_embeds = self.token_embed(input)

    # Generate position indices for a batch of input sequences.
    num_pos = input.shape[0] * self.max_seq_len
    pos_idx = np.resize(np.arange(self.max_seq_len), num_pos)
    pos_idx = np.reshape(pos_idx, input.shape)
    pos_embeds = self.pos_embed(pos_idx)

    x = self.dropout(token_embeds + pos_embeds, training=training)

    # Run input through successive encoder blocks.
    for block in self.blocks:
      x, weights = block(x, training, mask)

    return x, weights
```

If you're wondering about this code block here:

```
num_pos = input.shape[0] * self.max_seq_len
pos_idx = np.resize(np.arange(self.max_seq_len), num_pos)
pos_idx = np.reshape(pos_idx, input.shape)
pos_embeds = self.pos_embed(pos_idx)
```

This generates positional embeddings for a *batch* of input sequences. Suppose this was our batch of input sequences to the encoder.

```
# Batch of 3 sequences, each of length 10 (10 is also the
# maximum sequence length in this case).
seqs = np.random.randint(0, 10000, size=(3, 10))
print(seqs.shape)
print(seqs)
```

We need to retrieve a positional embedding for every element in this batch. The first step is to create the respective positional ids...

```
pos_ids = np.resize(np.arange(seqs.shape[1]), seqs.shape[0] * seqs.shape[1])
print(pos_ids)
```

...and then reshape them to match the input batch dimensions.

```
pos_ids = np.reshape(pos_ids, (3, 10))
print(pos_ids.shape)
print(pos_ids)
```

We can now retrieve position embeddings for every token embedding.

```
pos_embed(pos_ids)
```

Let's try our encoder on a batch of sentences.

```
input_batch = [
    "Where can I find a pizzeria?",
    "Mass hysteria over listeria.",
    "I ain't no circle back girl."
]

bpemb_en.encode(input_batch)
```

```
input_seqs = bpemb_en.encode_ids(input_batch)
print("Vectorized inputs:")
input_seqs
```

Note how the input sequences aren't the same length in this batch. In this case, we need to pad them out so that they are. If you're unfamiliar with why, refer to the notebook on Recurrent Neural Networks:
https://colab.research.google.com/github/nitinpunjabi/nlp-demystified/blob/main/notebooks/nlpdemystified_recurrent_neural_networks.ipynb

We'll do this using *pad_sequences*.
https://www.tensorflow.org/api_docs/python/tf/keras/utils/pad_sequences

```
padded_input_seqs = tf.keras.preprocessing.sequence.pad_sequences(input_seqs, padding="post")
print("Input to the encoder:")
print(padded_input_seqs.shape)
print(padded_input_seqs)
```

Since our input now has padding, now's a good time to cover **masking**.

So given a mask, wherever there's a mask position set to 0, the corresponding position in the attention scores will be set to *-inf*. The resulting attention weight for the position will then be zero and no attending will occur for that position.

In the slides, we covered *look-ahead* masks for the decoder to prevent it from attending to future tokens, but we also need masks for padding.

In total, there are three masks involved:

1. The *encoder mask* to mask out any padding in the encoder sequences.
2. The *decoder mask* which is used in the decoder's **first** multi-head self-attention layer. It's a combination of two masks: one to account for the padding in target sequences, and the look-ahead mask.
3. The *memory mask* which is used in the decoder's **second** multi-head self-attention layer. The keys and values for this layer are going to be the encoder's output, and this mask will ensure the decoder doesn't attend to any encoder output which corresponds to padding. In practice, 1 and 3 are often the same.

The *scaled_dot_product_attention* function has this line:

```
if mask is not None:
  scaled_scores = tf.where(mask==0, -np.inf, scaled_scores)
```

Let's create an encoder mask for our batch of input sequences.

Wherever there's padding, we want the mask position set to zero.

```
enc_mask = tf.cast(tf.math.not_equal(padded_input_seqs, 0), tf.float32)
print("Input:")
print(padded_input_seqs, '\n')
print("Encoder mask:")
print(enc_mask)
```

Keep in mind that the dimension of the attention matrix (for this example) is going to be:
(batch size, number of heads, query size, key size)
(3, 3, 10, 10)

So we need to expand the mask dimensions like so:

```
enc_mask = enc_mask[:, tf.newaxis, tf.newaxis, :]
enc_mask
```

This way, the encoder mask will now be *broadcasted*.
https://www.tensorflow.org/xla/broadcasting

Now we can declare an encoder and pass it batches of vectorized sequences.

```
num_encoder_blocks = 6

# d_model is the embedding dimension used throughout.
d_model = 12

num_heads = 3

# Feed-forward network hidden dimension width.
ffn_hidden_dim = 48

src_vocab_size = bpemb_vocab_size
max_input_seq_len = padded_input_seqs.shape[1]

encoder = Encoder(
    num_encoder_blocks,
    d_model,
    num_heads,
    ffn_hidden_dim,
    src_vocab_size,
    max_input_seq_len)
```

We can now pass our input sequences and mask to the encoder.

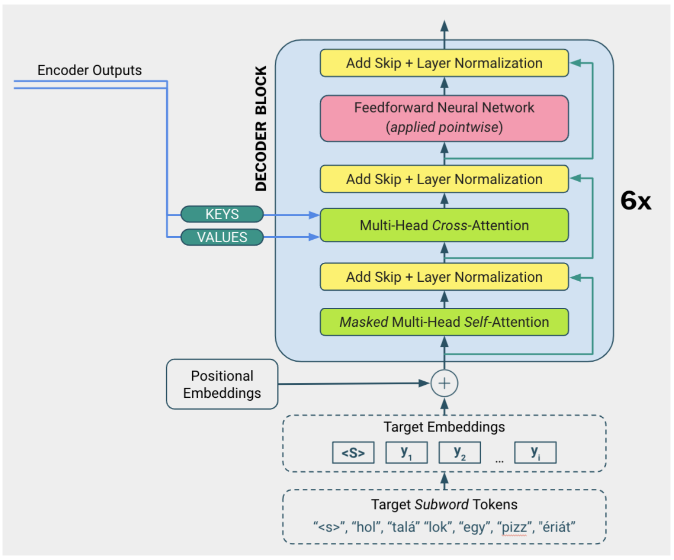
```
encoder_output, attn_weights = encoder(padded_input_seqs, training=True,
                                       mask=enc_mask)
```

```
print(f"Encoder output {encoder_output.shape}:")
print(encoder_output)
```

Decoder Block

Let's build the **Decoder Block**. Everything we did to create the **encoder** block applies here. The major differences are that the **Decoder Block** has:

1. a **Multi-Head Cross-Attention** layer which uses the encoder's outputs as the keys and values.
2. an extra skip/residual connection along with an extra layer normalization step.


Encoder Outputs
DECODER BLOCK
Add Skip + Layer Normalization
Feedforward Neural Network (applied pointwise)
Add Skip + Layer Normalization
6x
KEYS
VALUES
Multi-Head Cross-Attention
Add Skip + Layer Normalization
Masked Multi-Head Self-Attention
Positional Embeddings
Target Embeddings
<S> y1 y2 ... yi
Target Subword Tokens
"<s>", "hol", "talá" "lok", "egy", "pizz", "ériát"

```
class DecoderBlock(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, hidden_dim, dropout_rate=0.1):
    super(DecoderBlock, self).__init__()

    self.mhsa1 = MultiHeadSelfAttention(d_model, num_heads)
    self.mhsa2 = MultiHeadSelfAttention(d_model, num_heads)

    self.ffn = feed_forward_network(d_model, hidden_dim)

    self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
    self.dropout2 = tf.keras.layers.Dropout(dropout_rate)
    self.dropout3 = tf.keras.layers.Dropout(dropout_rate)

    self.layernorm1 = tf.keras.layers.LayerNormalization()
    self.layernorm2 = tf.keras.layers.LayerNormalization()
    self.layernorm3 = tf.keras.layers.LayerNormalization()

  # Note the decoder block takes two masks. One for the first MHSA, another
  # for the second MHSA.
  def call(self, encoder_output, target, training, decoder_mask, memory_mask):
    mhsa_output1, attn_weights = self.mhsa1(target, target, target, decoder_mask)
    mhsa_output1 = self.dropout1(mhsa_output1, training=training)
    mhsa_output1 = self.layernorm1(mhsa_output1 + target)

    mhsa_output2, attn_weights = self.mhsa2(mhsa_output1, encoder_output,
                                            encoder_output,
                                            memory_mask)
    mhsa_output2 = self.dropout2(mhsa_output2, training=training)
    mhsa_output2 = self.layernorm2(mhsa_output2 + mhsa_output1)

    ffn_output = self.ffn(mhsa_output2)
    ffn_output = self.dropout3(ffn_output, training=training)
    output = self.layernorm3(ffn_output + mhsa_output2)

    return output, attn_weights
```

Decoder

The decoder is almost the same as the encoder except it takes the encoder's output as part of its input, and it takes two masks: the decoder mask and memory mask.

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_blocks, d_model, num_heads, hidden_dim, target_vocab_size,
               max_seq_len, dropout_rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.max_seq_len = max_seq_len

    self.token_embed = tf.keras.layers.Embedding(target_vocab_size, self.d_model)
    self.pos_embed = tf.keras.layers.Embedding(max_seq_len, self.d_model)

    self.dropout = tf.keras.layers.Dropout(dropout_rate)

    self.blocks = [DecoderBlock(self.d_model, num_heads, hidden_dim, dropout_rate) for _ in range(num_blocks)]

  def call(self, encoder_output, target, training, decoder_mask, memory_mask):
    token_embeds = self.token_embed(target)

    # Generate position indices.
    num_pos = target.shape[0] * self.max_seq_len
    pos_idx = np.resize(np.arange(self.max_seq_len), num_pos)
    pos_idx = np.reshape(pos_idx, target.shape)

    pos_embeds = self.pos_embed(pos_idx)

    x = self.dropout(token_embeds + pos_embeds, training=training)

    for block in self.blocks:
      x, weights = block(encoder_output, x, training, decoder_mask, memory_mask)

    return x, weights
```

Before we try the decoder, let's cover the masks involved. The decoder takes two masks:

The *decoder mask* which is a combination of two masks: one to account for the padding in target sequences, and the look-ahead mask. This mask is used in the decoder's **first** multi-head self-attention layer.

The *memory mask* which is used in the decoder's **second** multi-head self-attention. The keys and values for this layer are going to be the encoder's output, and this mask will ensure the decoder doesn't attend to any encoder output which corresponds to padding.

Suppose this is our batch of vectorized target *input* sequences for the decoder. These values are just made up.

Note: If you need a refresher on how to prepare target input and output sequences for the decoder, refer to the seq2seq notebook.

```
# Made up values.
target_input_seqs = [
    [1, 652, 723, 123, 62],
    [1, 25,  98, 129, 248, 215, 359, 249],
    [1, 2369, 1259, 125, 486],
]
```

As we did with the encoder input sequences, we need to pad out this batch so that all sequences within it are the same length.

```
padded_target_input_seqs = tf.keras.preprocessing.sequence.pad_sequences(target_input_seqs, padding="post")
print("Padded target inputs to the decoder:")
print(padded_target_input_seqs.shape)
print(padded_target_input_seqs)
```

We can create the padding mask the same way we did for the encoder.

```
dec_padding_mask = tf.cast(tf.math.not_equal(padded_target_input_seqs, 0), tf.float32)
dec_padding_mask = dec_padding_mask[:, tf.newaxis, tf.newaxis, :]
print(dec_padding_mask)
```

As we covered in the slides, the look-ahead mask is a diagonal where the lower half are 1s and the upper half are zeros. This is easy to create using the *band_part* method:
https://www.tensorflow.org/api_docs/python/tf/linalg/band_part

```
target_input_seq_len = padded_target_input_seqs.shape[1]
look_ahead_mask = tf.linalg.band_part(tf.ones((target_input_seq_len,
                                               target_input_seq_len)), -1, 0)
print(look_ahead_mask)
```

To create the decoder mask, we just need to combine the padding and look-ahead masks. Note how the columns of the resulting decoder mask are all zero for padding positions.

```
dec_mask = tf.minimum(dec_padding_mask, look_ahead_mask)
print("The decoder mask:")
print(dec_mask)
```

We can now declare a decoder and pass it everything it needs. In our case, the *memory* mask is the same as the *encoder* mask.

```
decoder = Decoder(6, 12, 3, 48, 10000, 8)
decoder_output, _ = decoder(encoder_output, padded_target_input_seqs,
                            True, dec_mask, enc_mask)
print(f"Decoder output {decoder_output.shape}:")
print(decoder_output)
```

Transformer

We now have all the pieces to build the **Transformer** itself, and it's pretty simple.

```
class Transformer(tf.keras.Model):
  def __init__(self, num_blocks, d_model, num_heads, hidden_dim, source_vocab_size,
               target_vocab_size, max_input_len, max_target_len, dropout_rate=0.1):
    super(Transformer, self).__init__()

    self.encoder = Encoder(num_blocks, d_model, num_heads, hidden_dim, source_vocab_size,
                           max_input_len, dropout_rate)

    self.decoder = Decoder(num_blocks, d_model, num_heads, hidden_dim, target_vocab_size,
                           max_target_len, dropout_rate)

    # The final dense layer to generate logits from the decoder output.
    self.output_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, input_seqs, target_input_seqs, training, encoder_mask,
           decoder_mask, memory_mask):
    encoder_output, encoder_attn_weights = self.encoder(input_seqs,
                                                        training, encoder_mask)

    decoder_output, decoder_attn_weights = self.decoder(encoder_output,
                                                        target_input_seqs, training,
                                                        decoder_mask, memory_mask)

    return self.output_layer(decoder_output), encoder_attn_weights, decoder_attn_weights
```

```
transformer = Transformer(
    num_blocks = 6,
    d_model = 12,
    num_heads = 3,
    hidden_dim = 48,
    source_vocab_size = bpemb_vocab_size,
    target_vocab_size = 7000, # made-up target vocab size.
    max_input_len = padded_input_seqs.shape[1],
    max_target_len = padded_target_input_seqs.shape[1])

transformer_output, _, _ = transformer(padded_input_seqs,
                                       padded_target_input_seqs, True,
                                       enc_mask, dec_mask, memory_mask=enc_mask)
print(f"Transformer output {transformer_output.shape}:")
print(transformer_output) # If training, we would use this output to calculate losses.
```

That's the whole original transformer from scratch. From here, if you want to train this transformer, you can use the same approach we used when we built the translation model with attention in the seq2seq notebook. Remember to use a learning rate warmup (Refer to the paper for more information on this).

It's useful to know how these models work under the hood, but to train our own transformer to get impressive results is expensive. Both in terms of compute and data.

Fortunately, there's a zoo of **pretrained** transformer models we can use. We'll explore that next.

Pre-Training and Transfer Learning with Hugging Face and OpenAI

IMPORTANT
Enable **GPU acceleration** by going to *Runtime > Change Runtime Type*. Keep in mind that, on certain tiers, you're not guaranteed GPU access depending on usage history and current load.

Also, if you're running this in the cloud rather than a local Jupyter server on your machine, then the notebook will *timeout* after a period of inactivity.

Refer to this link on how to run Colab notebooks locally on your machine to avoid this issue:
https://research.google.com/colaboratory/local-runtimes.html

We'll explore pre-training and transfer learning using the **Transformers** library from Hugging Face. **Transformers** is an API and toolkit to download pre-trained models and further train them as needed.

We'll start with the **pipelines** module which abstracts a lot of operations such as tokenization, vectorization, inference, etc.

With **Transformers pipelines**, we can just feed text input and get text output. And there are **pipelines** for common tasks including classification, NER, summarization, etc.
https://huggingface.co/docs/transformers/index
https://huggingface.co/docs/transformers/main/en/main_classes/pipelines#pipelines

To get started, we'll need to install **Transformers**.

```
!pip install transformers
!pip install datasets
```

```
import operator
import pandas as pd
import tensorflow as tf
import transformers

from datasets import load_dataset
from tensorflow import keras
from transformers import AutoTokenizer
from transformers import pipeline
from transformers import TFAutoModelForQuestionAnswering
```

Getting up and running quickly with Hugging Face Pipelines

We'll use the **pipeline** (note the singular) abstraction which wraps all the other pipelines. Put simply, it'll be our interface to doing a bunch of NLP tasks.

Using the **pipeline** abstraction is easy. We can instantiate a pipeline with a particular task, and it'll automatically download a suitable tokenizer and model behind the scenes for us and take care of the input and output operations.
https://huggingface.co/docs/transformers/main/en/main_classes/pipelines#transformers.pipeline

Here, we're retrieving a pipeline for text-classification.

```
classifier = pipeline("text-classification")
```

Note the warning message about how no model was supplied. When we instantiate a pipeline for a task without specifying a particular model to perform the task, **Transformers** uses a default model. This is good enough for prototyping but for production, we'll want to specify which model to use for the task since the default can change. We'll see how to do this further below.

We can use the pipeline immediately to classify some text. Tokenization, vectorization, etc is taken care of behind the scenes.

```
classifier("Alice was excited to go the island but it didn't live up to the hype.")
```

```
classifier("Bob doesn't do well in group situations but he said it wasn't bad.")
```

There's support for summarization...

```
summarizer = pipeline("summarization")
```

```
text = """
Hans Niemann is launching a counterattack in his dispute with chess world
champion Magnus Carlsen, filing a federal lawsuit that accuses Carlsen of
maliciously colluding with others to defame the 19-year-old grandmaster and
ruin his career.

It's the latest move in a scandal that has injected unprecedented levels of
drama into the world of elite chess since early September, when Carlsen
suggested Niemann's upset victory over him at the Sinquefield Cup tournament
in St. Louis was the result of cheating.

Niemann wants a federal court in Missouri's eastern district to award him at
least $100 million in damages. Defendants in the lawsuit include Carlsen, his
company Play Magnus Group, the online platform Chess.com and its leader, Danny
Rensch, along with grandmaster Hikaru Nakamura.
"""
```

```
summarizer(text)
```

...and question answering (extractive in this example).

```
qa = pipeline("question-answering")
```

```
context="""
Hugging Face was founded in 2016 by Clément Delangue, Julien Chaumond, and
Thomas Wolf originally as a company that developed a chatbot app targeted at
teenagers.[2] After open-sourcing the model behind the chatbot, the company
pivoted to focus on being a platform for democratizing machine learning. In March
2021, Hugging Face raised $40 million in a Series B funding round.
"""

question = "Who are the Hugging Face founders?"

qa(question=question, context=context)
```

Extractive question-answering models work fine for certain domains, document structures, and questions. But situations that require reasoning, more complex parsing, or contain ambiguity can trip it up.

```
question = "What does Hugging Face do?"
qa(question=question, context=context)
```

There are ready-made pipelines for a number of tasks:
https://huggingface.co/docs/transformers/main/en/quicktour#pipeline

Let's say we want a pipeline that uses a particular model. On the Hugging Face model hub, you'll find both pre-trained models (e.g. BERT) *and* pre-trained models that have been fine-tuned for all sorts of tasks (e.g. BERT for text classification). These models are contributed by Hugging Face, other companies, institutions, and individuals. You can (and are encouraged) to train or fine-tune a model and upload it for others to use.
https://huggingface.co/models

For example, here's a collection of pre-trained models that have been tuned for text classification.
https://huggingface.co/models?pipeline_tag=text-classification&sort=downloads

This particular one is a pre-trained *Roberta-base* model that's been fine-tuned on Twitter data for sentiment analysis:
https://huggingface.co/cardiffnlp/twitter-roberta-base-sentiment

Note how you can try the model directly on the model page.

Let's say we want to download and use a particular model. For example, this *BERT-base* model fine-tuned for NER:
https://huggingface.co/dslim/bert-base-NER

We just need to pass the model path during pipeline instantiation.

```
ner = pipeline(model="dslim/bert-base-NER")
```

```
text = "Panic ensues in Redmond as love child of Microsoft and OpenAI declares humanity obsolete."
ner(text)
```

The **Transformers** library provides a bunch of helper classes to help with training models. And beyond the model hub, Hugging Face also hosts datasets, provides *spaces* where you can host your app, and offers a bunch of services such as cloud hardware and inference endpoints to help deploy your model.
Datasets: https://huggingface.co/datasets
Spaces: https://huggingface.co/spaces

With Hugging Face, you can build an ML app prototype within minutes and iterate quickly from there.
https://huggingface.co/docs

Learn more about how to build with Hugging Face through their free course and fantastic book:
https://huggingface.co/course
https://www.oreilly.com/library/view/natural-language-processing/9781098136789/

Fine-Tuning a Pre-Trained Model.

Let's say the model hub doesn't have a model that exactly suits your purpose. Perhaps you work in a particular domain and need to fine-tune a model using your own dataset.

In this section, we'll walk through how to download a pre-trained model and fine-tune it. Our example covers extractive question answering but it's the same idea with other tasks.

We'll fine-tune using a dataset from the **Datasets** hub.
https://huggingface.co/datasets

Hugging Face provides a **datasets** library to download and interact with the datasets. It's similar to the Tensorflow Dataset library we used in that it can hold data and provides a bunch of methods to preprocess that data.
https://huggingface.co/docs/datasets/ndex

The **Datasets** hub holds a bunch of question answering datasets.
https://huggingface.co/datasets?task_categories=task_categories:question-answering&sort=downloads

They differ based on data source, domain, and level of challenge. Since we're in a constrained environment (Colab free tier) and just learning how to fine-tune, we'll use SQuAD, a famous dataset comprised of crowd-sourced questions on a set of Wikipedia articles, and where the answer is a span of text in the article.
https://huggingface.co/datasets/squad

```
data = load_dataset("squad")
```

The **datasets** library downloads and automatically splits the data into train and validation sets. It returns a dictionary of **Dataset** objects:
https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.DatasetDict
https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset

A **Dataset** object wraps an Apache Arrow table and provides a bunch of helper functions on top of it.
https://arrow.apache.org/

```
data
```

Glaning at the data, we see every context (Wikipedia passage) is used multiple times. i.e., there are multiple questions and answers for each context.

Every answer is a span of text from the context and the character position where the answer starts in the context is given.

```
pd.DataFrame(data['train'][0, 1, 2, 100, 101, 102],
             columns=["context", "question", "answers"])
```

Here's what we need to do:

1. Choose a pre-trained model based on what we want to accomplish and our constraints.
2. Download the appropriate tokenizer for the pre-trained model.
3. Tokenize and vectorize our dataset.

4. Mark where each answer starts and ends in our vectorized dataset.
5. Download the pre-trained model.
6. Fine-tune the pre-trained model with the vectorized dataset.

Given the free tier of Colab doesn't have a lot of GPU memory and that we're just trying to fine-tune a simple, extractive question answering model, we'll use *distilroberta-base*.
https://huggingface.co/distilroberta-base

Recall from the slides that *DistilBERT* was created using a technique called *knowledge distillation*. The result is a model that performs almost as well as BERT but is 40% smaller and 60% faster.
DistilBert Paper: https://arxiv.org/abs/1910.01108
https://en.wikipedia.org/wiki/Knowledge_distillation

distilroberta-base was created by applying knowledge distillation to *Roberta-Base*, a more powerful model than BERT.
Roberta paper: https://arxiv.org/abs/1907.11692

The **Transformers** library provides a set of Auto Classes that can automatically retrieve configurations, tokenizers, and models based on a path or a name. We'll use the **AutoTokenizer** class to get the right tokenizer for *distilroberta-base*.
https://huggingface.co/docs/transformers/main/en/model_doc/auto
https://huggingface.co/docs/transformers/main/en/model_doc/auto#transformers.AutoTokenizer

```
model_name = 'distilroberta-base'
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Calling *encode* converts a string to a sequence of integer token ids.
https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils#transformers.PreTrainedTokenizerBase.encode

```
t = "Where can I find a pizzeria?"
print(tokenizer.encode(t))
```

But to tokenize, we call the tokenizer object directly (i.e. using __*call*__).
https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils#transformers.PreTrainedTokenizerBase.__call__

This returns a sequence of ids and an attention mask in a **BatchEncoding** object:
https://huggingface.co/docs/transformers/main/en/glossary#input-ids
https://huggingface.co/docs/transformers/main/en/main_classes/tokenizer#transformers.BatchEncoding

Since there's no padding on this sample string, the mask is all 1s.

```
encoded_t = tokenizer(t)
print(encoded_t)
```

We can convert the ids back to tokens using *convert_ids_to_tokens*.
https://huggingface.co/docs/transformers/main/en/main_classes/tokenizer#transformers.PreTrainedTokenizer.convert_ids_to_tokens

Note how the tokenizer added a start of sequence token (<s>), end of sequence token (</s>), and how it uses Ġ to signal a word has preceding whitespace. Keep in mind that what you're seeing here is the output from the *distilroberta-base* tokenizer. Other tokenizers may work differently.

```
print(tokenizer.convert_ids_to_tokens(encoded_t['input_ids']))
```

As we covered in the slides, for question answering, we need to encode the question and context as a pair. In our case, we can do that by passing in both strings separated by a comma.

```
encoded_pair = tokenizer("this is a question", "this is the context")
print(encoded_pair)
```

The *distilroberta-base* tokenizer uses a double </s></s> as a separator.

```
print(tokenizer.convert_ids_to_tokens(encoded_pair['input_ids']))
```

Side note:
Most of the tokenizers in the **Transformers** library come in two versions: a Python implementation and a faster Rust implementation. When

available, **Autotokenizer** will download the fast version.
https://huggingface.co/docs/transformers/main_classes/tokenizer

We can check whether we have a fast tokenizer.

```
assert isinstance(tokenizer, transformers.PreTrainedTokenizerFast)
```

Suppose we tokenize this question/context pair...

```
context = "Sarah went to The Mirthless Cafe last night to meet her friend."
question = "Where did Sarah go?"

# The answer span and the answer's starting character position in the context.
answer = "The Mirthless Cafe"
answer_start = 14
```

```
x = tokenizer(question, context)
x
```

Note how the word *Mirthless* gets tokenized into subwords. For legibility, we're using *batch_decode* to convert the input_ids to strings.
https://huggingface.co/docs/transformers/v4.23.1/en/internal/tokenization_utils#transformers.PreTrainedTokenizerBase.batch_decode

```
tokenizer.batch_decode(x['input_ids'])
```

When we tokenize our dataset, there will probably be question/context pairs which exceed our model's maximum sequence length. In *Roberta*'s case, that's 512. Available GPU memory may make us further reduce the maximum sequence length of our input.

Let's say the maximum sequence length we can handle is 15, so we truncate the context.

```
example_max_length = 15
x = tokenizer(question, context, max_length=example_max_length,
              truncation="only_second")
x
```

The problem here is that the answer span gets chopped off by truncation. In other situations, the answer may not be included at all.

```
tokenizer.batch_decode(x['input_ids'])
```

To ensure we tokenize all context tokens while respecting a maximum length, we can set *return_overflowing_tokens* to **True**. The end effect is to split the input into multiple question/context sequences, with each context sequence being a continuation of the previous one. Since the last one may be shorter than the max length, we set the right padding length as well.

What we get back are multiple *input_id* sequences.

```
x = tokenizer(question, context, max_length=example_max_length,
              truncation="only_second", return_overflowing_tokens=True,
              padding="max_length")
x
```

```
len(x['input_ids'])
```

Looking at the decoded sequences, we see the entire context is included across three sequences (along with padding on the last one).

```
tokenizer.batch_decode(x['input_ids'])
```

Note a few things from the encoded object *x*:

- The last *attention_mask* sequence has 0s to signify padding.
- The *overflow_to_sample_mapping* array tells us which question/context pair each *input_ids* sequence comes from. In our example, we tokenized a single question/context pair which resulted in three *input_ids* sequences, so *overflow_to_sample_mapping* is 3 0s.

If we tokenize two question/context pairs, we'll see the *overflow_to_sample_mapping* reflect that.

```
tokenizer(['question 1', 'question 2'],
          ['context 1', 'context 2'],
          return_overflowing_tokens=True)
```

But there's still a problem here in that none of the sequences contain the full answer ("The Mirthless Cafe"). Right now, the correct full answer is split across sequences.

To counter this, we can tokenize our question/context pair into overlapping sequences by setting a *stride* length. We did something similar when we prepared the dataset for our character-level language model.

```
stride = 5
x = tokenizer(question, context, max_length=example_max_length,
              truncation="only_second", return_overflowing_tokens=True,
              stride=stride, padding="max_length")
```

By setting a stride of 5, each context sequence starts 5 subwords back from the previous sequence.

This way, two of our tokenized sequences now contain the full answer.

```
tokenizer.batch_decode(x['input_ids'])
```

We now have a way to tokenize our question/context pairs.

Our tokenizer returned this **BatchEncoding** object:

```
print(x.keys(), '\n')
x
```

To fine-tune a model for question answering, our pre-trained *distilroberta-base* model expects this object to contain two more pieces of information:

- *start_positions*: the token positions where answers begin.
- *end_positions*: the token positions where answers end.

https://huggingface.co/docs/transformers/main/en/model_doc/roberta#transformers.RobertaForQuestionAnswering.forward

All we have in our example (and the SQuAD dataset) is the position of the starting character of the answer.

```
print(answer_start)
print(context[answer_start:answer_start+len(answer)])
```

We need to use this to locate the token positions where each answer starts and ends in every *input_ids* sequence. In some cases, the complete answer may not be in a particular sequence. We need to handle those cases as well.

To do this, we'll get more information by setting *return_offsets_mapping* to **True** in the tokenizer.

```
x = tokenizer(question, context, max_length=example_max_length,
              truncation="only_second", return_overflowing_tokens=True,
              stride=stride, return_offsets_mapping=True,
              padding="max_length")
x
```

This results in *offset_mapping* sequences, one for each *input_ids* sequence.

```
print(len(x['input_ids']))
print(len(x['offset_mapping']))
```

Each entry in an *offset_mapping* tells us the starting and ending character position of each token in the original string. An offset mapping of (0,0) represents a special token (e.g. <s>).

For example, here's the first *input_ids* sequence along with its respective *offset_mapping*.

```
print(x['input_ids'][0])
print(x['offset_mapping'][0])
```

If we convert the first non-special input id to a token, and use the first non-special offset_mapping to extract a span from the question string, we get a match.

```
print("First non-special input_id converted to token:")
print(tokenizer.convert_ids_to_tokens(x['input_ids'][0][1]), "\n")

offset = x['offset_mapping'][0][1]
```

```
print(f"Span extracted from context using corresponding offset_mapping {offset}:")
print(question[offset[0]:offset[1]])
```

Since we know the character position of where the answer starts, we can use that and *offset_mapping* to get the start and ending token positions of the answer span.

The only remaining issue is identifying whether an offset is for a question or a context. Looking at the first two *offset_mappings*, note that:

1. In the first sequence, both the question and context *offset_mappings* start from zero.
2. In the second sequence, the context *offset_mapping* values carry on from the previous sequence (after accounting in the stride).

```
print(x['offset_mapping'][0])
print(x['offset_mapping'][1])
```

This means we need to identify

1. which *offset_mappings* belong to a context.
2. whether a particular sequence contains the answer at all.

The first can be done using the *sequence_ids* method on the encoding object. Each *input_ids* sequence has a corresponding *sequence_ids* list which tells us whether a token is part of a question, part of a context, or a special token.
https://huggingface.co/docs/transformers/main/en/main_classes/tokenizer#transformers.BatchEncoding.sequence_ids

```
print(x['input_ids'][0])
print(x.sequence_ids(0))
```

So to identify whether a token is part a context, we can use *sequence_ids* to check whether a token position maps to 1.

For the second issue, we can check whether the answer start and end character positions are within the lowest and highest offset mapping values respectively.

```
# We can calculate the answer end character position using the answer length.
answer_end = answer_start + len(answer)

print("Answer start character position:", answer_start)
print("Answer end character position:", answer_end)
print("Answer pulled from context:", context[answer_start:answer_end])
```

Let's find the start and end token positions from our collection of sequences. The full answer is not in the first sequence, but is in the third sequence. So let's experiment with those.

```
tokenizer.batch_decode(x['input_ids'])
```

First get all the information we need for the first sequence.

```
input_ids = x['input_ids'][0]
offset_mapping = x['offset_mapping'][0]
seq_ids = x.sequence_ids(0)
```

Determine where the context tokens start and end in the sequence.

```
# These are the sequence ids
print("Sequence IDs: ", seq_ids)


# Get the start index position (i.e. the first occurrence of 1).
context_pos_start = seq_ids.index(1)


# Utility function to find the *last* occurrence of a sequence.
def rindex(lst, value):
    return len(lst) - operator.indexOf(reversed(lst), value) - 1

# Get the end index position (i.e. the last occurrence of 1).
context_pos_end = rindex(seq_ids, 1)


print("Context tokens begin at position", context_pos_start)
print("Context tokens end at position", context_pos_end)
```

Now that we know which tokens are part of the context, we can look at their corresponding offset mappings to check whether the start and end character positions are within the offsets.

```
# These are the corresponding offsets.
context_offsets = offset_mapping[context_pos_start:context_pos_end+1]
print(context_offsets)
```

```
print("Is the lowest offset value lower than or equal to the starting character position?")
print("Answer starting character position:", answer_start)
print("First offset:", context_offsets[0])

# Note how we're checking the first tuple value.
print(context_offsets[0][0] <= answer_start)
```

```
print("Is the highest offset value higher than or equal to the ending character position?")
print("Answer ending character position:", answer_end)
print("Last offset:", context_offsets[-1])

# Note how how we're checking the second tuple value.
print(context_offsets[-1][1] >= answer_end)
```

So the first sequence contains a part of the answer but the full answer gets truncated. This matches a visual inspection:

```
print(tokenizer.batch_decode(input_ids))
```

Let's now do the same with the third sequence.

```
input_ids = x['input_ids'][2]
offset_mapping = x['offset_mapping'][2]
seq_ids = x.sequence_ids(2)

context_pos_start = seq_ids.index(1)
context_pos_end = rindex(seq_ids, 1)

context_offsets = offset_mapping[context_pos_start:context_pos_end+1]

print("Is the lowest offset value lower than or equal to the starting character position?")
print("Answer starting character position:", answer_start)
print("First offset:", context_offsets[0])

# Note how we're checking the first tuple value.
print(context_offsets[0][0] <= answer_start)

print("Is the highest offset value higher than or equal to the ending character position?")
print("Answer ending character position:", answer_end)
print("Last offset:", context_offsets[-1])

# Note how how we're checking the second tuple value.
print(context_offsets[-1][1] >= answer_end)
```

Now that we've confirmed the third sequence contains the full answer, we need to identify where the answer starts and ends in the *input_ids*. We can do this by scanning the offset_mapping from the left to find the start, and from the right to find the end.

```
s = e = 0

# Start scanning the offset_mapping from the
# left to find the token position where the answer starts.
# It's not guaranteed a tokenizer will output a token where the
# starting character matches the first answer character. When
# this happens, we take the previous token's position as our start.
i = context_pos_start
while offset_mapping[i][0] < answer_start:
  i += 1
if offset_mapping[i][0] == answer_start:
  s = i
else:
  s = i - 1

# Same idea when finding the ending token position.
j = context_pos_end
while offset_mapping[j][1] > answer_end:
  j -= 1
if offset_mapping[j][1] == answer_end:
  e = j
else:
  e = j + 1


print("Answer start token position in context:", s)
print("Answer end token position in context:", e)


print("Answer lifted from context:")
tokenizer.batch_decode(input_ids[s:e+1])
```

All the logic we stepped through so far is encapsulated in the following method. We'll use this to process our dataset.

```
def prepare_dataset(examples):
  # Some tokenizers don't strip spaces. If there happens to be question text
  # with excessive spaces, the context may not get encoded at all.
  examples["question"] = [q.lstrip() for q in examples["question"]]
  examples["context"] = [c.lstrip() for c in examples["context"]]

  # Tokenize.
  tokenized_examples = tokenizer(
      examples['question'],
      examples['context'],
      truncation="only_second",
      max_length = max_length,
      stride=stride,
      return_overflowing_tokens=True,
      return_offsets_mapping=True,
      padding="max_length"
  )

  # We'll collect a list of starting positions and ending positions.
  tokenized_examples['start_positions'] = []
  tokenized_examples['end_positions'] = []

  # Work through every sequence.
  for seq_idx in range(len(tokenized_examples['input_ids'])):
    seq_ids = tokenized_examples.sequence_ids(seq_idx)
    offset_mappings = tokenized_examples['offset_mapping'][seq_idx]

    cur_example_idx = tokenized_examples['overflow_to_sample_mapping'][seq_idx]
    answer = examples['answers'][cur_example_idx]
    answer_text = answer['text'][0]
    answer_start = answer['answer_start'][0]
    answer_end = answer_start + len(answer_text)

    context_pos_start = seq_ids.index(1)
    context_pos_end = rindex(seq_ids, 1)

    s = e = 0
    if (offset_mappings[context_pos_start][0] <= answer_start and
        offset_mappings[context_pos_end][1] >= answer_end):
      i = context_pos_start
      while offset_mappings[i][0] < answer_start:
        i += 1
      if offset_mappings[i][0] == answer_start:
        s = i
      else:
        s = i - 1

      j = context_pos_end
      while offset_mappings[j][1] > answer_end:
        j -= 1
      if offset_mappings[j][1] == answer_end:
        e = j
      else:
        e = j + 1

    tokenized_examples['start_positions'].append(s)
    tokenized_examples['end_positions'].append(e)

  return tokenized_examples
```

Before we process, we'll set maximum sequence length, stride, and batch size values.

I arrived at these values through experimentation. Even though *distilroberta-base* has a maximum sequence length of 512, using the full capacity (or a large batch value) results in an out-of-memory error while the attention scores are being calculated. This is on Colab's free tier. On the premium tier, you can use larger sequence lengths or batch values.

The nature of the data will also influence the values.

```
max_length = 400
stride = 100
batch_size = 32
```

We can map over the **Dataset** objects and apply our prepare method to the examples in batches.
https://huggingface.co/docs/datasets/main/en/nlp_process
https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset.map

remove_columns removes the original data columns and leaves only the post-tokenization columns in place. We can also parallelize processing by using the *num_proc* parameter.
https://huggingface.co/docs/datasets/main/en/process#multiprocessing

```
tokenized_datasets = data.map(
  prepare_dataset,
  batched=True,
  remove_columns=data["train"].column_names,
  num_proc=2,
)
```

Our tokenized dataset still contains two entries (*offset_mapping* and *overflow_to_sample_mapping*) our model won't expect, so we'll remove them.
https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset.remove_columns

```
data = tokenized_datasets.remove_columns(["offset_mapping",
                                          "overflow_to_sample_mapping"])
```

The last preparation step is to convert the Hugging Face **Dataset** objects into a Tensorflow-compatible datasets.
https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset.to_tf_dataset
https://huggingface.co/docs/datasets/main/en/use_with_tensorflow#when-to-use-totfdataset

```
train_set = data['train'].to_tf_dataset(batch_size=batch_size)
validation_set = data['validation'].to_tf_dataset(batch_size=batch_size)
```

We can now download a pre-trained model for fine-tuning. Just like we did with the tokenizer, we'll use an Auto Class to download the right model. In this case, we're using **TFAutoModelForQuestionAnswering**. This will download a Tensorflow implementation of the pre-trained model with a question answering head on it.

The head in this case is a dense layer that returns *start_logits* and *end_logits*. We can take the argmax of each to determine the start and end of the answer span (see model code for details).
https://huggingface.co/docs/transformers/main/en/model_doc/auto#transformers.TFAutoModelForQuestionAnswering
https://github.com/huggingface/transformers/blob/main/src/transformers/models/roberta/modeling_tf_roberta.py#L1629

```
model = TFAutoModelForQuestionAnswering.from_pretrained(model_name)
```

The following method attempts to answer a question given a context. It tokenizes the question and context, runs it through the model, takes the argmax of the start and end logits, and uses the result to extract an answer span from the context.

```
def get_answer(tokenizer, model, question, context):
  inputs = tokenizer([question], [context], return_tensors="np")
  outputs = model(inputs)
  start_position = tf.argmax(outputs.start_logits, axis=1)
  end_position = tf.argmax(outputs.end_logits, axis=1)
  answer = inputs["input_ids"][0, int(start_position) : int(end_position) + 1]
  return tokenizer.decode(answer).strip()
```

While the model body (the pre-trained *distilroberta-base* model) is trained, the head is not. So if we try to use our model to answer a question, it should fail or perform poorly (your output will differ because of different initial head weight values).

```
c = "Sarah went to The Mirthless Cafe last night to meet her friend."
q = "Where did Sarah go?"
get_answer(tokenizer, model, q, c)
```

```
# https://www.tensorflow.org/guide/mixed_precision
keras.mixed_precision.set_global_policy("mixed_float16")
```

```
# Use a learning rate recommended by the BERT authors.
# https://github.com/google-research/bert
model.compile(optimizer=keras.optimizers.Adam(learning_rate=3e-5))
```

We'll now fine-tune the model. Note that we didn't freeze the layers of the pre-trained body, so its weights will be tuned along with the head's weights.

Because the body is already pre-trained, we don't need a lot of epochs. 2-4 is typically enough (BERT authors recommend 4). Here, we're using 1 to demonstrate the power of pre-training.

Note: If you have GPU enabled and you're using Colab's free tier, the training time can be all over the place depending on which GPU you get assigned (anywhere from 20 minutes to an hour).

```
model.fit(train_set, validation_data=validation_set, epochs=1)
```

After completing our fine-tuning, we should now have a decent extractive question answering model.

```
c = "Sarah went to The Mirthless Cafe last night to meet her friend."
q = "Where did Sarah go?"
get_answer(tokenizer, model, q, c)
```

```
q = "Who did Sarah meet?"
get_answer(tokenizer, model, q, c)
```

```
q = "When did Sarah meet her friend?"
get_answer(tokenizer, model, q, c)
```

```
q = "Who went to the restaurant?"
get_answer(tokenizer, model, q, c)
```

But as we saw earlier, extractive question answering has its limits.

```
# Asking a logic teaser question is difficult despite the
# answer being available. To be fair, there is ambiguity here.
q = "Who did Sarah's friend meet?"
get_answer(tokenizer, model, q, c)
```

```
# The model can't determine when a question can't be
# answered. Some question answering datasets explicitly
# train for this.
q = "How did Sarah get to the restaurant?"
get_answer(tokenizer, model, q, c)
```

```
# The model isn't generative, either.
q = "What is a possible reason for why Sarah met her friend?"
get_answer(tokenizer, model, q, c)
```

But despite this model's limitations, I hope this shows the power of pre-training and how fast you can get something cool and useful up and running.

I encourage you to make an account on Hugging Face and push your model to the hub. Learn how here: https://huggingface.co/docs/transformers/model_sharing

We're just scratching the surface of question answering. Indeed you could dedicate an entire career to it. Areas to explore:

- Right now, we have to supply the context along with the question. A more sophisticated system would load all relevant documents into some database and search over it for an appropriate context/passage and then extract the answer from it. If multiple answers are extracted, then maybe some ranking system can be included as well.
- Another enhancement is extracting answers from different kinds of data. Beyond text, there are images, audio, graphs, tables, charts, etc.
- Abstractive answering involves composing answers (possibly multiple lines) rather than extracting them. Open book means having the system search for the answer first in a database, then composing an answer based on what it's found. Closed book means the model relies on its internal knowledge only. This is what a large language model like GPT-3 would do.

Speaking of GPT-3, let's play with a few more prompts. At this point, we'll switch to the OpenAI Playground and try out the prompts below. Check out the module video for commentary.

Before you can try out these prompts yourself, you'll need to open an account an OpenAI. At the time of this recording, OpenAI was providing a few dollars of credit to get started. It's more than enough to run the prompts below.

GPT-3 prompts to try out:

He said hello
She said bonjour
He said goodbye
She said

My name is Kilgore. I run a revenge-for-hire business called Delightful Reprisals. I billed my customer the wrong amount. Write an apology email to the customer and sign it with my name and company. Include a reason blaming Elon Musk.

He brought three pies to the office. He gave one to his co-workers, and threw one at his boss' face. How many pies did he have left?

Answer in the style of Jeopardy. He was the 26th President of the United States.

Sarah went to The Mirthless Cafe last night to meet her friend.

Print each answer to the following questions on separate lines.

Where did Sarah go?
Who did Sarah meet?
When did Sarah meet her friend?
How did Sarah get to the restaurant?
What's a possible reason why they met?

What did Marie Antoinette say about avocado toast?

What's a word that rhymes with money?

What's a word that rhymes with Kafkaesque?

Write a haiku about The Terminator.

Write a limerick about The Terminator.

There are open-source generative decoder models available on Hugging Face as well. Among them are:

- GPT-2
- EleutherAI/gpt-neo-1.3B
- bigscience/bloom-560m

Be on the lookout for GPT-4!

Further Exploration

OpenAI API docs to learn how to build products using their models:
https://openai.com/api/
https://beta.openai.com/docs/introduction

A catalog of transformer models:
https://amatriain.net/blog/transformer-models-an-introduction-and-catalog-2d1e9039f376/

Wordpiece and Sentencepiece:
https://huggingface.co/course/chapter6/6?fw=pt
https://github.com/google/sentencepiece

Papers
Attention Is All You Need (original Transformer paper): https://arxiv.org/abs/1706.03762

The Annotated Transformer: http://nlp.seas.harvard.edu/annotated-transformer/

GPT-3: https://arxiv.org/abs/2005.14165

BERT: https://arxiv.org/abs/1810.04805

RoBERTa paper: https://arxiv.org/abs/1907.11692

ALBERT paper: https://arxiv.org/abs/1909.11942

DistilBert paper: https://arxiv.org/abs/1910.01108

Electra paper: https://arxiv.org/abs/2003.10555

XLM: https://arxiv.org/abs/1901.07291