Transformers

By Felix Dallinger

Note: I used this notebook as a code example: https://colab.research.google.com/github/tensorflow/text/blob/master/docs/tutorials/transformer.ipynb and these 3 youtube videos, to understand how a transformer works:

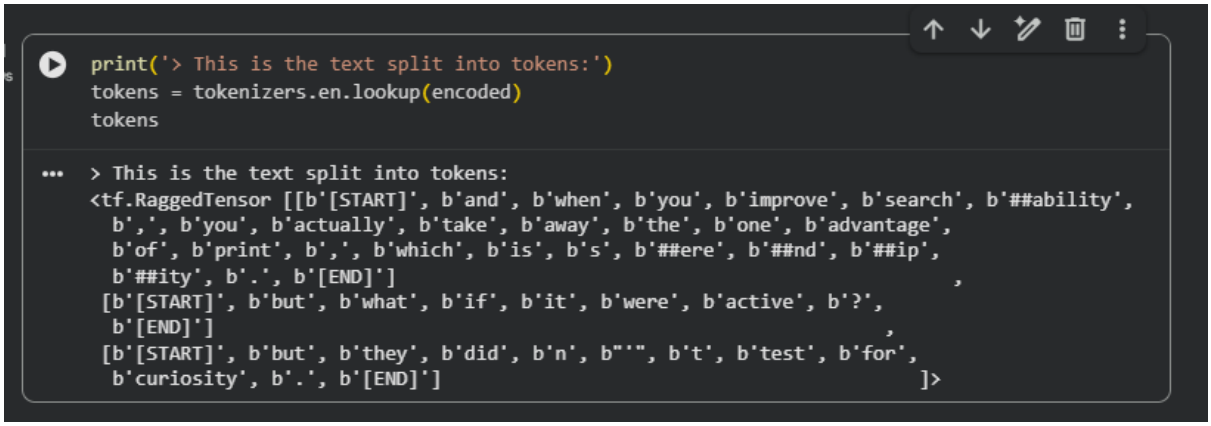
- https://www.youtube.com/watch?v=SZorAJ4I-sA
- https://www.youtube.com/watch?v=zxQyTK8quyY
- https://www.youtube.com/watch?v=wjZofJX0v4M

Additionally, I used AI to explain some concepts a little bit easier.

Transformers are a new form of neural networks. It is mostly used to process language data. Use cases would be automated translation or chatbots. I will discuss the most important principles here.

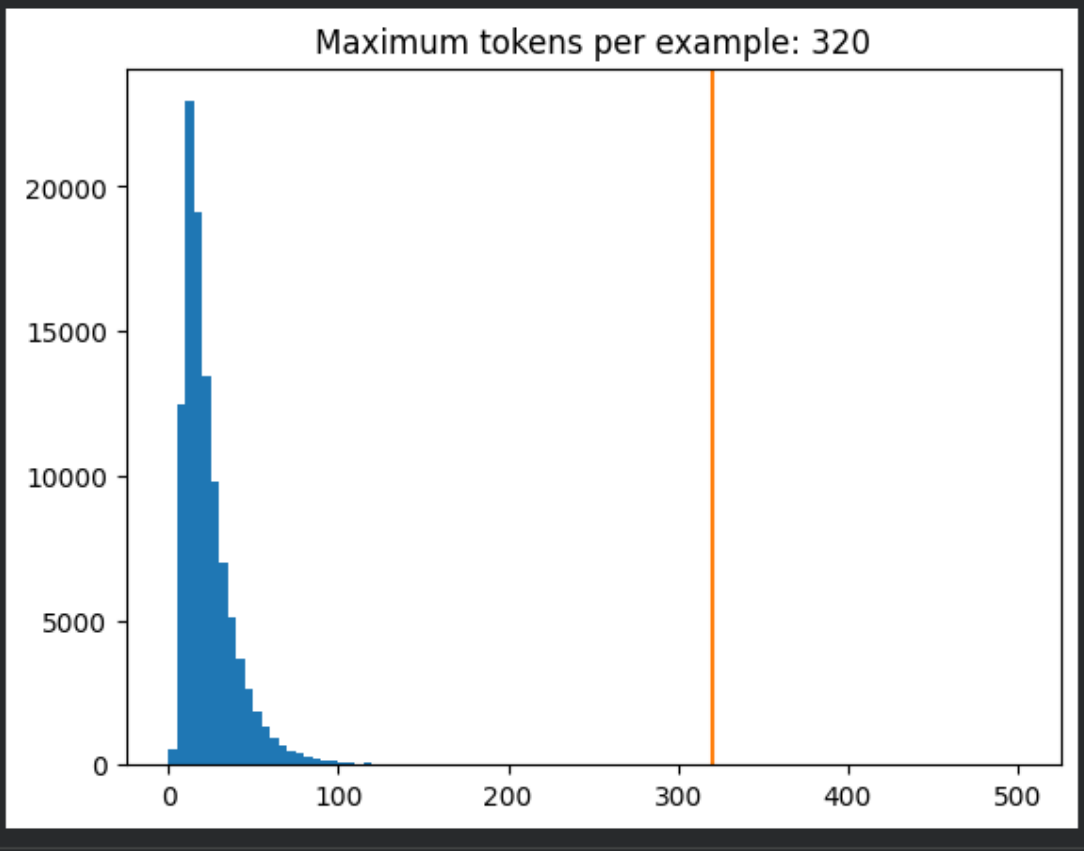
Tokenization

After loading a dataset, the first important step in a transformer is to split the text data into tokens. Each token refers to one specific meaning. A token is mostly a word, for example the word “you” is one token. Words like searchability are split into two tokens “search” and “ability”. Some words are split into multiple tokens because endings could differ and have a specific implication on the word state. In the notebook, the word “serendipity“ is splited into „s“, “ere”, “nd”, “ip” and “ity” or the word didn’t is split into “did”, “n”, “’” and “t”.



After this step, a histogram shows the tokens per sentence. This metrics is important to define in the model how many tokens should be processed per batch. If the defined max tokens are lower than the longest sentence, the sentence is cut off and Information goes missing. This is not a huge problem but if many sentences have more tokens than the max tokens variable, more information will be lost. The higher the max tokens value is, the more computing power it needs. Since the histogram shows, that most sentences are lower than ~128, the default value of 128 will not be changed.

```
all_lengths = np.concatenate(lengths)

plt.hist(all_lengths, np.linspace(0, 500, 101))
plt.ylim(plt.ylim())
max_length = max(all_lengths)
plt.plot([max_length, max_length], plt.ylim())
plt.title(f'Maximum tokens per example: {max_length}');
```



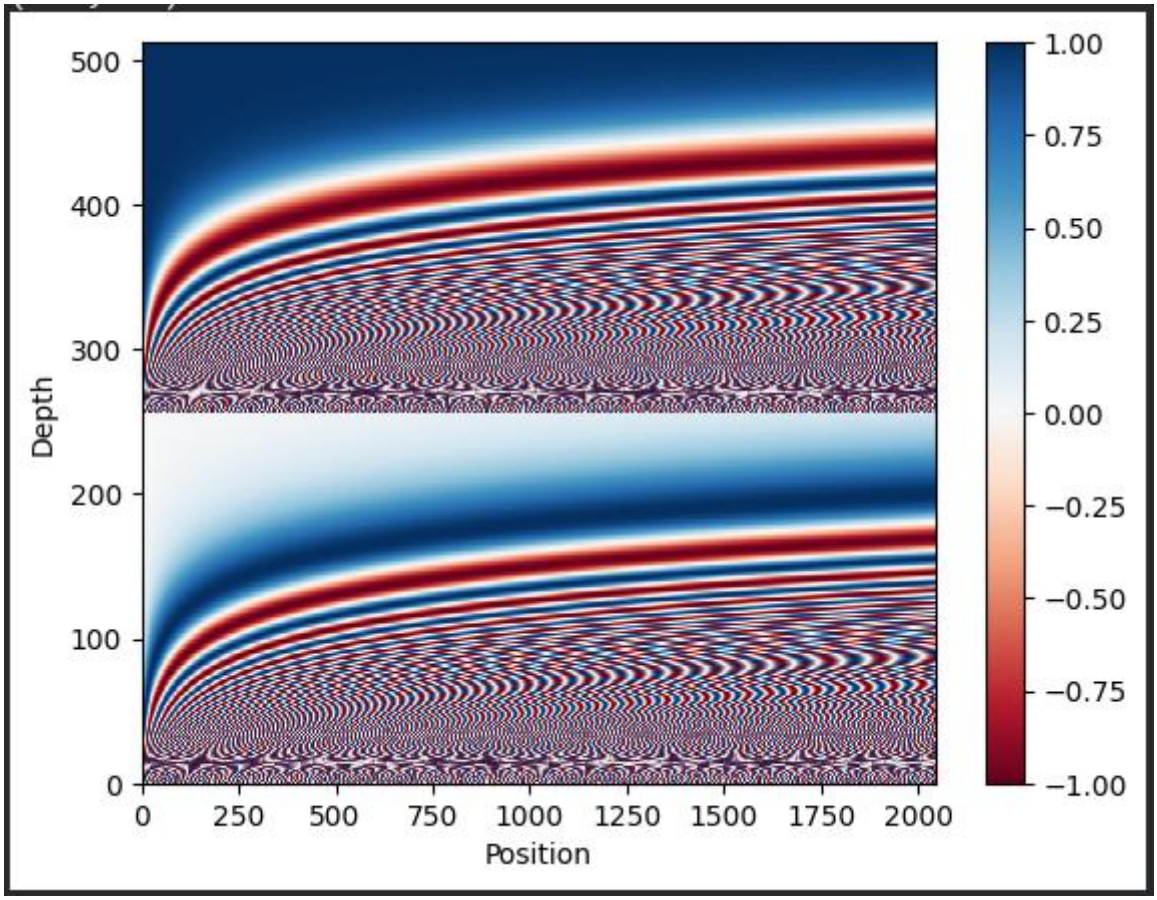
The data gets batched afterwards to have a better performance.

Positional embedding

Positional embedding means, that the position of a word in a sentence is added to Word as a vector. This means that the vector of the word “Dog” at the first position of a sentence is different from the vector of “Dog” at the last position. The position of a word is calculated with sines and cosines waves. A simple version of positional embedding shows this code snippet:

```python
def positional_encoding(length, depth):
  depth = depth/2

  positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
  depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

  angle_rates = 1 / (10000**depths)         # (1, depth)
  angle_rads = positions * angle_rates      # (pos, depth)

  pos_encoding = np.concatenate(
      [np.sin(angle_rads), np.cos(angle_rads)],
      axis=-1)

  return tf.cast(pos_encoding, dtype=tf.float32)
```

With a plot, it is possible to see a pattern in the vector data. This shows (at least a little bit) how math finds a structure in language:



Attention

Attention means, that an algorithm finds a structure between a selected word and all the other words in a sentence. This means that it not only analyses the word itself but also its surroundings. As an example, in the sentences “Server, can I have the check please” and “I think I just crashed the server”, the word “Server” has a completely different meaning. A machine realizes that, if it analyses a lot of data and compares the word server with the word “check” and “crashes” to understand the different meanings. Multihead attention means, that this comparison is done multiple times, parallel and from different perspectives, to find different patterns. The base class of Attention looks like this:

```python
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

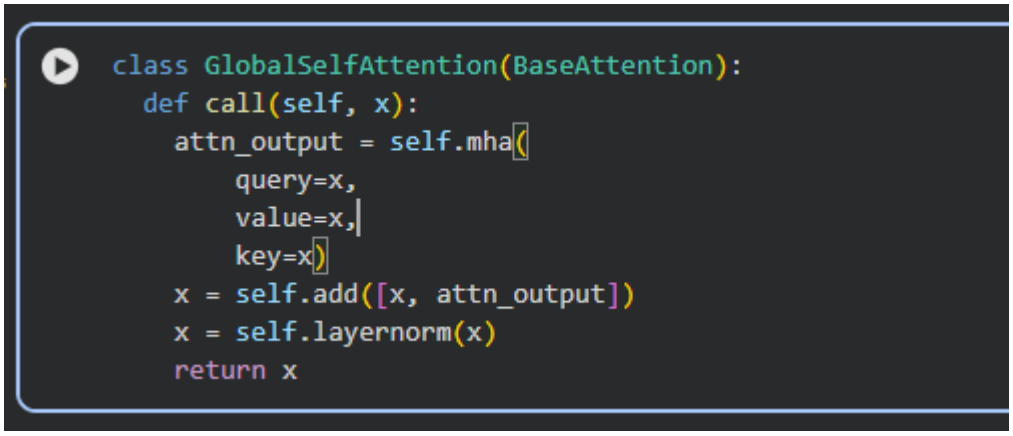
Cross Attention

Cross Attention is to find patterns within the data from the encoder and decoder.

```python
class CrossAttention(BaseAttention):
  def call(self, x, context):
    attn_output, attn_scores = self.mha(
        query=x,
        key=context,
        value=context,
        return_attention_scores=True)

    # Cache the attention scores for plotting later.
    self.last_attn_scores = attn_scores

    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x
```

Global Selfattention

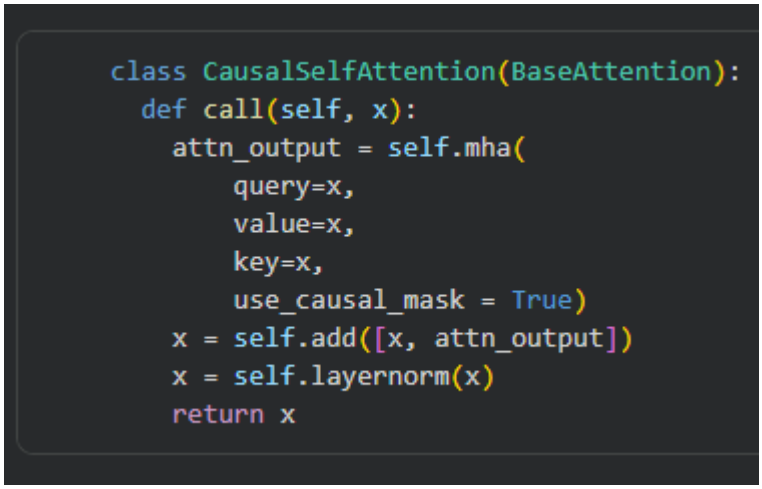
The global self attention is a function that compares a token with every token from the decoder. Compared to Attention, where a token searches for another token with a higher connection, self attention compares the token with everything to find new connections.



```
class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

Causal Self Attention

The causal self attention is a class that uses a mask to ensure that an output is only based on words that are already part of the output. The transformer is not allowed to “look into the future”.



```
class CausalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x,
        use_causal_mask = True)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

Neural Network

To process the attentions of a token, a neural network is used:

```python
class FeedForward(tf.keras.layers.Layer):
  def __init__(self, d_model, dff, dropout_rate=0.1):
    super().__init__()
    self.seq = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model),
      tf.keras.layers.Dropout(dropout_rate)
    ])
    self.add = tf.keras.layers.Add()
    self.layer_norm = tf.keras.layers.LayerNormalization()

  def call(self, x):
    x = self.add([x, self.seq(x)])
    x = self.layer_norm(x)
    return x
```

Creating the Encoder and Decoder

After defining the attentions and the neural network, it is time to build the encoder and decoder.An encoder processes a dataset of tokens to add context and a decoder finds the right output for the tokens. Every encoder consists of multiple layers. A layer consists of the GlobalSelfAttention and a neural network that was defined before.

```python
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.1):
    super().__init__()

    self.self_attention = GlobalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x):
    x = self.self_attention(x)
    x = self.ffn(x)
    return x
```

When building the encoder, the encoder layers and the positional embedding is added to the layer.

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```

For creating the decoder, causal self attention and cross attention as well as a neural network is added to the layer. The decoder consist like the encoder of a positional embedding and multiple decoder layers.

Build the transformer

In order to complete the transformer, the encoder and decoder are added to the Transformer class. Additionally a final layer is added to the transformer. This final layer creates probabilities for each word out of the decoded vectors to predict new words.

```python
class Transformer(tf.keras.Model):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
               input_vocab_size, target_vocab_size, dropout_rate=0.1):
    super().__init__()
    self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=input_vocab_size,
                           dropout_rate=dropout_rate)

    self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=target_vocab_size,
                           dropout_rate=dropout_rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inputs):
    # To use a Keras model with `.fit` you must pass all your inputs in the
    # first argument.
    context, x  = inputs

    context = self.encoder(context)  # (batch_size, context_len, d_model)

    x = self.decoder(x, context)  # (batch_size, target_len, d_model)

    # Final linear layer output.
    logits = self.final_layer(x)  # (batch_size, target_len, target_vocab_size)

    try:
      # Drop the keras mask, so it doesn't scale the losses/metrics.
      # b/250038731
      del logits._keras_mask
    except AttributeError:
      pass

    # Return the final output and the attention weights.
    return logits
```