# The Transformer: Code

We will examine a notebook that builds a miniature version of GPT: tutorial view (https://keras.io/examples/generative/text_generation_with_miniature_gpt/)

- Colab notebook (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipy

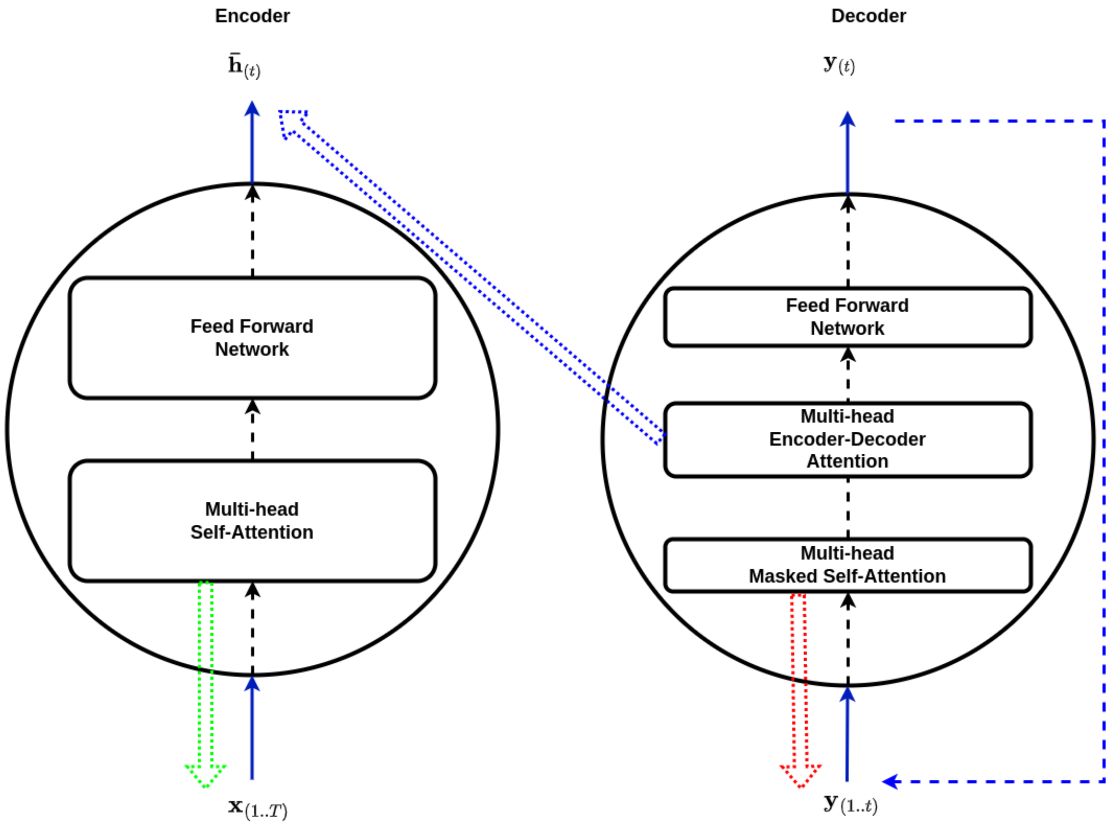
For an excellent tutorial on all the concepts, along with code, see (https://www.tensorflow.org/text/tutorials/transformer)

GPT-3 is a Decoder style Transformer

- autoregressive

Recall from our introduction to the Transformer (Encoder-Decoder)

**Transformer Layer (Encoder/Decoder)**




The Decoder is the RHS of the image.

[Here (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scr](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scr)
we can see the Decoder

We first see a definition of the constants:

```
vocab_size = 20000  # Only consider the top 20k words
maxlen = 80  # Max sequence size
embed_dim = 256  # Embedding size for each token
num_heads = 2  # Number of attention heads
feed_forward_dim = 256  # Hidden layer size in feed forward network inside tran
sformer
```

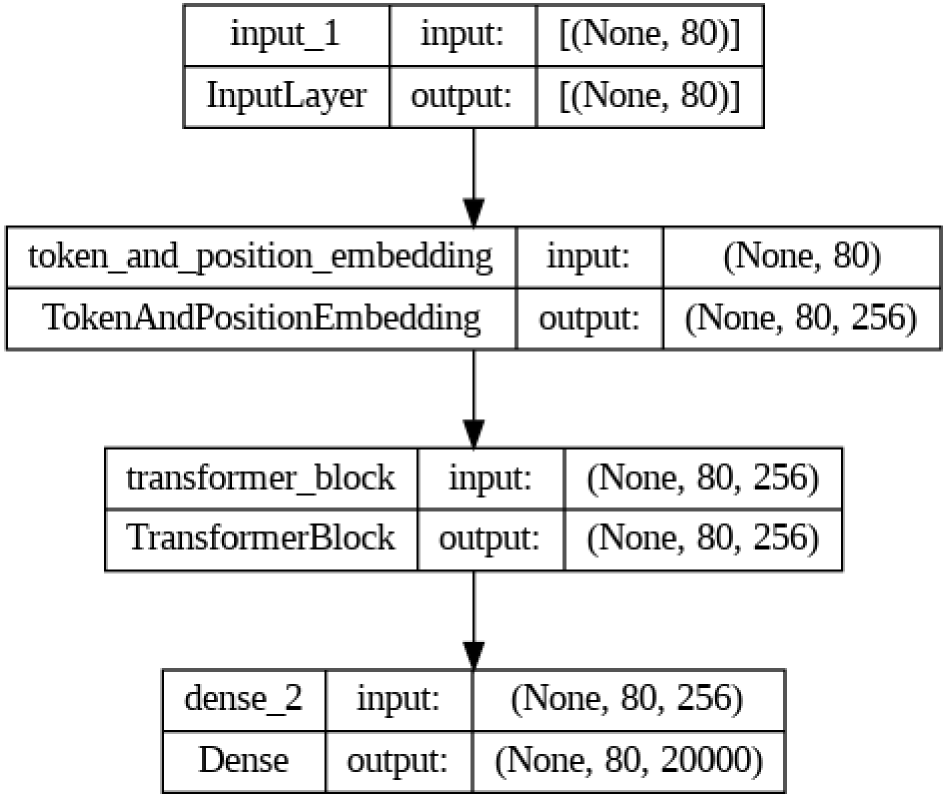
Relating the variable names to our notation

| **Notation** | **variable** | **value** |
|---|---|---|
| $d_{\text{model}}$ | embed_dim | 256 |
| $T$ | max_len | 80 |
| $n_{\text{heads}}$ | num_heads | 2 |
| | vocab_size | 20,000 |

And the Decoder model:

```
def create_model():
    inputs = layers.Input(shape=(maxlen,), dtype=tf.int32)
    embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
    x = embedding_layer(inputs)
    transformer_block = TransformerBlock(embed_dim, num_heads, feed_forward_di
m)
    x = transformer_block(x)
    outputs = layers.Dense(vocab_size)(x)
    model = keras.Model(inputs=inputs, outputs=[outputs, x])
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    model.compile(
        "adam", loss=[loss_fn, None],
    )  # No loss and optimization based on word embeddings from transformer blo
ck
    return model
```

Here is the plot:



Examining each layer

- `Input`
    - sequence (length $T = 80$) of integers (index of a character within vocabulary) $\mathbf{y}_{(1:T)}$
- `TokenAndPositionEmbedding`
    - maps sequence (length $T = 80$) of integers (index of character)
    - into sequence (length $T = 80$) of $d_{\text{model}} = 256$ size representations
- `TransformerBlock`
    - maps sequence (length $T = 80$) into sequence of latents $\mathbf{h}_{(1:T)}$
        - one latent per position in input

- `Dense`
    - Classifier layer
    - maps sequence of latents
    - to sequence of probability vectors
        - each position is a probability vector of length `vocab_size` $= 20000$
        - position $i$: probability that output is element $i$ of vocabulary
        - sum across positions in each vector is 100%

# Loss function

The `create_model` method also defines the Loss Function

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

as Cross Entropy, as is common for a Classifier

Notice that the `SparseCategoricalCrossentropy` takes a vector (of length `vocab_size`) of **logits** rather than **probabilities**.

# TransformerBlock

Let's examine the TransformerBlock (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrb) in more detail

In [ ]:

```
    class TransformerBlock(layers.Layer):
        def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
            super().__init__()
            self.att = layers.MultiHeadAttention(num_heads, embed_dim)
            self.ffn = keras.Sequential(
                [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_di
m),]
            )
            self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
            self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
            self.dropout1 = layers.Dropout(rate)
            self.dropout2 = layers.Dropout(rate)

        def call(self, inputs):
            input_shape = tf.shape(inputs)
            batch_size = input_shape[0]
            seq_len = input_shape[1]
            causal_mask = causal_attention_mask(batch_size, seq_len, seq_len, t
f.bool)
            attention_output = self.att(inputs, inputs, attention_mask=causal_ma
sk)
            attention_output = self.dropout1(attention_output)
            out1 = self.layernorm1(inputs + attention_output)
            ffn_output = self.ffn(out1)
            ffn_output = self.dropout2(ffn_output)
            return self.layernorm2(out1 + ffn_output)
```

We can see that the TransformerBlock is implemented as a Layer (`layers.Layer`)

- so it will translate its input into output via a `call` method

The class `__init__` method defines the components of the Transformer

- stores them in instance variables:
    - Attention: `self.att`
    - Feed Forward Network FFN: `self.ffn`
    - Other: Layer Norms, Dropouts

The `call` method does the actual work

- Masked self-attention to $\mathbf{y}_{(1:T)}$
    - Creates casual mask `causal_mask` to prevent peeking ahead at not-yet-generated output
        - `seq_len` is current length $t$ of $\mathbf{y}_{1:t)}$
    - Attention block `self.att` applied to causally-masked input

      ```
      attention_output = self.att(inputs, inputs, attention_mask=causal_mask)
      ```
- Dropout `self.dropout1` and LayerNorm `layernorm1` applied to attention output
- Result passed through Feed Forward Network `self.ffn`

## TokenAndPositionEmbedding

Let's examine the [TokenAndPositionEmbedding (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrc)](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrc)

```
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super().__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions
```

We can see that it too is implemented as a Layer.

The `call` method

- translates the input sequence
    - each position in the sequence is an integer index within the vocabulary

- into a sequence of pairs

    - first element: token embedding

      ```
      x = self.token_emb(x)
      ```

    - second element: position embedding

      ```
      positions = tf.range(start=0, limit=maxlen, delta=1)
      positions = self.pos_emb(positions)
      ```

As explained [in a prior module (Transformer_PositionalEmbedding.ipynb#Representing-the-combined-token-and-positional-encoding)](Transformer_PositionalEmbedding.ipynb#Representing-the-combined-token-and-positional-encoding)

- The output is not actually a sequence of *pairs*
    - it is a sequence of numbers
    - the token and positional emeddings are *added* not concatenated
        - concatenation would double the length
        - all layers in Transformer preserve output length equal input length = $d_{\text{model}}$
- See the module's explanation as to why addition works

# Dense (Feed Forward Network)

We can see that the Feed Forward Network are two Dense layers

```
self.ffn = keras.Sequential(
            [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_dim),]
        )
```

We may have been expecting the final layer of `TransformerBlock` to be outputting a probability vector (over the Vocabulary)

- a vector of length `vocab_size`
    - position $i$ is probability that output is element $i$ of the Vocabulary
- using a `softmax` activation
    - to make sure sum (across the `vocab_size` elements of the vector) of probabilities is `00%

But we see that the output is

- a singleton (not a vector)
- of size equal to `embed_dim` = $d_{\text{model}}$

That is:

- the `Dense` component of the `TransformerBlock` is outputing the embedding of $\hat{\mathbf{y}}_{(t)}$ rather than a probability vector

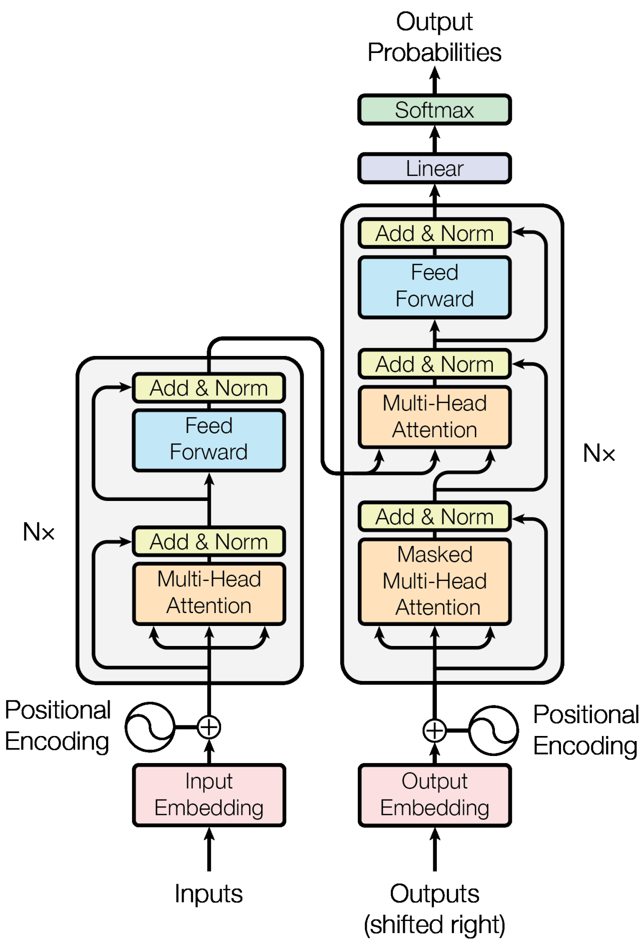
As we will see

- there is a layer in the Model *after* the `TransformerBlock`
- that produces the probability vector

# Skip connections

Here is a more detailed view of the Transformer

**Transformer (Encoder/Decoder)**



In particular, please focus on the arrows *into the "Add & Norm" layers.*

These are *skip connections* that bypass the Attention layers.

- *Residual Networks*

Where is this reflected in the code ?

It is a little subtle and easy to miss.

With the `call` method of the `TransformerBlock` please notice the statement

```
out1 = self.layernorm1(inputs + attention_output)
```

- `inputs` is the input to the Attention layer

  attention_output = self.att(inputs, inputs, attention_mask=causal_mask)

So the addition

```
inputs + attention_output
```

is joining (via addition)

- the output of the Attetnion layer
- the input of the Attention layer

This is the skip connection !

Similar code appears

```
ffn_output = self.ffn(out1)
ffn_output = self.dropout2(ffn_output)
return self.layernorm2(out1 + ffn_output)
```

where

- the input to the FFN (i.e., `out1`)
- is joined (via addition) to the output of the FFN (i.e., `ffn_output` )

    ```
    out1 + ffn_output
    ```

# Model

By examining the `create_model` function, we see that the output of the `TransformerBlock`

- is fed into a `Dense` layer
- which outputs a vector of length `vocab_size` (the correct length of a probability vector)
- and the output of this `Dense` layer is the output of the **model**
    - not the output of the `TransformerBlock`

      ```
      outputs = layers.Dense(vocab_size)(x)
      model = keras.Model(inputs=inputs, outputs=[outputs, x])
      ```
- Technically: the output vector is of *un-normalized logits* rather than probabilities

### - the logit vector can be turned into a probability vector via a `softmax`

Thus, the Model outputs a vector of logits.

We can see how a token is sampled

- by converting the logit vector into a probability vector
- with the `sample_from` method of the `TextGenerator` callback

  def sample_from(self, logits):

  ```
          logits, indices = tf.math.top_k(logits, k=self.k, sorted=True)
          indices = np.asarray(indices).astype("int32")
          preds = keras.activations.softmax(tf.expand_dims(logits, 0))[0]
          preds = np.asarray(preds).astype("float32")
          return np.random.choice(indices, p=preds)
  ```

Rather than outputting a probability vector

- which would require the user choosing one element from the vector (a word in the vocabulary)
- what is output is the *embedding* of the chosen word in the vocabulary

Since this output is compared against the correct label (i.e, $\mathbf{y}_{(t+1)}$ for position $t$)

- we should also see that the *labels* used are embeddings

## Training

A [`TextGenerator` (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrf)](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrf) call-back is used during training

- at the end every `self.print_every` epochs
- a sample of $\hat{\mathbf{y}}_{(1:T)}$ will be drawn
- to illustrate what the model output would be up to that point in training

The heart of the call-back

```
while num_tokens_generated <= self.max_tokens:
    ...
    y, _ = self.model.predict(x)
    sample_token = self.sample_from(y[0][sample_index])
    ...
```

- is a loop over positions $t$
- that extends a fixed input (prefix of text) `start_tokens`
- to full length $T$
- by sampling a token from the output for position $t$

This is useful

- to see whether our model is learning as epochs advance
- to confirm the shape and type of the model output is a vector of logits
    - the model output for position $t$: `y, _ = self.model.predict(x)`
    - is passed to `sample_from`
    - which samples from the probability distribution derived from the logits (model output)

In [2]:
```
print("Done")
```

```
Done
```