

## TensorFlow Transformer Tutorial (<https://www.tensorflow.org/text/tutorials/transformer>)

The link in the title is to an *excellent* tutorial on the Transformer

- more in-depth than this notebook
- background

It is recommended especially for those readers without a background in Transformers from the Intro course.

We will take a look at the actual code of a Transformer.

There are many pieces, which we will examine individually.

We will proceed starting with a high level view and descend to a lower level

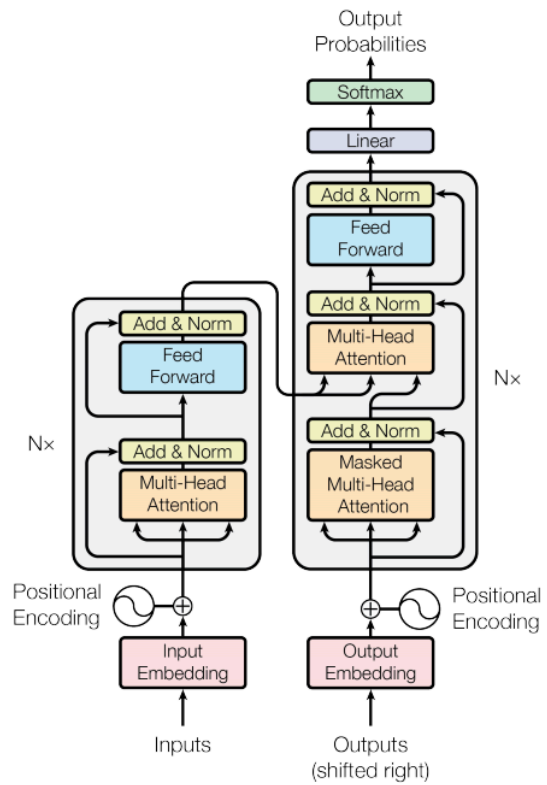
- means reading the code from bottom to top

There are many subtle points which we will highlight with the tag **SUBTLETY**

One of the key components of a Transformer is the Attention mechanism.

In the code we examine, the base Attention class is via a `MultiHeadAttention` layer type

- we will study this layer separately
- so as not to distract from the other details of the Transformer architecture



## The Model: Transformer ([https://www.tensorflow.org/text/tutorials/transformer#the\\_transformer](https://www.tensorflow.org/text/tutorials/transformer#the_transformer))

- The Transformer is a Model: a subclass of `tf.keras.Model`
- The initializer creates
  - An Encoder
  - A Decoder
  - a `final_layer` which converts the vector at each position into logits over the distribution of tokens

```
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)
```

The model overrides the `call` method

- defines what happens when we pass an input to the Transformer
- passes the `context` input to the Encoder
- the Encoder output is passed to the Decoder
- the Decoder output (logits) is passed through a layer to produce a logit (at each position)

```

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_s
ize)

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

```



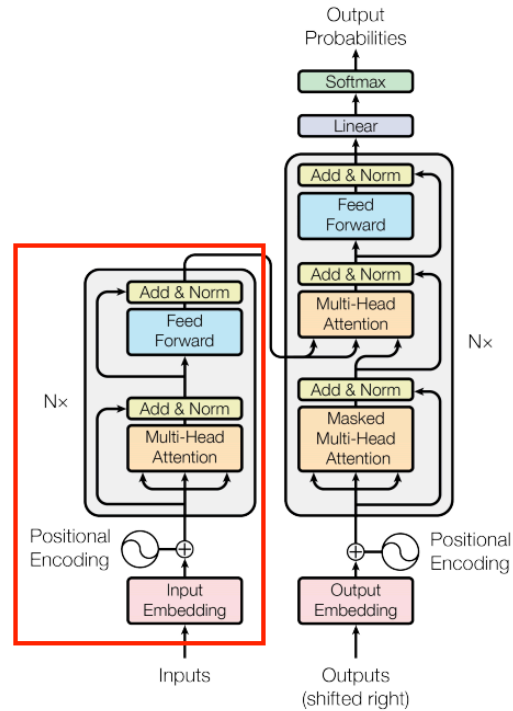
### **Remember**

- the `init` method of a model usually defines the model's components
- the `call` method describes how model input is transformed to model output

These are two very good places to begin when trying to understand most models.

## The Encoder

([https://www.tensorflow.org/text/tutorials/transformer#the\\_encoder\\_layer](https://www.tensorflow.org/text/tutorials/transformer#the_encoder_layer))



### Confusion warning

The Encoder is a stack of  $N$  "blocks"

- that is what the "Nx" in the diagram refers to

Each block is implemented by the class `EncoderLayer` .

The `Encoder` object is the *stack* of encoder blocks ( `EncoderLayer` 's)

The `Encoder` is a `Layer`: sub-class of `tf.keras.layers.Layer`)

- The initializer creates the sub-components of the `Encoder`
  - Positional Embedding
  - A sub-component (`enc_layers`) which is an **array** of `EncoderLayer` blocks whose elements are objects containing
    - Self-Attention
    - Feed-forward network
  - This array (of length `num_layers`) is the *stack* of blocks

```

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)

```

The `call` method defines how the layer behaves when presented with input

- calls the Positional Embedding on the Encoder input
- passes the result through each block in the stacked EncoderLayer's
  - Self-Attention followed by Feed Forward

```
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)`.
```

The `EncoderLayer` (one block) class has

- an initializer that creates sub-components
- a `call` method is over-ridden to pass inputs through the sub-components



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

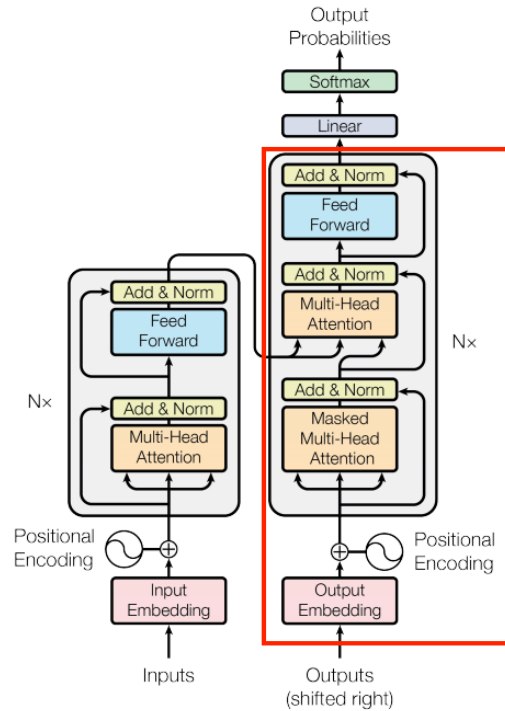
        self.self_attention = GlobalSelfAttention(
            num_heads=num_heads,
            key_dim=d_model,
            dropout=dropout_rate)

        self.ffn = FeedForward(d_model, dff)

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

## The Decoder

([https://www.tensorflow.org/text/tutorials/transformer#the\\_decoder](https://www.tensorflow.org/text/tutorials/transformer#the_decoder))



### Confusion warning

The `Decoder` object is the *stack* of decoder blocks (which are called `DecoderLayer` 's)

The `Decoder` is a `Layer`: sub-class of `tf.keras.layers.Layer` )

- The initializer creates the sub-components of the `Decoder`
  - Positional Embedding
  - A sub-component (confusingly named `DecoderLayer`) which is an **array** of blocks whose elements are objects containing
    - Self-Attention
    - Cross-Attention
    - Feed-forward network
  - This array (of length `num_layers`) is the *stack* of blocks

```

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

        self.d_model = d_model
        self.num_layers = num_layers

        self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                                  d_model=d_model)
        self.dropout = tf.keras.layers.Dropout(dropout_rate)
        self.dec_layers = [
            DecoderLayer(d_model=d_model, num_heads=num_heads,
                        dff=dff, dropout_rate=dropout_rate)
            for _ in range(num_layers)]

        self.last_attn_scores = None

```

The `call` method defines how the layer behaves when presented with input

- calls the Positional Embedding on the Decoder input
- passes the result to the stacked `DecoderLayer`'s
  - *Causal Self-Attention* followed by
  - Cross-Attention followed by Feed Forward

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

    x = self.dropout(x)

    for i in range(self.num_layers):
        x = self.dec_layers[i](x, context)

    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x
```

## The DecoderLayer

- initializer creates sub-components

```

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

        self.causal_self_attention = CausalSelfAttention(
            num_heads=num_heads,
            key_dim=d_model,
            dropout=dropout_rate)

        self.cross_attention = CrossAttention(
            num_heads=num_heads,
            key_dim=d_model,
            dropout=dropout_rate)

        self.ffn = FeedForward(d_model, dff)

```



The `call` method is over-ridden to pass inputs through the sub-components

```
def call(self, x, context):
    x = self.causal_self_attention(x=x)
    x = self.cross_attention(x=x, context=context)

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

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

Let us focus on the two forms of Attention

- context is the Encoder output
- $x$  is the Decoder input

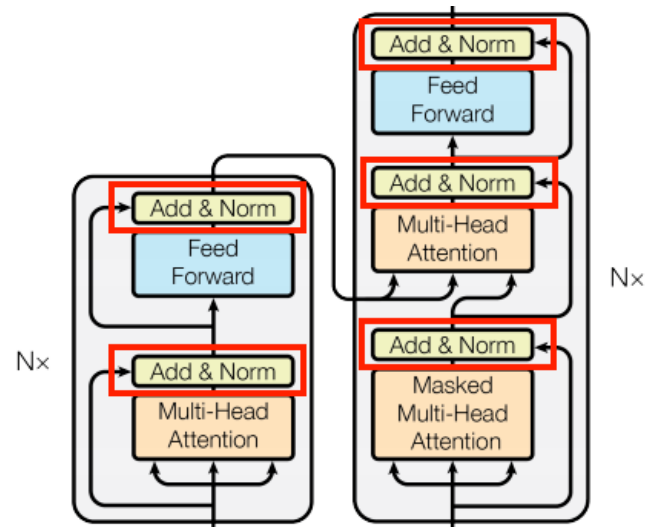
The *Causal* Self-Attention uses query  $x$  (at each position) to attend to entire sequence  $x$ .

- the attention is *causal*: for each position, future positions *may not* be attended to

The Cross-Attention uses query  $x$  (at each position) to attend to Encoder output context

## Add and Normalize

([https://www.tensorflow.org/text/tutorials/transformer#add\\_and\\_normalize](https://www.tensorflow.org/text/tutorials/transformer#add_and_normalize))



## Review: Layer Normalization

- The variance of outputs tends to grow from layer to layer
  - Large variance causes gradient updates to become unstable
  - [Layer Normalization](https://proceedings.neurips.cc/paper_files/paper/2019/file/2f4fe03d77724a7217/Paper.pdf) ([https://proceedings.neurips.cc/paper\\_files/paper/2019/file/2f4fe03d77724a7217/Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2019/file/2f4fe03d77724a7217/Paper.pdf)) reduces the variance of the input distribution to unit variance
-

## SUBTLETY

The output of Attention layers (both Self Attention and Cross Attention) are feed into an Add & Norm block.

In what seems to be a "coding convenience"

- the code creates a common base class `BaseAttention`
  - for both Self Attention and Cross Attention
  - that is partially responsible for **both**
    - Attention
    - Add & Norm

This is much more subtle than "coding convenience" !

The initializer of `BaseAttention` creates sub-components

- Attention
- Layer Normalization
- Add

```
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()
```

**but** doesn't actually perform the normalization or addition.

- there is no `call` method of the base class
- these are left to the child (Attention) classes

The *child* sub-class (GlobalSelfAttention or CrossAttention)

- **uses** the components of its parent class
  - `self.mha`: to implement the attention call
  - `self.layernorm, self.add`: to implement the Add & Norm

To illustrate, here is the code for the `CrossAttention` sub-class

- note the arguments to `mha`
  - query is `x`; key and value are both `context` (the Encoder output)

```
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.layer_norm(x)

        return x
```



### **What is the purpose of Add & Norm**

The "obvious" purpose is to normalize the Attention outputs

- using a `tf.keras.layers.LayerNormalization` layer
- that is the Norm part of Add & Norm

It is *easy to miss* the role of the Add part.

Mechanically: the `Add` is uninteresting.

The `Add` part adds the block's two inputs (i.e, Attention input and Attention output)

- before Normalization
- In both the Self-Attention and Cross Attention children, the `call` method performs the `Add` and `Norm` via statements

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

- where `x` is the Attention input and `attn_output` is the Attention output.

But what is the **purpose** of adding Attention input and Attention output ?

This creates a *residual* or *skip* connection

- on the forward pass, the input to Attention can "skip over" the Attention block
- more importantly: on the backward pass: the loss gradient can skip over the Attention block

[Review: Residual connections \(RNN\\_Residual\\_Networks.ipynb#Residual-connections:-a-gradient-highway\).](#)

- Gradients can vanish or explode as they traverse an increasing number of layers during back propagation
- A zero gradient causes the Gradient update step to leave weights unchanged
  - the model can't "learn"
- The skip connection prevents gradients from vanishing or exploding by allowing them to by-pass one or more layers in the backward pass

So Add & Norm is much more than "good coding"

- observing that Attention outputs are always fed into common blocks

It is also the mechanism by which the residual connections are implemented.

## Attention

The Self Attention (the class is called `GlobalSelfAttention`) and Cross Attention blocks are both derived from `BaseAttention`

- which we explained in the section on "Add and Norm".

The sub-components (including the class `MultiHeadAttention` that implements `Attention`) are created by the parent class.

The child classes mainly implement the `call` method

- that invokes the sub-components in sequence
- and implement the residual connection

We already saw the call for CrossAttention above.

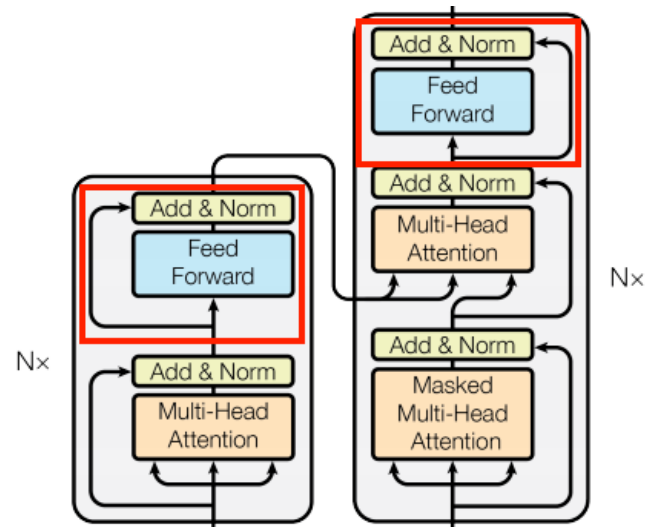
For Self-Attention, the call follows note the arguments to mha

- query, key and value are identical and equal to `x` (the Decoder input)

```
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
```

## Feed forward

([https://www.tensorflow.org/text/tutorials/transformer#the\\_feed\\_forward\\_network](https://www.tensorflow.org/text/tutorials/transformer#the_feed_forward_network))





The purpose of the Feed Forward block

- is to transform the output of an Attention layer
- into
  - Encoder: the "context sensitive" representation
  - Decoder: (logit of the) next output token

This is also where it is believed that "world knowledge" encountered during training is stored.

The typical Feed Forward network is two Dense layers

- the first has  $d_{\text{ff}}$  units
  - creating  $d_{\text{ff}}$  synthetic features from the  $d_{\text{model}}$  features of the Attention output
- the second has  $d_{\text{model}}$  units
  - re-sizing the output to the standard  $d_{\text{model}}$  output size of all blocks in a Transformer through two Dense layers.

In the original paper

$$d_{\text{ff}} = 4 * d_{\text{model}}$$

and this seems to have become a common choice.

Here is the code:

```
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
```

## SUBTLETY

The Feed Forward output is passed to an Add & Norm block

- which has **two inputs**
  - Feed Forward output and Feed Forward input
  - the Feed Forward input is a residual connection
- similar to the residual connection we saw in the "Add and Normalize" section.

The residual connection is implemented in the `call` via the statements

```
x = self.add([x, self.seq(x)])  
x = self.layer_norm(x)
```

where

- `x` is the input to the Feed Forward block
- `self.seq(x)` is the output of the Feed Forward block
  - the input passed through the two Dense layers, implemented as a `Sequential` model

## Training

(<https://www.tensorflow.org/text/tutorials/transformer#training>).

### Teacher forcing

#### SUBTLETY

A Generative task (like the LLM objective) exhibits Autoregressive behavior

- the Decoder output  $\hat{\mathbf{y}}_{(t-1)}$  at position  $(t - 1)$  is fed back as *input* for position  $t$ .

In the Transformer, the position  $(t - 1)$  output is appended to all previous outputs.

Thus, at *inference* time: the input for position  $t$  is  $\hat{\mathbf{y}}_{([1:t-1])}$

But, this **exact** behavior is not conducive to *training*

- Suppose  $\hat{\mathbf{y}}_{(t-1)}$  is incorrect and not equal to correct label  $\mathbf{y}_{(t-1)}$
- this error cascades into the prediction of all subsequent positions  $\hat{\mathbf{y}}_{([t:])}$

So, during **training** time: the input for position  $t$  is  $\mathbf{y}_{(1:t-1)}$

- the *correct* sequence
- rather than the *predicted* sequence

This is why **causal masking** is necessary for the Decoder

- **Masked Self-Attention**
- don't want to look into the future
  - i.e., at positions  $t' \geq t$  when the decoder is producing  $\hat{\mathbf{y}}_{(t)}$



Feeding the *correct target* (rather than the actual *predicted target*) as the next input at training time

- is called *Teacher Forcing*
- does *not* occur at inference time
  - *predicted target* is used

It's very easy to *not notice* Teacher Forcing when it occurs because it is subtle.

Can you see where it occurs ?

It is in the *construction* of the Training examples

- the input for position  $t$  are the features of example  $t$ :  $\mathbf{y}_{([1:t-1])}$ 
  - *not* the Autoregressive constructed  $\hat{\mathbf{y}}_{([t:])}$

$i$	$\mathbf{x}^{(i)}$	$\mathbf{y}^{(i)}$
1	$\mathbf{y}_{(0)}$	$\mathbf{y}_{(1)}$
2	$\mathbf{y}_{([0:1])}$	$\mathbf{y}_{(2)}$
$\vdots$		
$t$	$\mathbf{y}_{([1:t-1])}$	$\mathbf{y}_{(t)}$
$\vdots$		
$T$	$\mathbf{y}_{([1:T-1])}$	$\mathbf{y}_{(T)}$

During training, each example trains for one "step"

- so we don't see the effect of  $\hat{\mathbf{y}}_{(t-1)}$  being fed back to the input for the next step  $t$

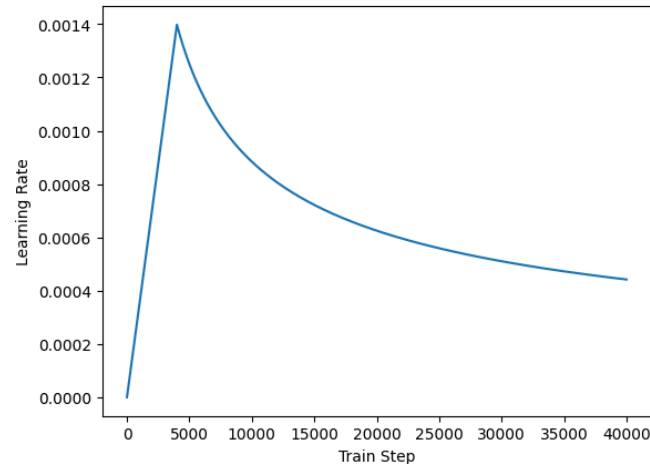
## Custom Learning Rate Schedule ([https://www.tensorflow.org/text/tutorials/transformer#set\\_up\\_the\\_optimizer](https://www.tensorflow.org/text/tutorials/transformer#set_up_the_optimizer))

A custom learning rate schedule (subclassed from `tf.keras.optimizers.schedules.LearningRateSchedule`) is created

- varies learning rate  $\alpha$  of Gradient update by epoch

$$\mathbf{W}_{(\text{epoch}+1)} = \mathbf{W}_{(\text{epoch})} - \alpha * \frac{\partial \mathcal{L}_{(\text{epoch})}}{\partial \mathbf{W}_{(\text{epoch})}}$$

- a warm-up period where  $\alpha$  increases
- a post-warm-up period where  $\alpha$  decays





The CustomSchedule object implements a customized Learning rate

- sub-class from  
tf.keras.optimizers.schedules.LearningRateSchedule

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):  
    def init(self,  
    d_model, warmup_steps=4000):  
        super().init()
```

```
        self.d_model = d_model  
        self.d_model = tf.cast(self.d_model, tf.float32)
```

```
        self.warmup_steps = warmup_steps
```

```
    def __call__(self, step):  
        step = tf.cast(step, dtype=tf.float32)  
        arg1 = tf.math.rsqrt(step)  
        arg2 = step * (self.warmup_steps ** -1.5)  
  
        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
```

### Loss and metrics

([https://www.tensorflow.org/text/tutorials/transformer#set\\_up\\_the\\_loss\\_and\\_metrics](https://www.tensorflow.org/text/tutorials/transformer#set_up_the_loss_and_metrics))

Since the targets are Categorical values, Cross Entropy is used as a loss.

**But:** the target is a sequence with *padding*

- the padding should not figure into the Loss
- so the loss is "masked" whenever the target `label` is a padding token (0)

Similarly the Accuracy metric is modified so that padding characters don't participate in the calculation.

```
def masked_loss(label, pred):
    mask = label != 0
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')
    loss = loss_object(label, pred)

    mask = tf.cast(mask, dtype=loss.dtype)
    loss *= mask

    loss = tf.reduce_sum(loss)/tf.reduce_sum(mask)
    return loss

def masked_accuracy(label, pred):
    pred = tf.argmax(pred, axis=2)
    label = tf.cast(label, pred.dtype)
    match = label == pred

    mask = label != 0
```

# Where do all the weights come from ?

Ignoring the weights associated with the various embeddings, the weights come from

- Attention
- Feed forward Network

This is for *each* Transformer block

- we will stack  $n_{\text{layer}}$  such blocks



For Attention, the weights/parameters are in the matrices  $\mathbf{W}_Q$ ,  $\mathbf{W}_K$ ,  $\mathbf{W}_V$  and  $\mathbf{W}_O$

- all of size  $\mathcal{O}(d_{\text{model}}^2)$ , total:  
 $4 * \mathcal{O}(d_{\text{model}}^2)$

For the Feed forward network, there are two Dense layers

- the first mapping attention output of size  $d_{\text{model}}$  to size  $d_{\text{ff}}$
- the second mapping size  $d_{\text{ff}}$  to standard output size  $d_{\text{model}}$
- total Feed forward weights are  $2 * (d_{\text{model}} * d_{\text{ff}})$

Using the standard

$$d_{\text{ff}} = 4 * d_{\text{model}}$$

total Feed forward weights per block

$$2 * (d_{\text{model}} * 4 * d_{\text{model}}) = 8 * \mathcal{O}(d_{\text{model}}^2)$$

$$\begin{aligned}
\text{Total parameters} &= \text{Projection matrix parms} + \text{FFN parms} \\
&= 4 * \mathcal{O}(d_{\text{model}}^2) + 8 * \mathcal{O}(d_{\text{model}}^2) \\
&= 12 * \mathcal{O}(d_{\text{model}}^2)
\end{aligned}$$

This is for a single Transformer block

- This gets multiplied by the number  $n_{\text{layer}}$  stacked blocks..

## Notice

- that  $\frac{1}{3}$  of the total weights
- come from *linear* projections
  - the matrices associated with Attention
- rather than non-linearities
  - confined to Feed forward network

For GPT-3

- $n_{\text{layer}} = 96$
- $d_{\text{model}} = 12 * 1024$

Total Transformer (non-embedding) weights

$$96 * 12 * (12 * 1024)^2 = 174 \text{ billion}$$

## Second example: Mini-GPT ([https://keras.io/examples/generative/text\\_generation\\_with\\_miniature\\_gpt/](https://keras.io/examples/generative/text_generation_with_miniature_gpt/))

We will examine a notebook that builds a miniature version of GPT: [tutorial view](#)  
([https://keras.io/examples/generative/text\\_generation\\_with\\_miniature\\_gpt/](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.ipynb) ([https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text\\_generation\\_with\\_miniature\\_gpt.ipynb](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb))
-

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 transformer
```

Relating the variable names to our notation

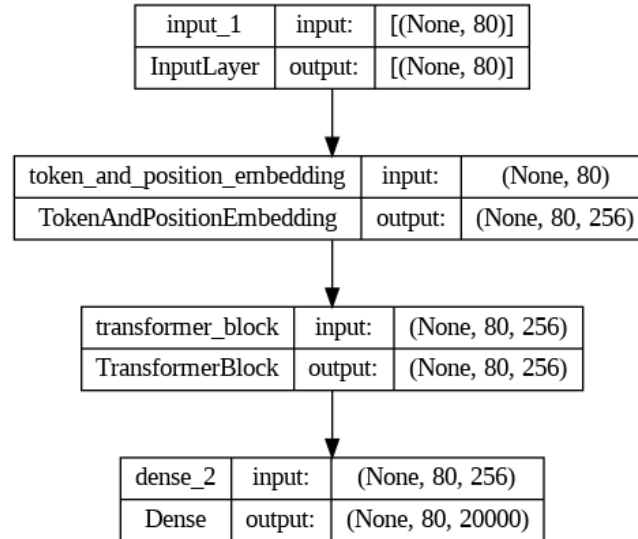
Notation	variable	value
$d_{\text{model}}$	embed_dim	256
$T$	max_len	80
$n_{\text{heads}}$	num_heads	2
	vocab_size	20,000

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_dim)
    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 block
    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#scrollTo=...) ([https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text\\_generation\\_with\\_miniature\\_gpt.ipynb#scrollTo=...](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scrollTo=...)) in more detail

---

```

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_dim),]
        )
        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, tf.bo
ol)
        attention_output = self.att(inputs, inputs, attention_mask=causal_mask)
        attention_output = self.dropout1(attention_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#scroll=0)  
([https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text\\_generation\\_with\\_miniature\\_gpt.ipynb#scroll=0](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scroll=0)).

```
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 \(Transformer\\_PositionalEmbedding.ipynb#Representing-the-combined-token-and-positional-encoding\)](#).

- 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 100%

But we see that the output is

- a singleton (not a vector)
- of size equal to `embed_dim = dmodel`

That is:

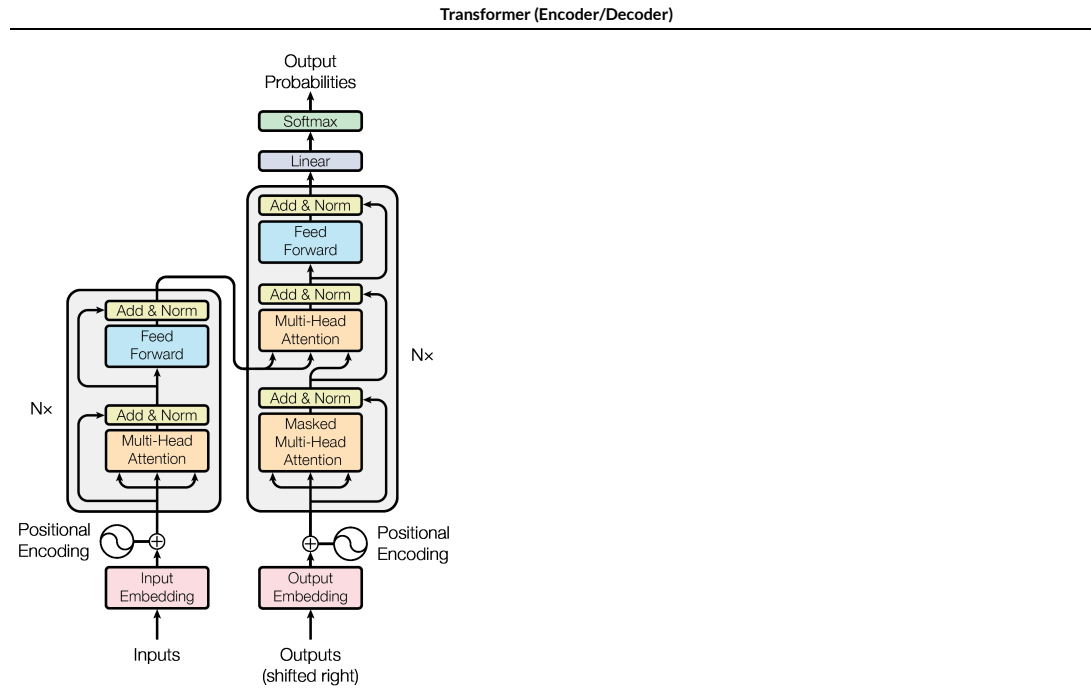
- the `Dense` component of the `TransformerBlock` is outputting the embedding of  $\hat{\mathbf{y}}_{(t)}$  rather than a probability vector

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



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 Attention 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.layer_norm2(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#scroll=1](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/text_generation_with_miniature_gpt.ipynb#scroll=1)) 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

