```
import tensorflow as tf
import numpy as np

# Positional Encoding
def positional_encoding(max_position, d_model):
angles = np.arange(max_position)[:, np.newaxis] / np.power(10000, (2 * (np.arange(d_model)[np.newaxis, :] // 2)) / d_model)
pos_encoding = np.zeros((max_position, d_model))
pos_encoding[:, 0::2] = np.sin(angles[:, 0::2])
pos_encoding[:, 1::2] = np.cos(angles[:, 1::2])
return tf.constant(pos_encoding[np.newaxis, ...], dtype=tf.float32)

# Multi-Head Attention
class MultiHeadAttention(tf.keras.layers.Layer):
def __init__(self, d_model, num_heads):
super(MultiHeadAttention, self).__init__()
self.num_heads = num_heads
self.d_model = d_model

assert d_model % num_heads == 0

self.depth = d_model // num_heads
self.wq = tf.keras.layers.Dense(d_model)
self.wk = tf.keras.layers.Dense(d_model)
self.wv = tf.keras.layers.Dense(d_model)
self.dense = tf.keras.layers.Dense(d_model)

def split_heads(self, x, batch_size):
x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
return tf.transpose(x, perm=[0, 2, 1, 3])

def call(self, q, k, v, mask=None):
batch_size = tf.shape(q)[0]

q = self.split_heads(self.wq(q), batch_size)
k = self.split_heads(self.wk(k), batch_size)
v = self.split_heads(self.wv(v), batch_size)

matmul_qk = tf.matmul(q, k, transpose_b=True)
dk = tf.cast(tf.shape(k)[-1], tf.float32)
scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

if mask is not None:
scaled_attention_logits += (mask * -1e9)

attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
```

```
scaled_attention = tf.matmul(attention_weights, v)

scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
concat_attention = tf.reshape(scaled_attention, (batch_size, -1, self.d_model))
return self.dense(concat_attention)

# Point-wise Feed-Forward Network
def point_wise_feed_forward_network(d_model, dff):
return tf.keras.Sequential([
tf.keras.layers.Dense(dff, activation='relu'),
tf.keras.layers.Dense(d_model)
])

# Encoder Layer
class EncoderLayer(tf.keras.layers.Layer):
def ___init___(self, d_model, num_heads, dff, dropout_rate=0.1):
super(EncoderLayer, self).___init___()
self.mha = MultiHeadAttention(d_model, num_heads)
self.ffn = point_wise_feed_forward_network(d_model, dff)

self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

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

def call(self, x, training, mask):
attn_output = self.mha(x, x, x, mask)
attn_output = self.dropout1(attn_output, training=training)
out1 = self.layernorm1(x + attn_output)

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

# Encoder
class Encoder(tf.keras.layers.Layer):
def ___init___(self, num_layers, d_model, num_heads, dff, input_vocab_size,
maximum_position_encoding, dropout_rate=0.1):
super(Encoder, self).___init___()
self.d_model = d_model
self.num_layers = num_layers

self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
self.pos_encoding = positional_encoding(maximum_position_encoding,
d_model)
```

```
self.enc_layers = [
EncoderLayer(d_model, num_heads, dff, dropout_rate) for _ in range(num_layers)
]

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

def call(self, x, training, mask):
seq_len = tf.shape(x)[1]

x = self.embedding(x)
x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
x += self.pos_encoding[:, :seq_len, :]

x = self.dropout(x, training=training)

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

return x

# Decoder Layer
class DecoderLayer(tf.keras.layers.Layer):
def __init__(self, d_model, num_heads, dff, dropout_rate=0.1):
super(DecoderLayer, self).__init__()
self.mha1 = MultiHeadAttention(d_model, num_heads)
self.mha2 = MultiHeadAttention(d_model, num_heads)

self.ffn = point_wise_feed_forward_network(d_model, dff)

self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

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

def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
attn1 = self.mha1(x, x, x, look_ahead_mask)
attn1 = self.dropout1(attn1, training=training)
out1 = self.layernorm1(x + attn1)

attn2 = self.mha2(out1, enc_output, enc_output, padding_mask)
attn2 = self.dropout2(attn2, training=training)
out2 = self.layernorm2(out1 + attn2)
```

```
ffn_output = self.ffn(out2)
ffn_output = self.dropout3(ffn_output, training=training)
return self.layernorm3(out2 + ffn_output)

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

self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
self.pos_encoding = positional_encoding(maximum_position_encoding,
d_model)

self.dec_layers = [
DecoderLayer(d_model, num_heads, dff, dropout_rate) for _ in range(num_layers)
]

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

def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
seq_len = tf.shape(x)[1]
attention_weights = {}

x = self.embedding(x)
x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
x += self.pos_encoding[:, :seq_len, :]

x = self.dropout(x, training=training)

for i in range(self.num_layers):
x = self.dec_layers[i](x, enc_output, training, look_ahead_mask, padding_mask)

return x

# Transformer
class Transformer(tf.keras.Model):
def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
target_vocab_size, pe_input, pe_target, dropout_rate=0.1):
super(Transformer, self).__init__()

self.encoder = Encoder(num_layers, d_model, num_heads, dff,
input_vocab_size, pe_input, dropout_rate)
```

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

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

def call(self, inp, tar, training, enc_padding_mask, look_ahead_mask, dec_padding_mask):
enc_output = self.encoder(inp, training, enc_padding_mask)
dec_output = self.decoder(tar, enc_output, training, look_ahead_mask, dec_padding_mask)
return self.final_layer(dec_output)

# Example Usage
sample_transformer = Transformer(
num_layers=4,
d_model=128,
num_heads=8,
dff=512,
input_vocab_size=8500,
target_vocab_size=8000,
pe_input=10000,
pe_target=6000
)

sample_input = tf.random.uniform((64, 37), dtype=tf.int64, minval=0, maxval=200)
sample_target = tf.random.uniform((64, 35), dtype=tf.int64, minval=0, maxval=200)
output = sample_transformer(sample_input, sample_target, training=False, enc_padding_mask=None, look_ahead_mask=None, dec_padding_mask=None)
print(output.shape)
```