MID SEMESTER PROJECT PRESENTATION

BY

Komal Reddy (20103005)

Jayanth Reddy (20103010)

Jnana Yasaswini (20103023)

Akhila Banoth (20103052)

DESIGN AND DEVELOPMENT OF ENGLISH TO TELUGU TRANSLATION SYSTEM BASED ON TRANSFORMER WITH DEEP LEARNING

CONTENTS

AIM AND OBJECTIVES

AIM

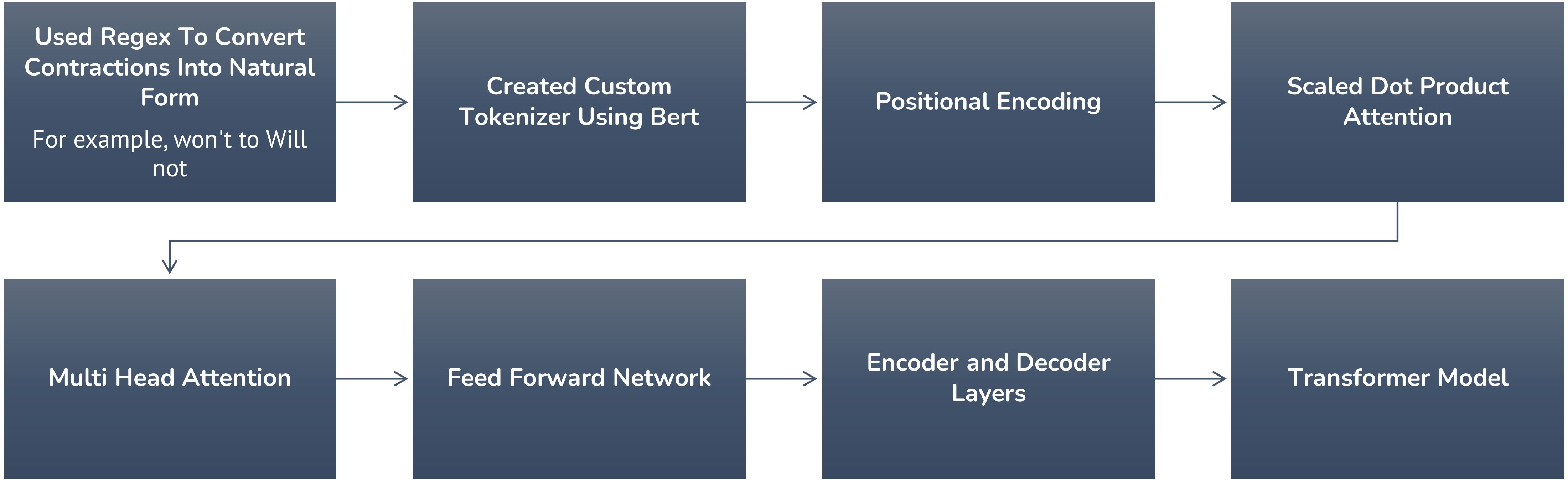
This project aims to create a proficient English to Telugu translation system using transformer architectures in deep learning. By bridging language gaps, it enhances communication and accessibility, offering an advanced tool for seamless translation between English and Telugu.

OBJECTIVES

- Implement a transformer-based deep learning model for text translation.
- Fine-tune the model to handle language-specific nuances and improve performance.
- Implement appropriate evaluation metrics (e.g.,BLEU-score,accuracy)to quantitatively assess the translation quality.
- Create an intuitive and user-friendly interface for users to input English text and receive the corresponding translated Telugu text.

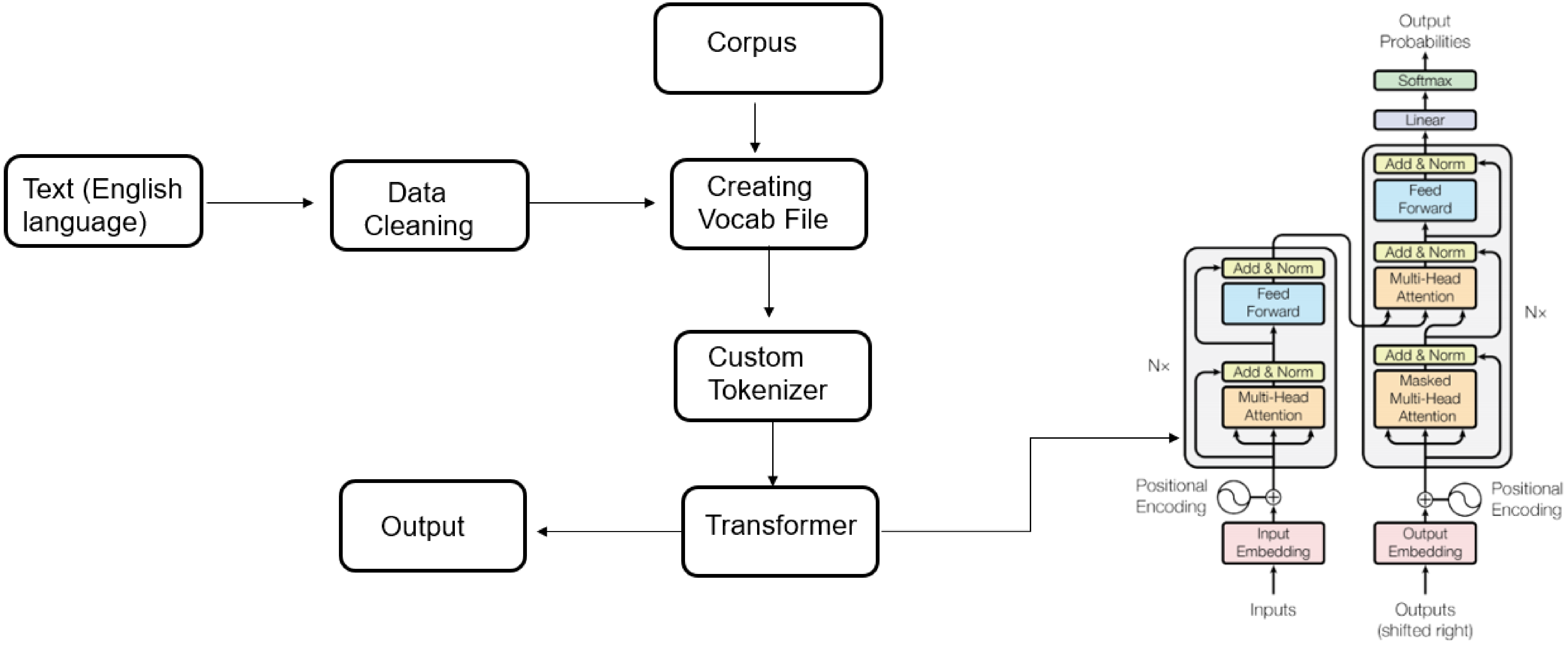
PROGRESS

PROGRESS

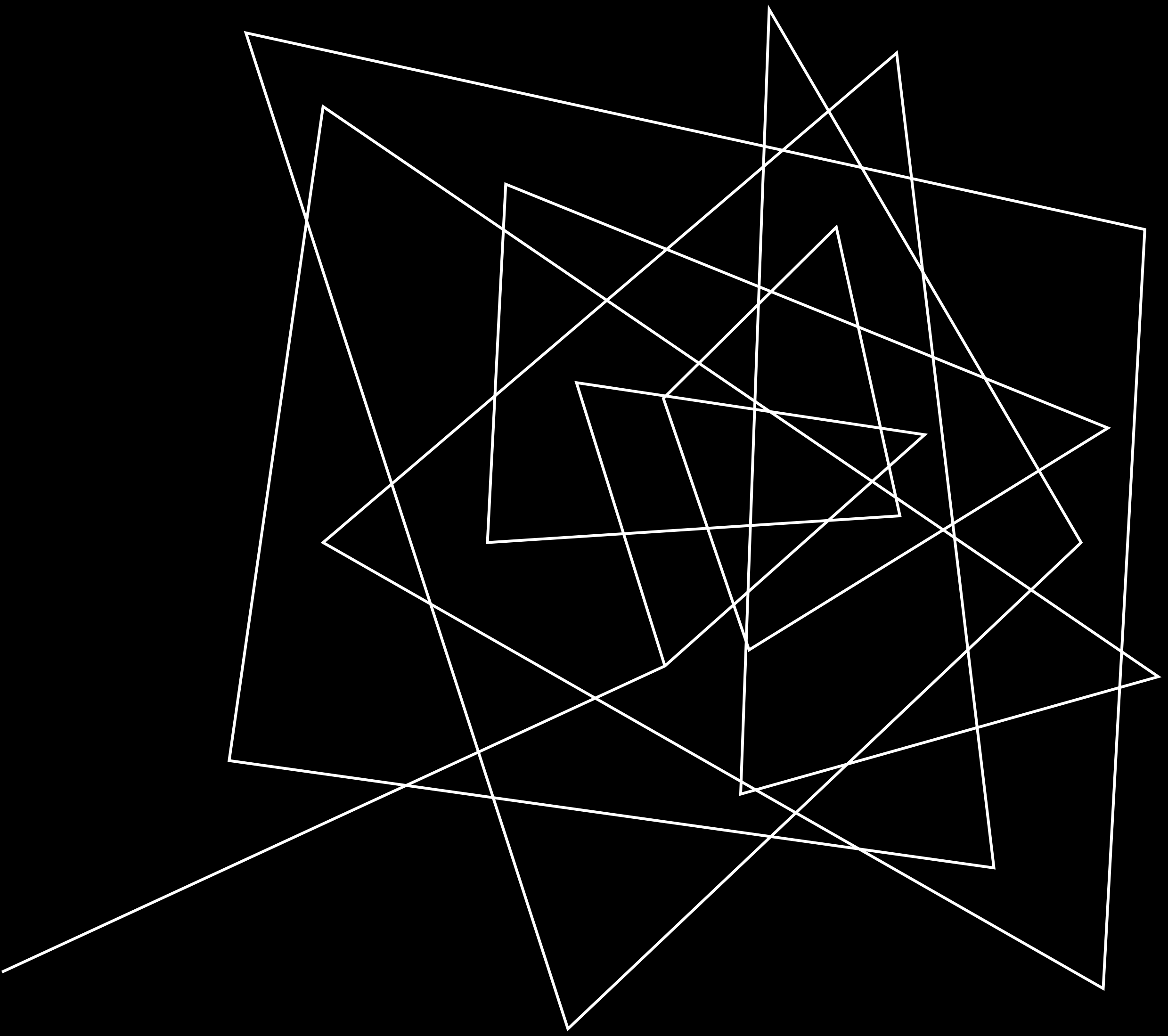

Used Regex To Convert Contractions Into Natural Form
For example, won't to Will not
Created Custom Tokenizer Using Bert
Positional Encoding
Scaled Dot Product Attention
Multi Head Attention
Feed Forward Network
Encoder and Decoder Layers
Transformer Model

ARCHITECTURE AND FLOW DIAGRAM

ARCHITECTURE AND FLOW DIAGRAM

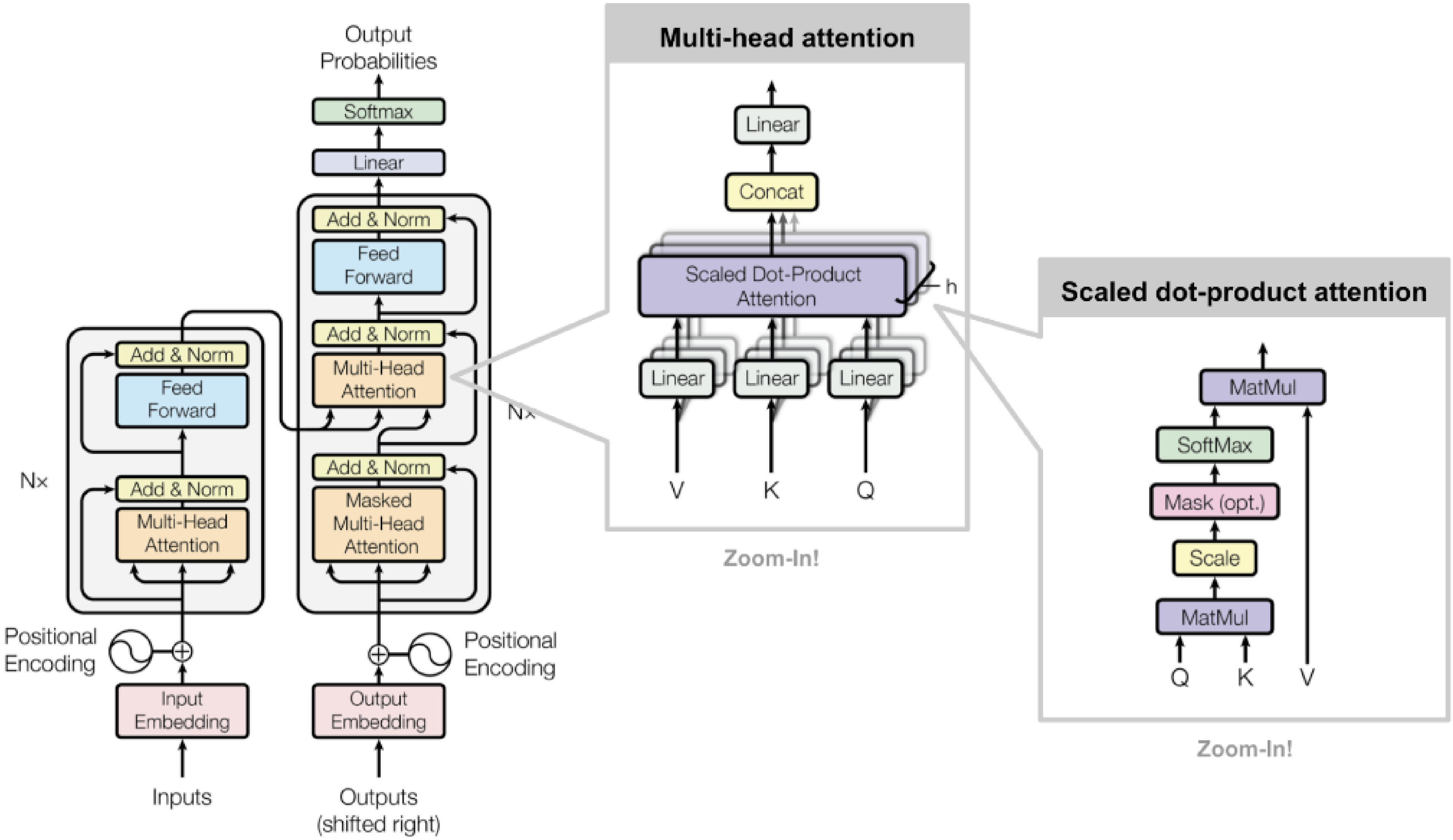


PARTIAL RESULTS



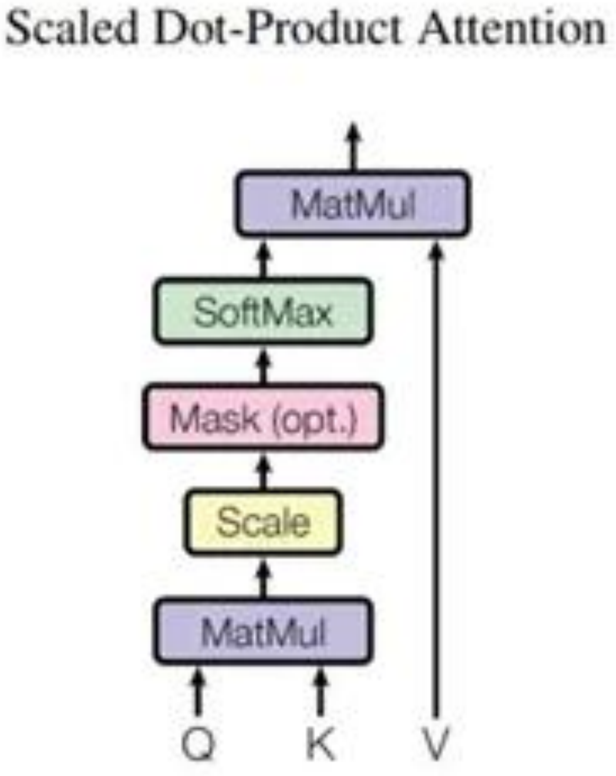
ATTENTION MECHANISM

ATTENTION MECHANISM



SCALED DOT PRODUCT ATTENTION

SCALED DOT PRODUCT ATTENTION

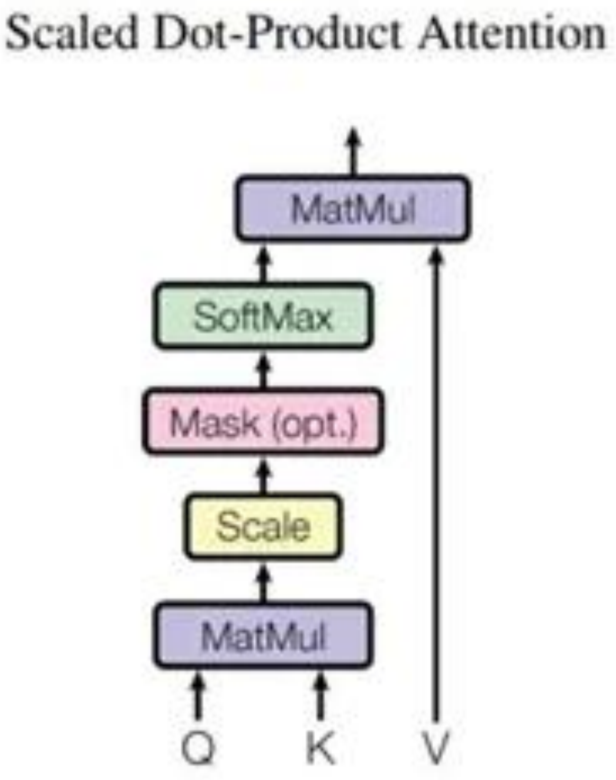


The attention function used by the transformer takes three inputs: Q (query), K (key), V (value). The equation used to calculate the attention weights is:

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

Query vector: Represented by a word vector in the sequence and defines the hidden state of the decoder.
Key vector: Represented by all the words in the sequence and defines the hidden state of the encoder.
Value vector: Represented by all the words in the sequence and defines the attention weights of the encoder hidden states.
The dot-product attention is scaled by a factor of square root of the depth. This is done because for large values of depth, the dot product grows large in magnitude pushing the softmax function where it has small gradients resulting in a very hard softmax.

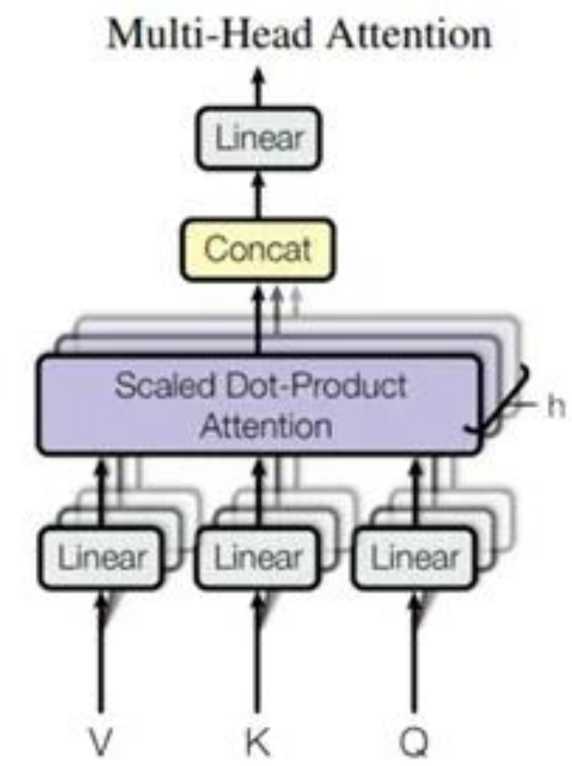
SCALED DOT PRODUCT ATTENTION




```python
def scaled_dot_product_attention(q, k, v, mask):
  """Calculate the attention weights.
  q, k, v must have matching leading dimensions.
  k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
  The mask has different shapes depending on its type(padding or look ahead)
  but it must be broadcastable for addition.

  Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

  Returns:
    output, attention_weights
  """

  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

  # scale matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # add the mask to the scaled tensor.
  if mask is not None:
    scaled_attention_logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k) so that the scores
  # add up to 1.
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (..., seq_len_q, seq_len_k)

  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

  return output, attention_weights
```

MULTI HEAD ATTENTION

MULTI HEAD ATTENTION



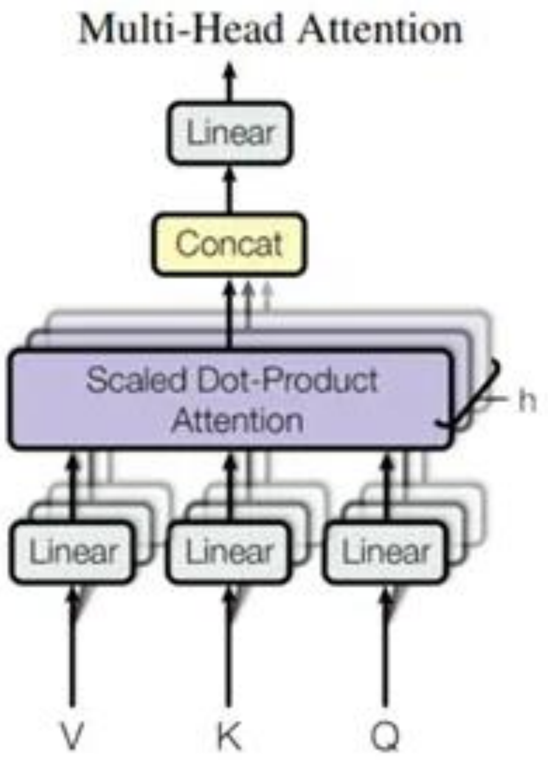
Multi-head attention consists of four parts: **Linear layers and split into heads, Scaled dot-product attention, Concatenation of heads, Final linear layer** . Each multi-head attention block gets three inputs**; Q (query), K (key), V (value).** These are put through linear (Dense) layers and split up into multiple heads.

The scaled_dot_product_attention defined above is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated and put through a final Dense layer.

Instead of one single attention head, Q, K, and V are split into multiple heads because it allows the model to jointly attend to information from different representation subspaces at different positions. After the split each head has a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.

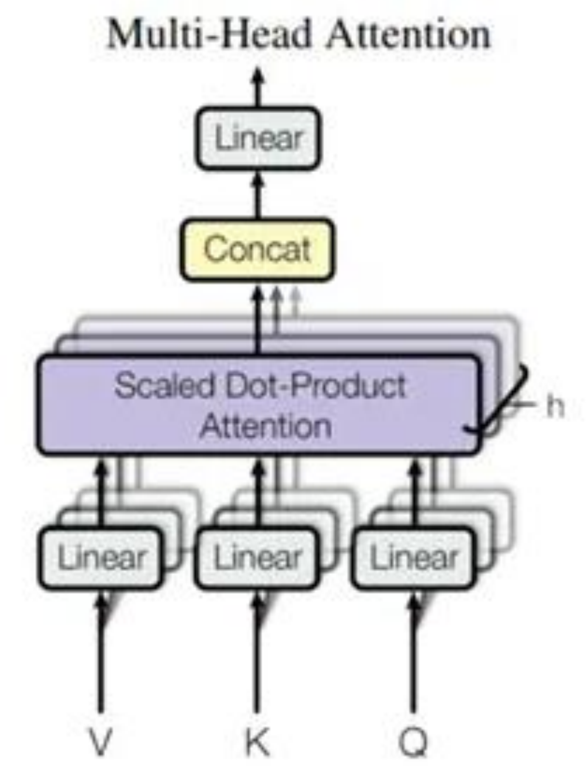
The output represents the multiplication of the attention weights and the V (value) vector. This ensures that the words we want to focus on are kept as-is and the irrelevant words are flushed out.

MULTI HEAD ATTENTION

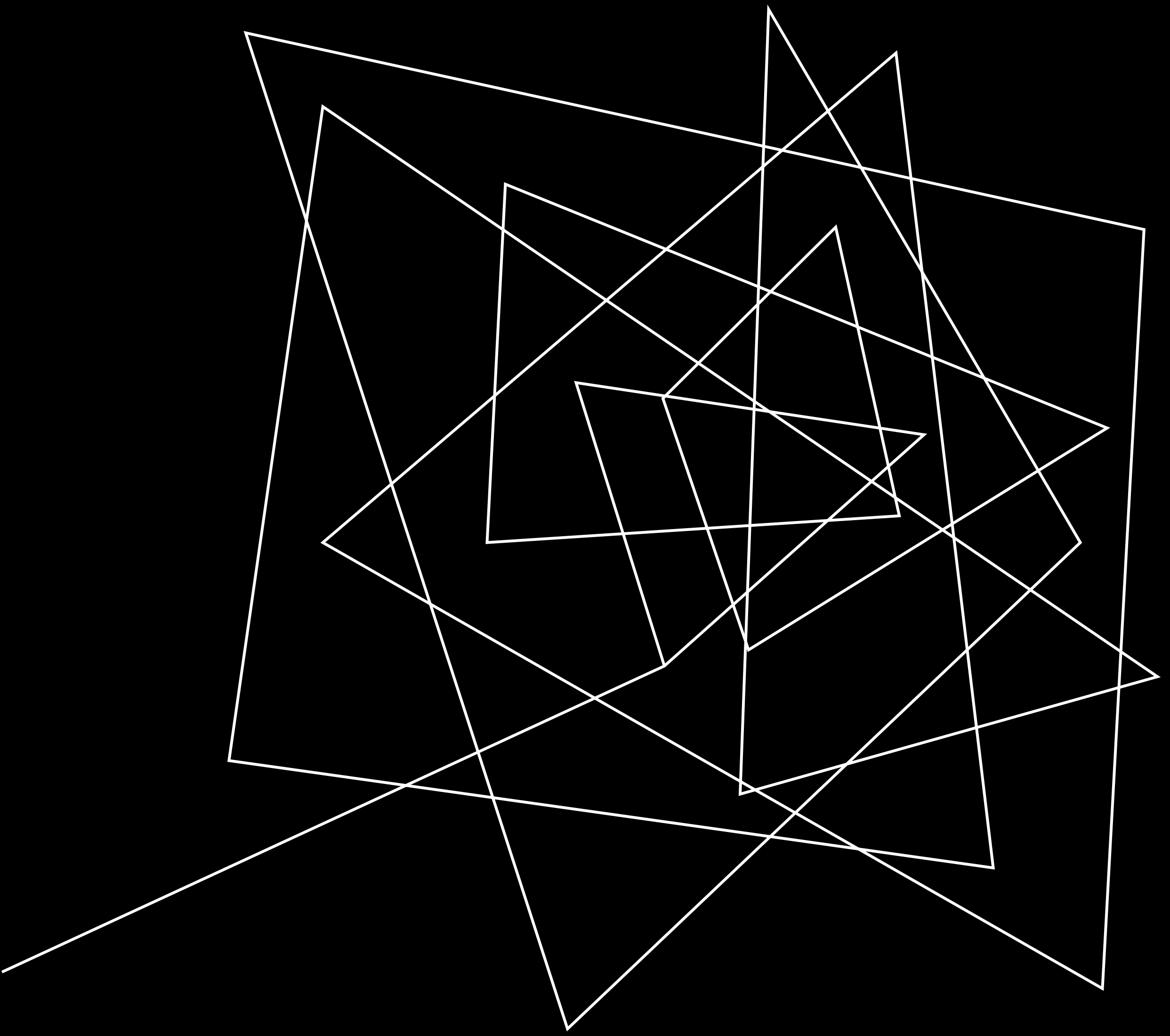



```python
class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.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):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, num_heads, seq_len, depth)
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])
```

MULTI HEAD ATTENTION




```python
def call(self, v, k, q, mask):
  batch_size = tf.shape(q)[0]

  q = self.wq(q)  # (batch_size, seq_len, d_model)
  k = self.wk(k)  # (batch_size, seq_len, d_model)
  v = self.wv(v)  # (batch_size, seq_len, d_model)

  q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth)
  k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth)
  v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth)

  # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
  # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
  scaled_attention, attention_weights = scaled_dot_product_attention(
      q, k, v, mask)

  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_size, seq_len_q, num_heads, depth)

  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))  # (batch_size, seq_len_q, d_model)

  output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

  return output, attention_weights
```

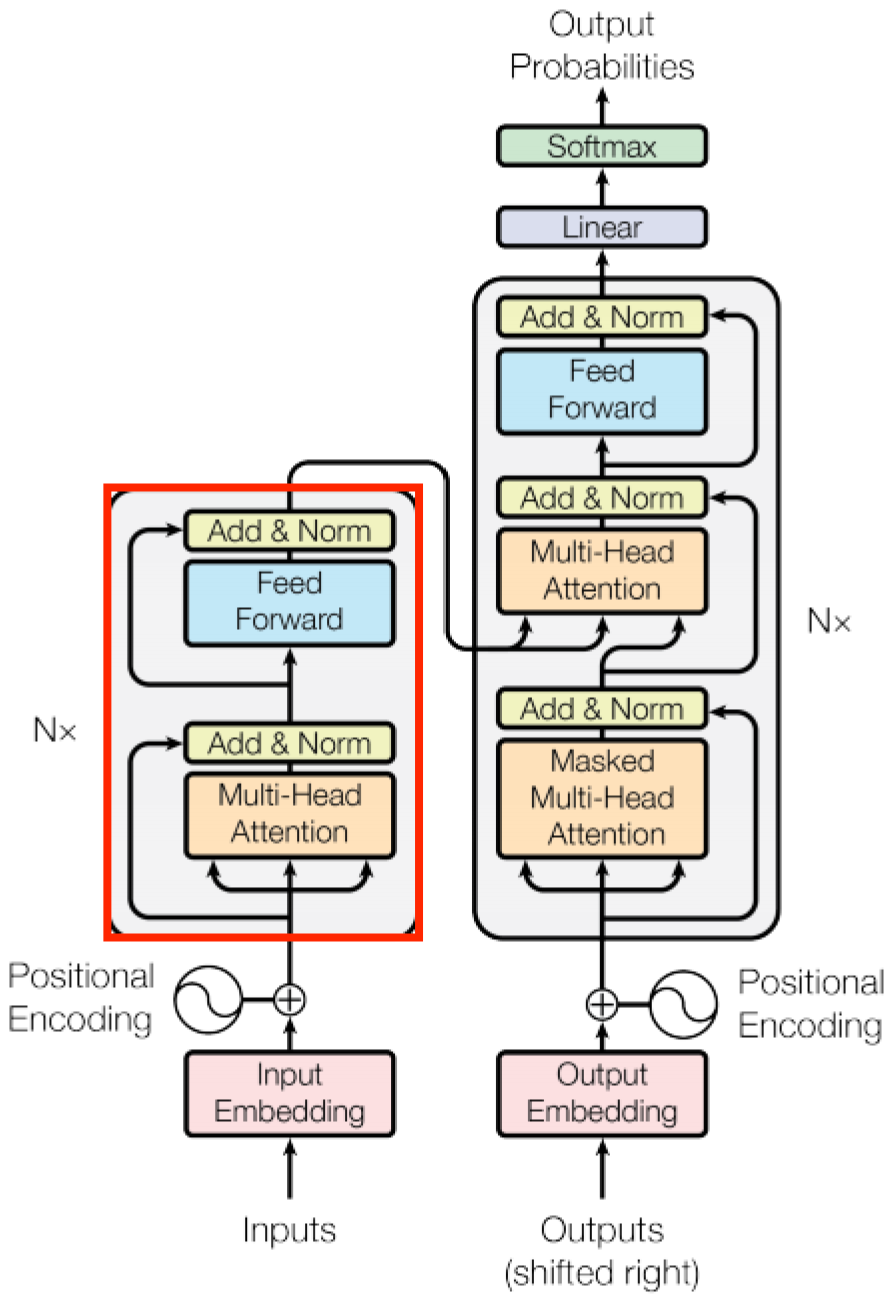


ENCODER

ENCODER LAYER

The encoder contains a stack of N encoder layers. Where each contains a GlobalSelfAttention and FeedForward layer

The encoder takes each word in the input sentence, process it to an intermediate representation and compares it with all the other words in the input sentence. The result of those comparisons is an attention score that evaluates the contribution of each word in the sentence to the key word. The attention scores are then used as weights for words' representations that are fed the fully-connected network that generates a new representation for the key word. It does so for all the words in the sentence and transfers the new representation to the decoder that by this information can have all the dependencies that it needs to build the predictions.



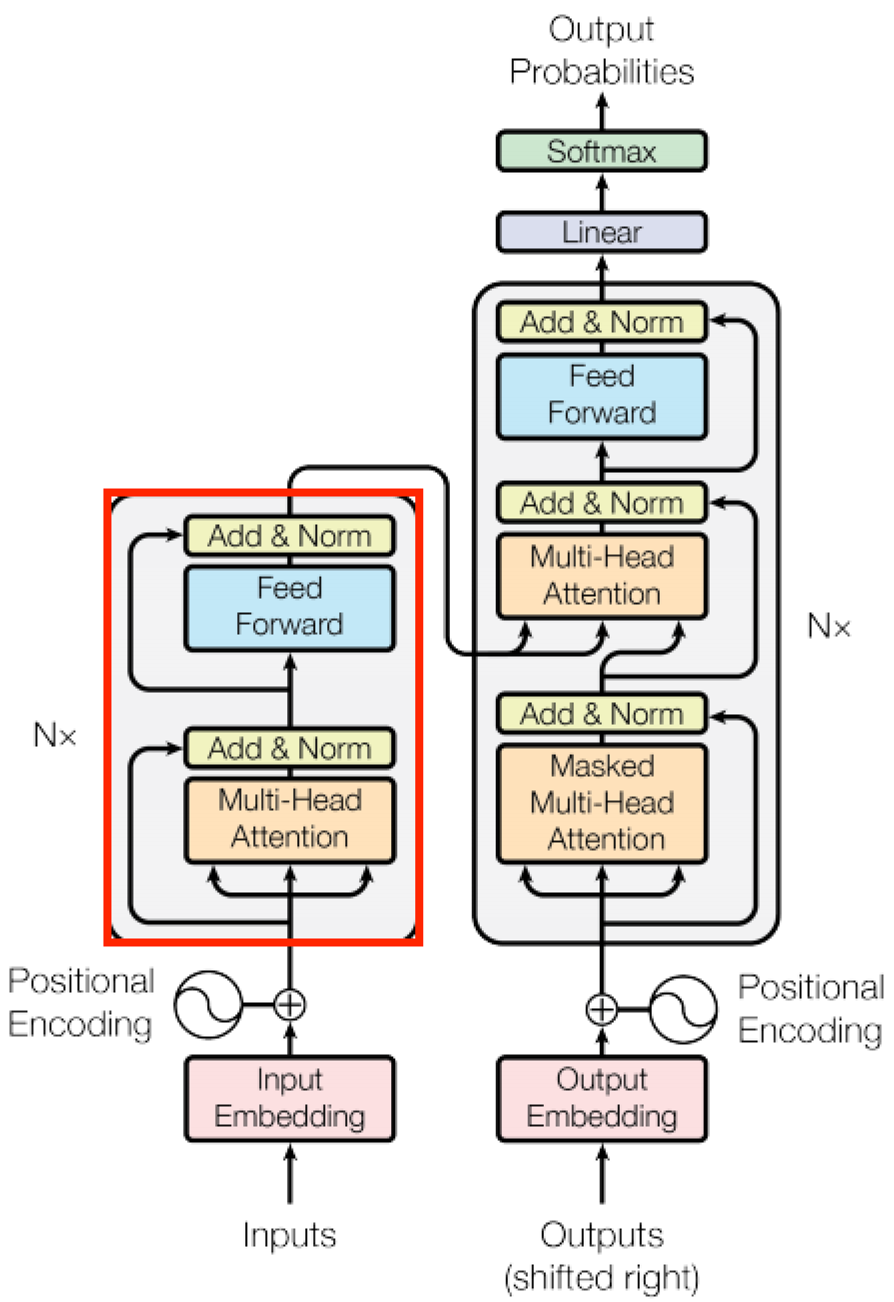
ENCODER LAYER

```python
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, 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(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)

  def call(self, x, training, mask):

    attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_model)
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model)

    ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_model)

    return out2
```

```python
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

```python
sample_ffn = point_wise_feed_forward_network(512, 2048)
sample_ffn(tf.random.uniform((64, 50, 512))).shape
```

```
TensorShape([64, 50, 512])
```

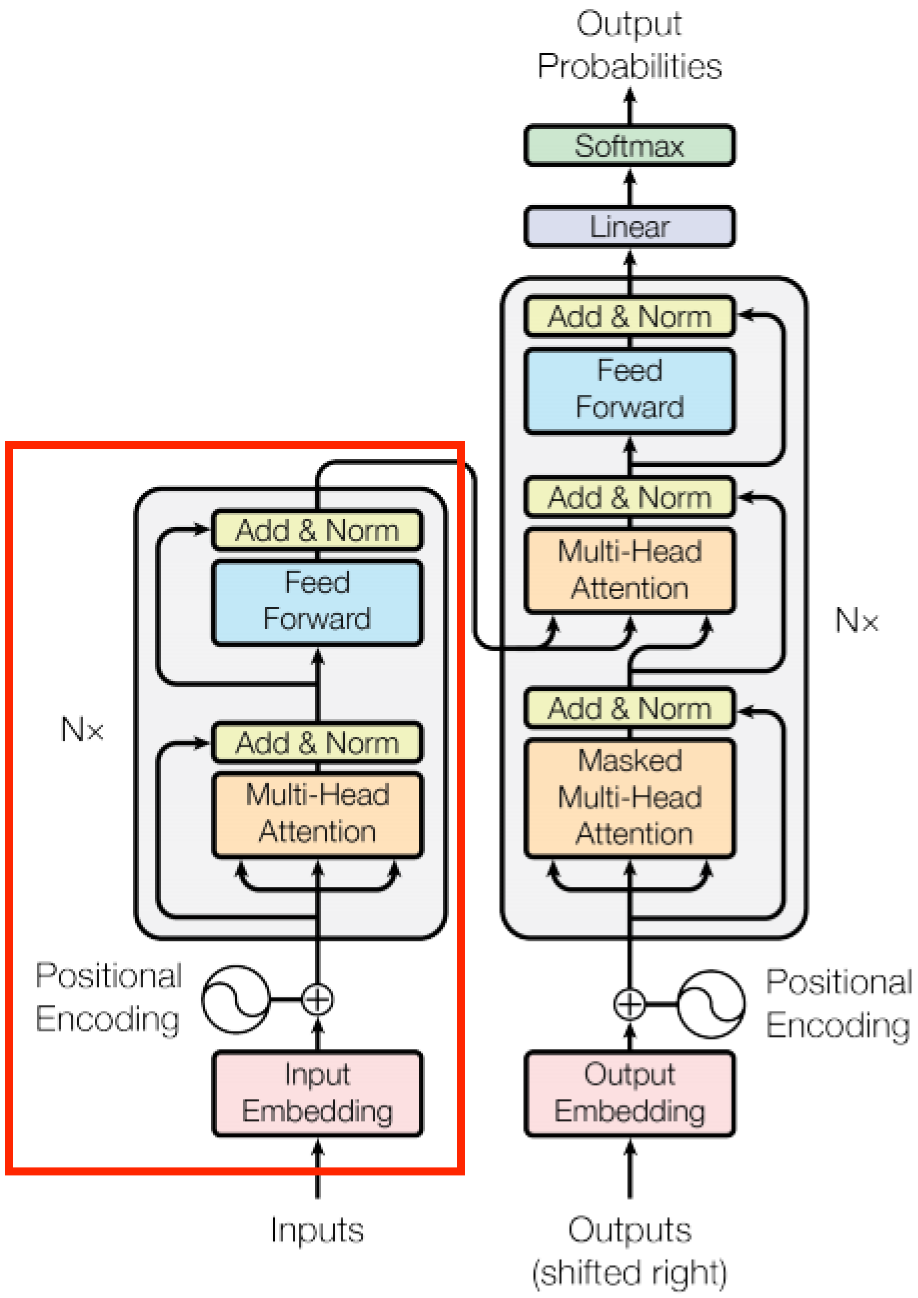


ENCODER

The encoder consists of:

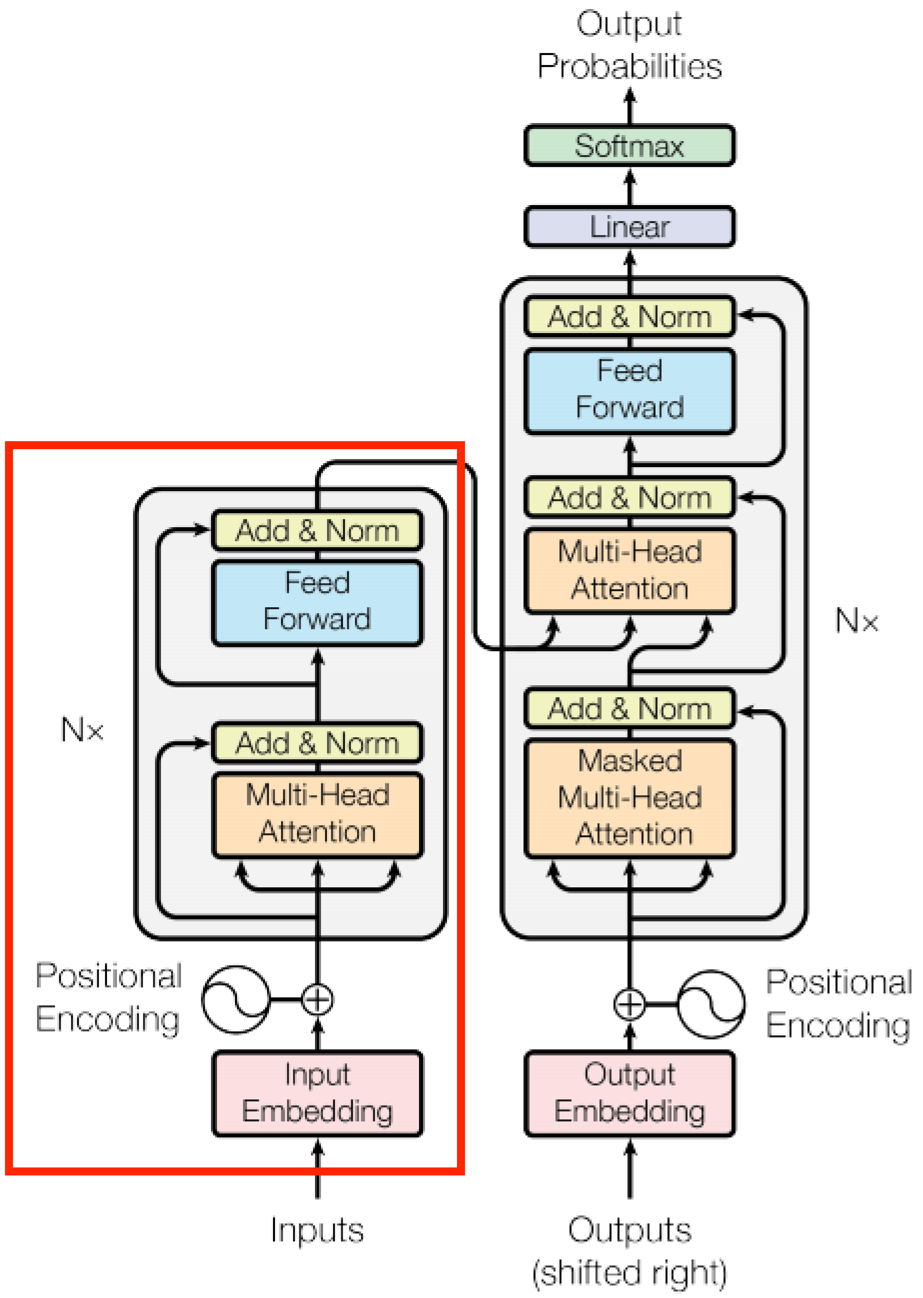
- A PositionalEmbedding layer at the input.
- A stack of EncoderLayer layers.

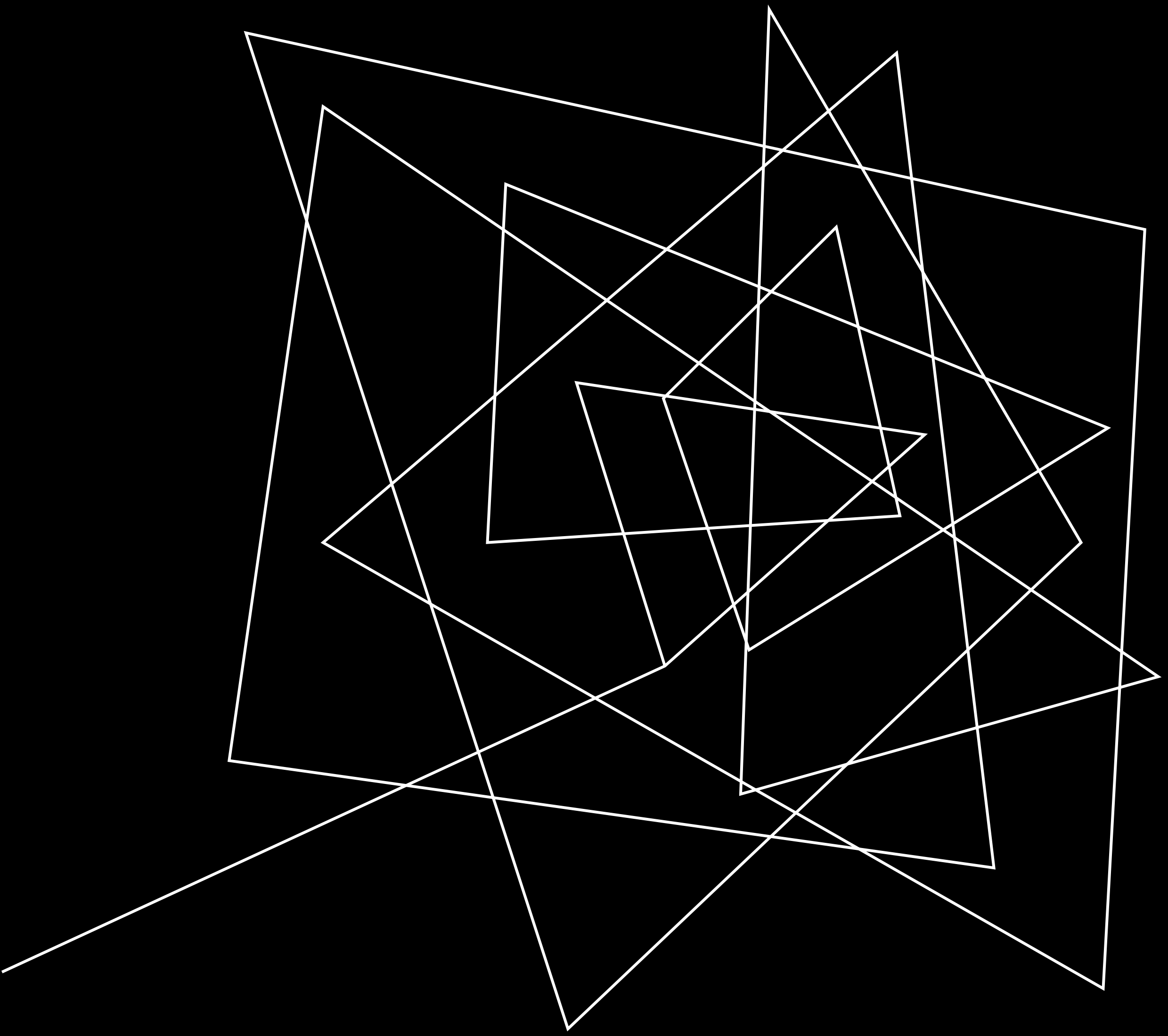
The encoder receives the embedding vectors as a list of vectors, each of 512 (can be tuned as a hyper-parameter) size dimension. Both the encoder and the decoder add a positional encoding (that will be explained later) to their input. Both also use a bypass that is called a residual connection followed by an addition of the original input of the sub-layer and another normalization layer (which is also known as a batch normalization)



ENCODER

```python
class Encoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               maximum_position_encoding, 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,
                                            self.d_model)

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

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

  def call(self, x, training, mask):

    seq_len = tf.shape(x)[1]

    # adding embedding and position encoding.
    x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
    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  # (batch_size, input_seq_len, d_model)
```





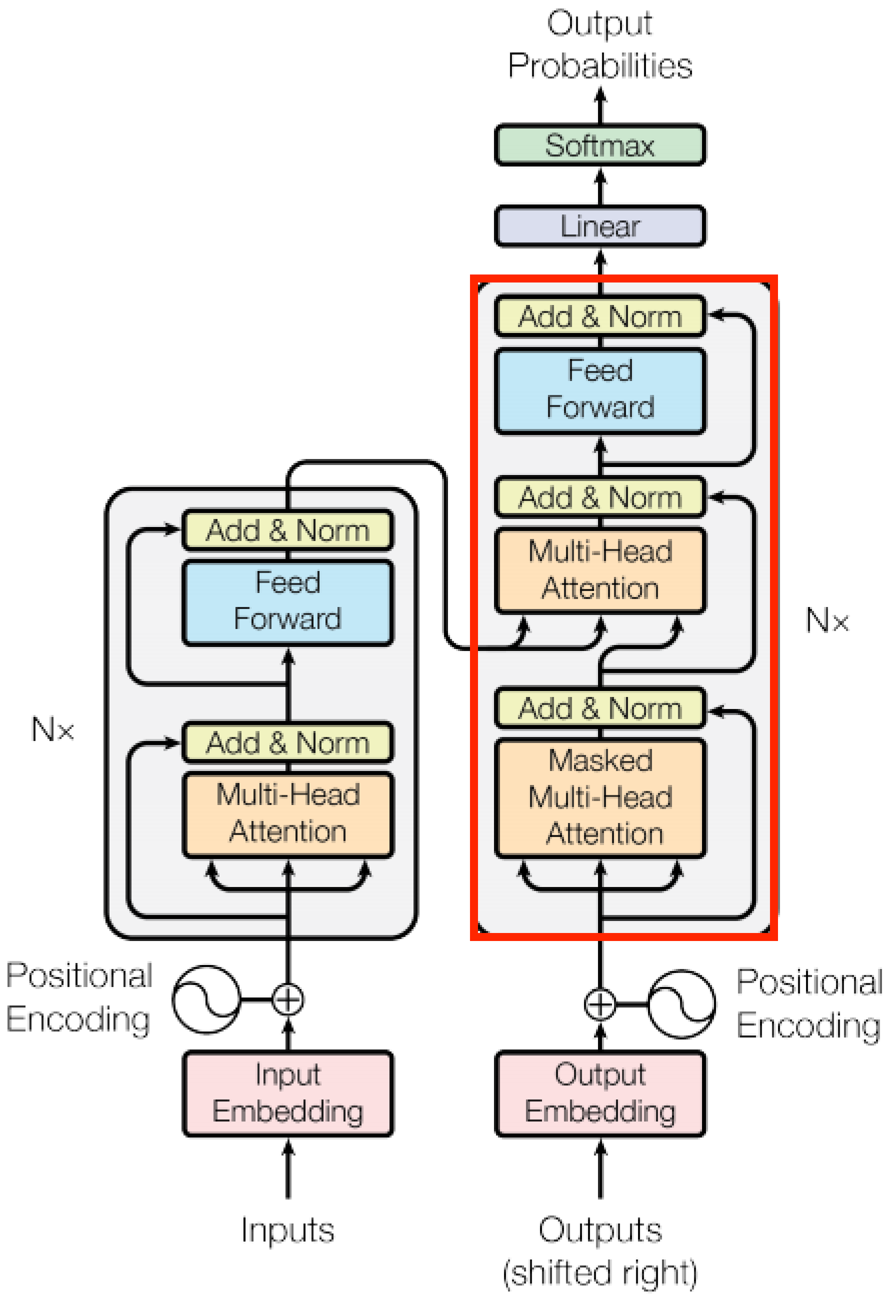
DECODER

DECODER LAYER

The decoder's stack is slightly more complex, with each DecoderLayer containing a CausalSelfAttention, a CrossAttention, and a FeedForward layer

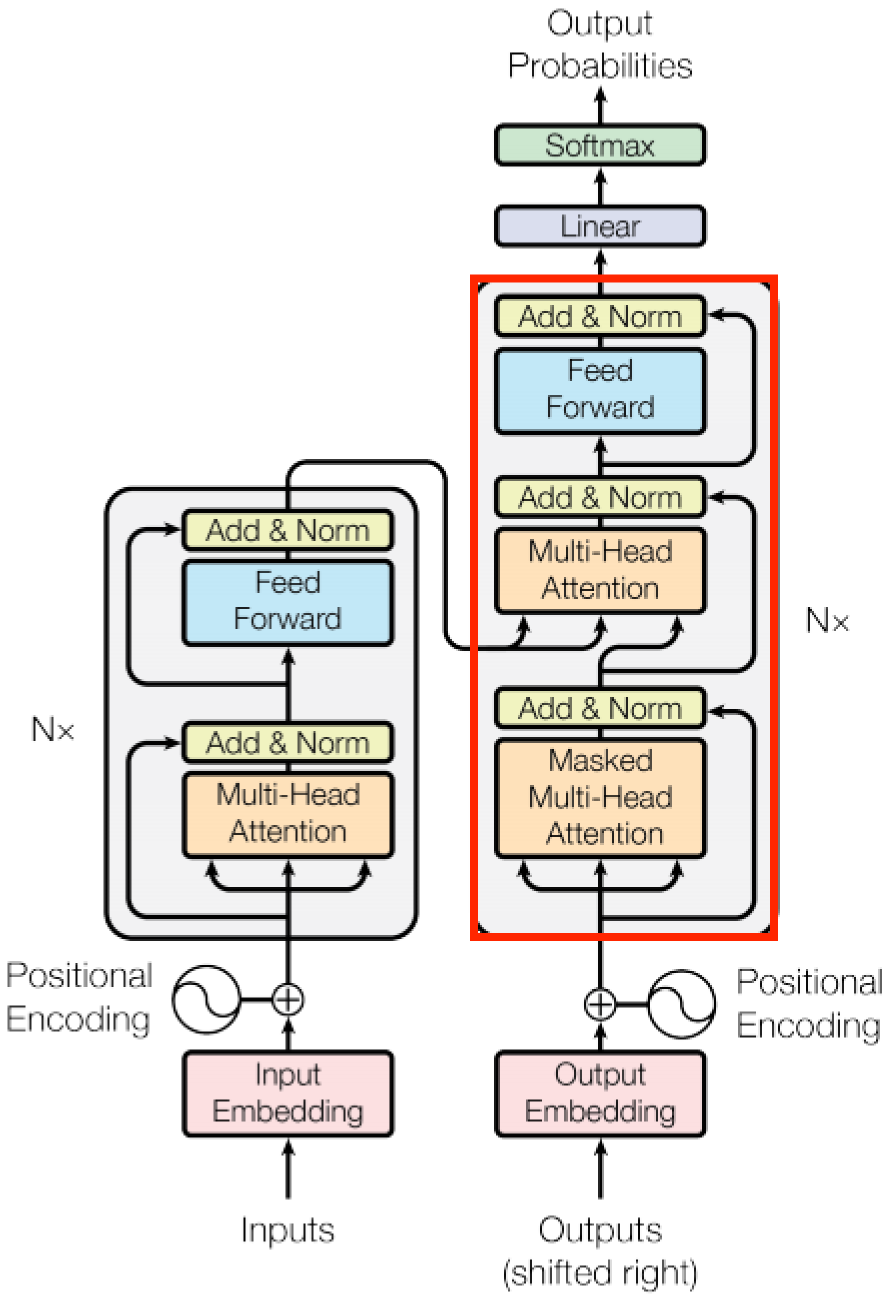
Similar to the Encoder, the Decoder consists of a PositionalEmbedding, and a stack of DecoderLayers

Unlike the encoder, the decoder uses an addition to the Multi-head attention that is called masking. This operation is intended to prevent exposing posterior information from the decoder. It means that in the training level the decoder doesn’t get access to tokens in the target sentence that will reveal the correct answer and will disrupt the learning procedure. It’s really important part in the decoder because if we will not use the masking the model will not learn anything and will just repeat the target sentence.



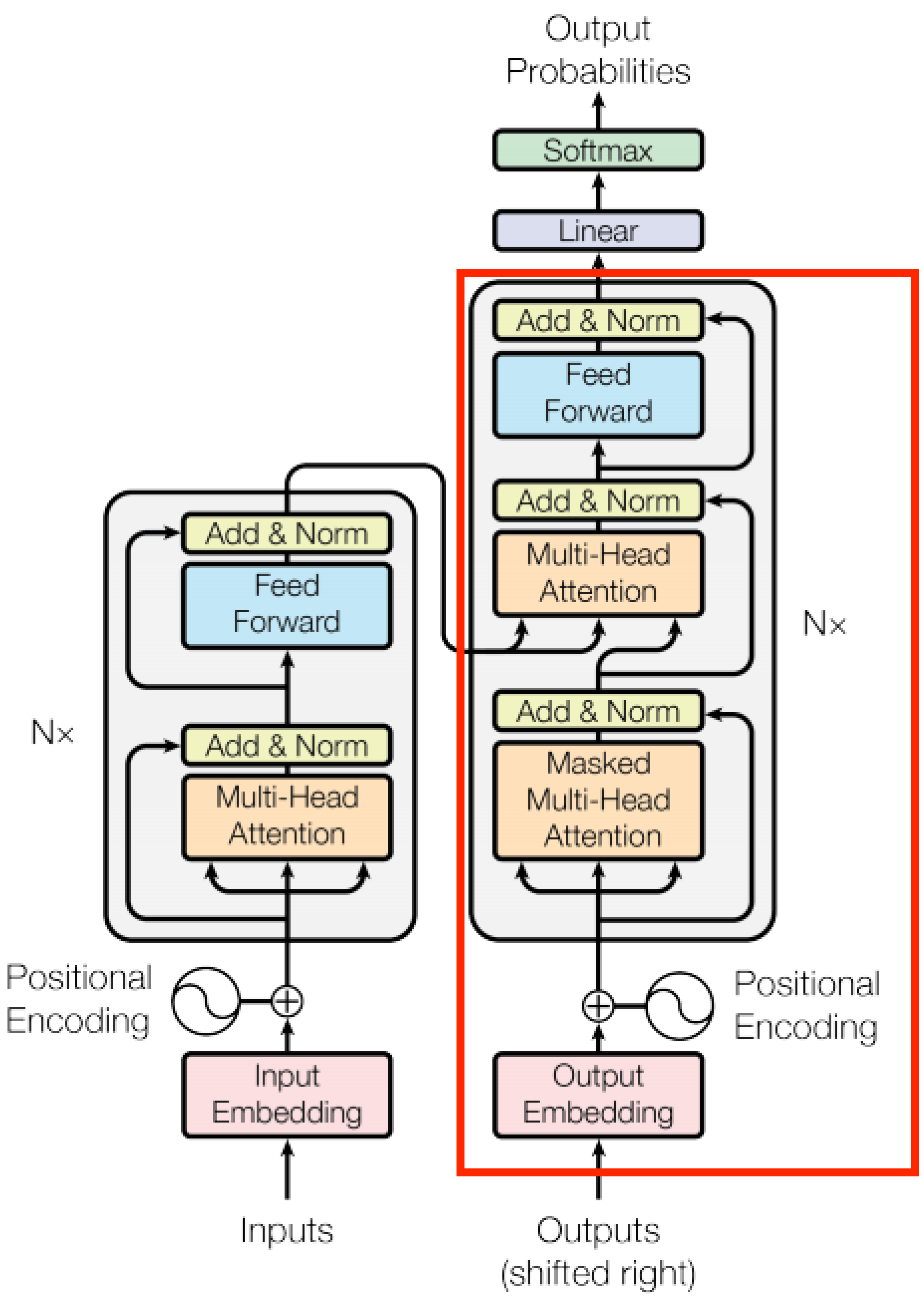
DECODER LAYER

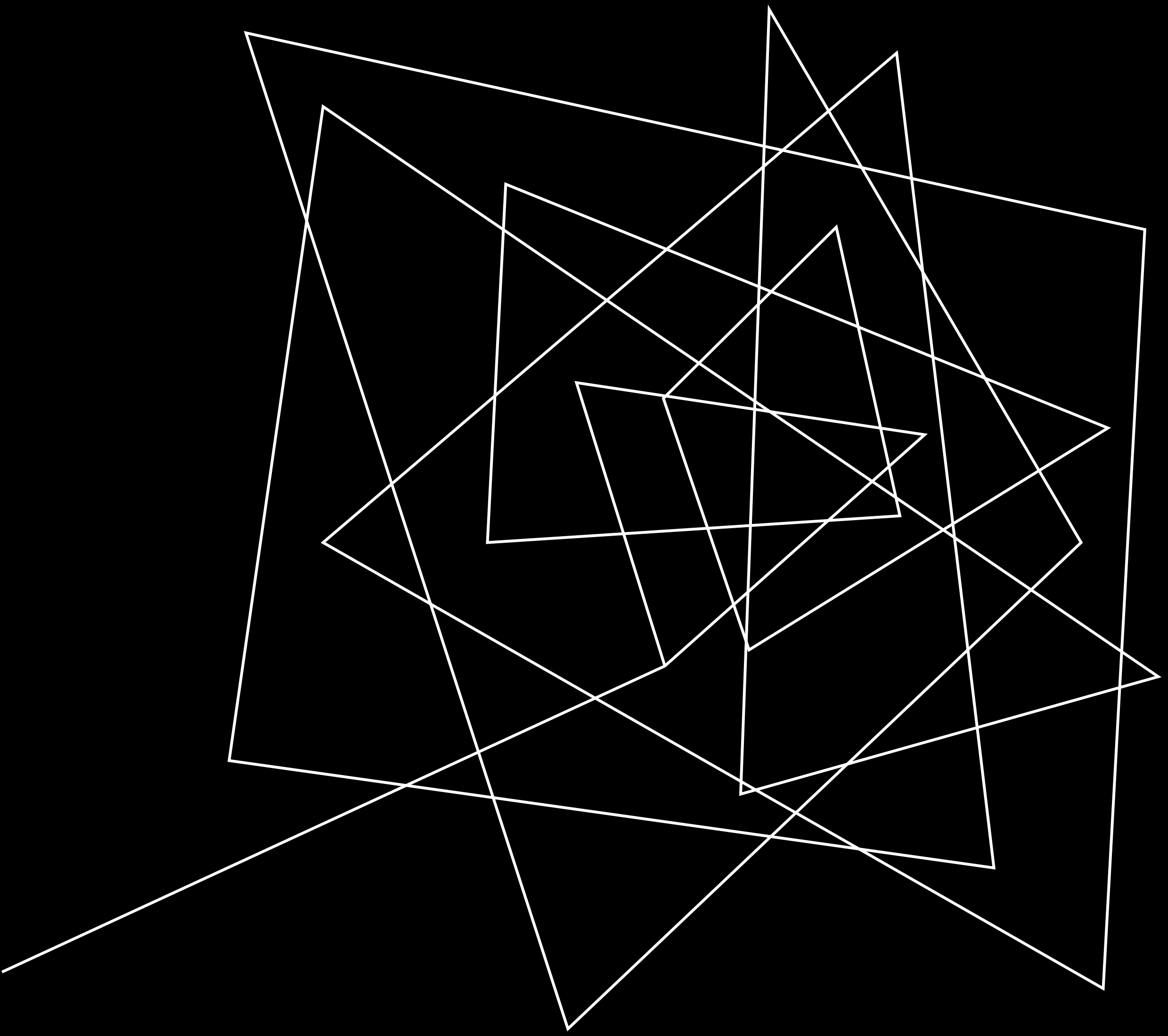
```
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(DecoderLayer, self).__init__()

    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(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
    self.dropout3 = tf.keras.layers.Dropout(rate)

  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):
    # enc_output.shape == (batch_size, input_seq_len, d_model)

    attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_size, target_seq_len, d_model)
    attn1 = self.dropout1(attn1, training=training)
    out1 = self.layernorm1(attn1 + x)

    attn2, attn_weights_block2 = self.mha2(
        enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len, d_model)
    attn2 = self.dropout2(attn2, training=training)
    out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)

    ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
    ffn_output = self.dropout3(ffn_output, training=training)
    out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_model)

    return out3, attn_weights_block1, attn_weights_block2
```



DECODER

```python
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, 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, rate)
                       for _ in range(num_layers)]
    self.dropout = tf.keras.layers.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)  # (batch_size, target_seq_len, d_model)
    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, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                             look_ahead_mask, padding_mask)

      attention_weights[f'decoder_layer{i+1}_block1'] = block1
      attention_weights[f'decoder_layer{i+1}_block2'] = block2

    # x.shape == (batch_size, target_seq_len, d_model)
    return x, attention_weights
```





TRANSFORMER

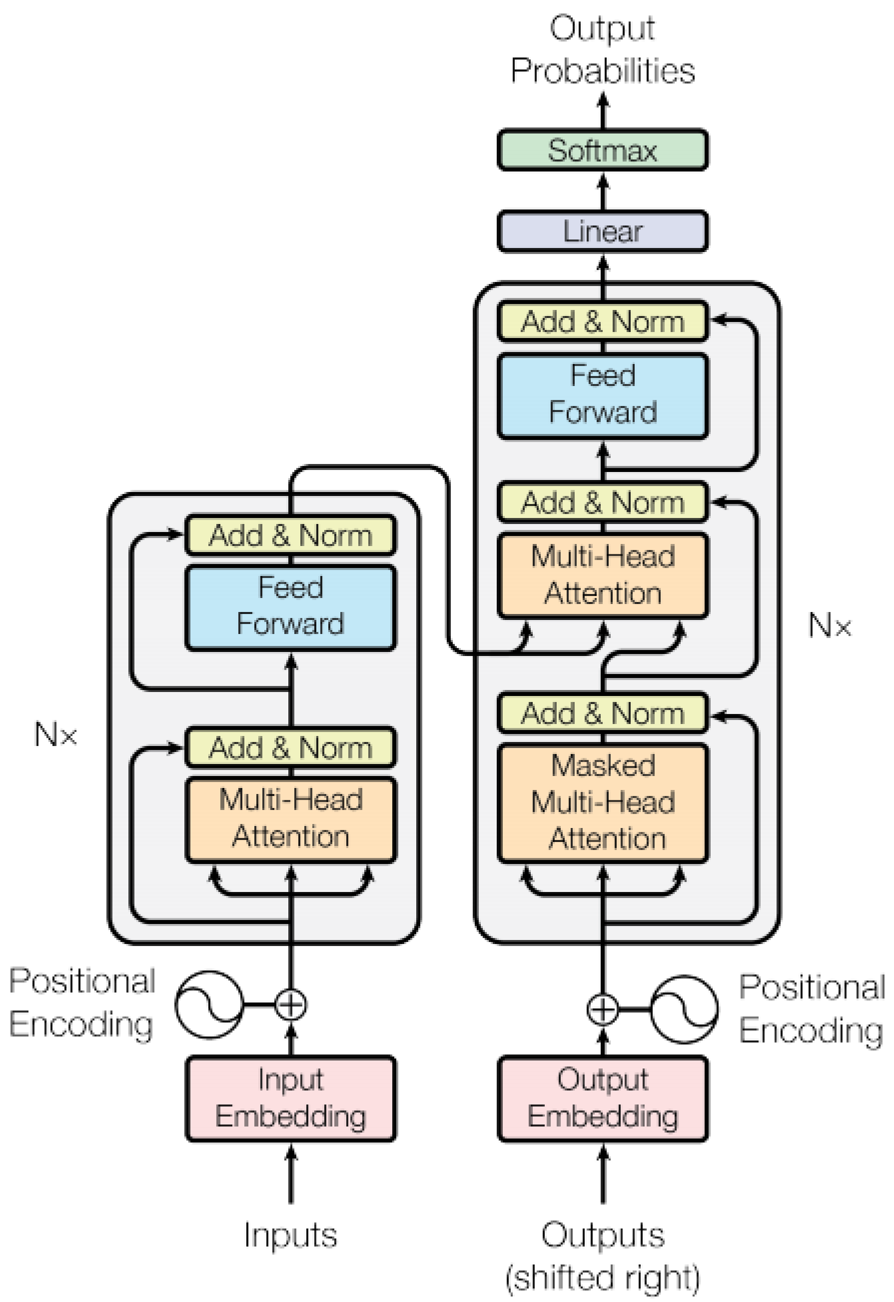
TRANSFORMER

We now have Encoder and Decoder. To complete the Transformer model, we need to put them together and add a final linear (Dense) layer which converts the resulting vector at each location into output token probabilities.

The output of the decoder is the input to this final linear layer.

```
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, rate=0.1):
    super(Transformer, self).__init__()

    self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                             input_vocab_size, pe_input, rate)

    self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                           target_vocab_size, pe_target, 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.tokenizer(inp, training, enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

    # dec_output.shape == (batch_size, tar_seq_len, d_model)
    dec_output, attention_weights = self.decoder(
        tar, enc_output, training, look_ahead_mask, dec_padding_mask)

    final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, target_vocab_size)

    return final_output, attention_weights
```

The transformer



TIMELINE

TIMELINE

Oct. 2023

Dataset Preparation

Data Preprocessing

Nov. 2023

Creating Vocab File

Custom tokenizer

Dec. 2023

Building Attention Mechanism

Jan. 2024

Encoder Layer

Feb. 2024

Decoder Layer

Mar. 2024

Transformer

Apr. 2024

Training and testing

May. 2024

Deployment

CONCLUSION AND FUTURE WORK

CONCLUSION AND FUTURE WORK

In conclusion, our team's next crucial step involves training the model and evaluating its performance using the BLEU score metric. This evaluation will provide valuable insights into the accuracy and efficacy of our translation system. Upon achieving satisfactory results, we will proceed with the deployment phase, ensuring widespread accessibility and usability of our advanced English to Telugu translation tool.

WE

Komal Reddy (20103005)

Jayanth Reddy (20103010)

Jnana Yasaswini (20103023)

Akhila Banoth (20103052)

AS A TEAM

THANK YOU