INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY BANGALORE

NATURAL LANGUAGE PROCESSING AI-829

# MATHIQ

Aditi Singh
MT2023085
aditi.singh@iiitb.ac.in

Parag Dutt Sharma
MT2023095
parag.sharma@iiitb.ac.in

# Index

# MANDATE 2 RECAP

The process of lexical preprocessing in natural language processing (NLP) involves several essential steps to prepare textual data for analysis by machine learning models which was done in the previous mandate.

**Lowercasing :** All characters are converted to lowercase to ensure uniformity and prevent the model from treating words differently based on case.

**Padding spaces and removing extra white spaces :** Proper spacing is maintained between words to facilitate tokenization, and extra white spaces are removed to enhance data cleanliness.

**Tokenization :** The text is broken down into smaller units, typically words or subwords, to provide the model with discrete elements for analysis.

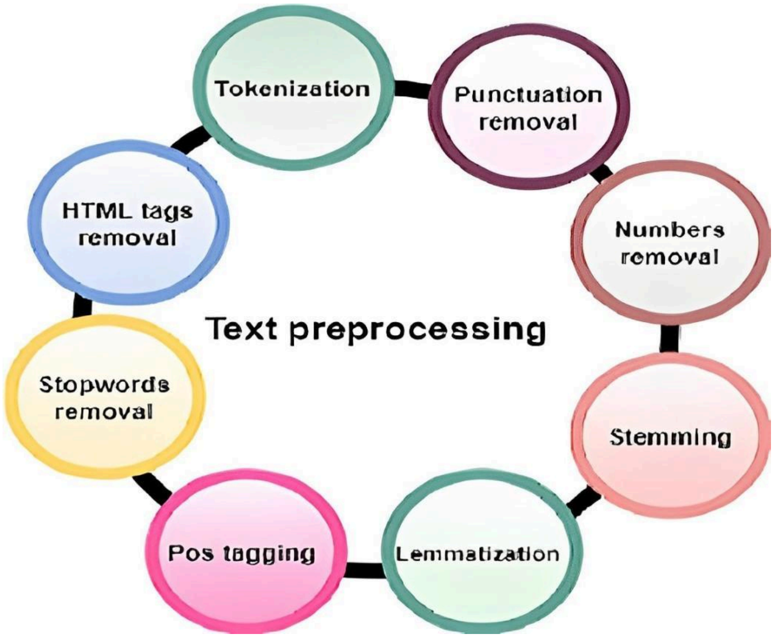
**Converting to tensor sequences :** Text data is converted into numerical representations using tensors, enabling efficient processing by machine learning algorithms.

**Zero padding for equalizing sequence lengths :** To address challenges posed by varying sequence lengths, zeros are added to shorter sequences to ensure equal lengths, facilitating consistent input dimensions for neural networks.

# MANDATE 3

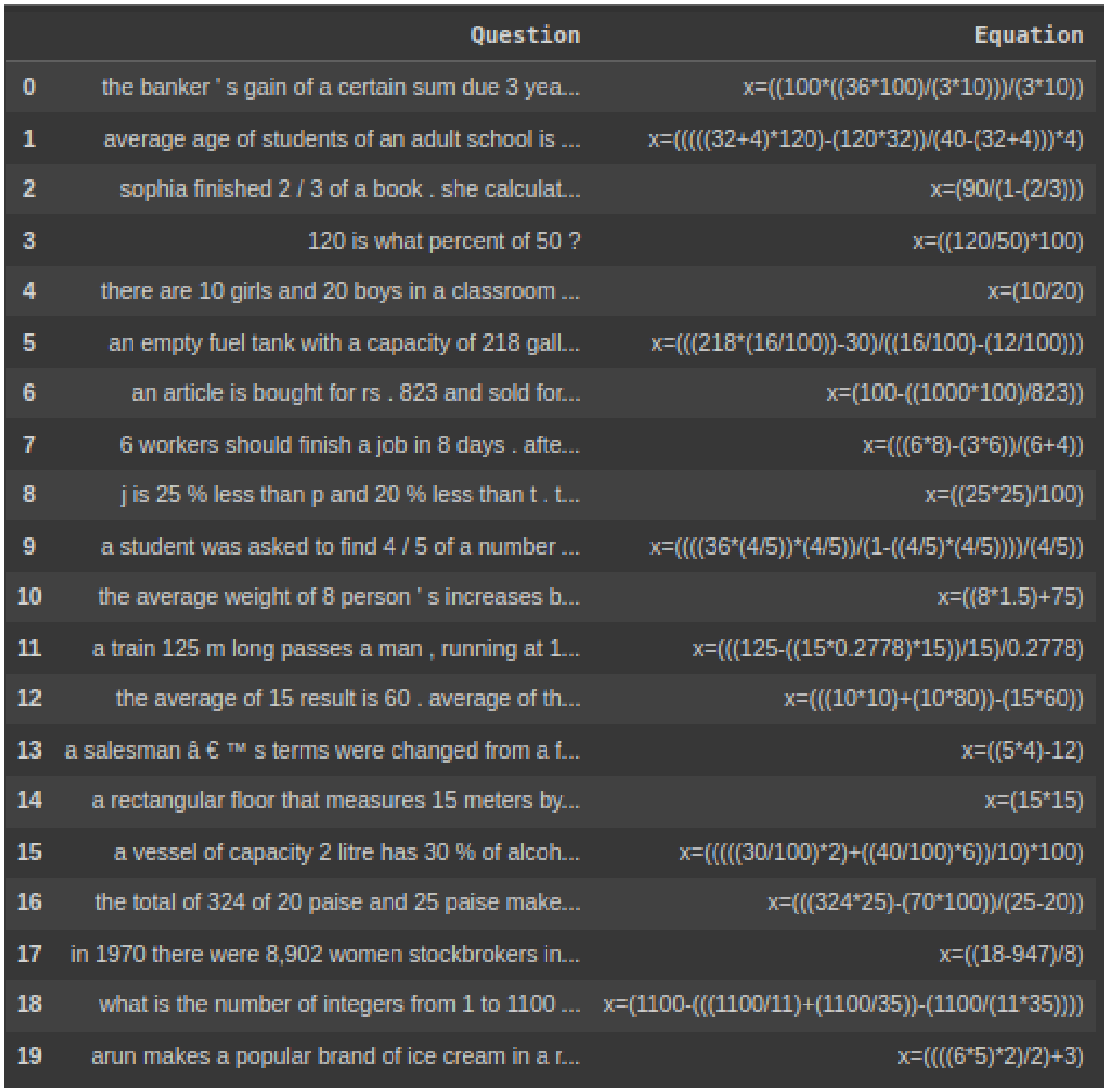
## Introduction

The integration of shallow parsing and POS tagging within Natural Language Processing (NLP) systems plays a pivotal role in establishing the necessary grammatical structures within problem statements. By harnessing techniques such as Hidden Markov Models (HMMs) alongside the Viterbi Heuristic, these systems effectively model word sequences, thereby enhancing their comprehension of mathematical terms and accommodating variations in problem presentation. In the context of solving math word problems, our project will implement a Transformer-based model, which capitalizes on advanced NLP methodologies like Syntactic Preprocessing, Embedding, and Sequence-to-Sequence modeling. The implementation and integration of Shallow Parsing or POS tagging could augment the model's performance by providing additional linguistic context. Furthermore, the mention of models like BERT, GPT, and LAMA 2 suggests potential avenues for further enhancement through the incorporation of pre-trained language models.



## Dataset

The dataset comprises around 30,000 mathematical word problems, predominantly featuring algebraic questions and quantitative aptitude queries. These scenarios are accompanied by their respective equations, where the imaginary variable 'x' symbolizes the sought-after value.



|  | Question | Equation |
|---|---|---|
| 0 | the banker ' s gain of a certain sum due 3 yea... | x=((100*((36*100)/(3*10)))/(3*10)) |
| 1 | average age of students of an adult school is ... | x=(((((32+4)*120)-(120*32))/(40-(32+4)))*4) |
| 2 | sophia finished 2 / 3 of a book . she calculat... | x=(90/(1-(2/3))) |
| 3 | 120 is what percent of 50 ? | x=((120/50)*100) |
| 4 | there are 10 girls and 20 boys in a classroom ... | x=(10/20) |
| 5 | an empty fuel tank with a capacity of 218 gall... | x=(((218*(16/100))-30)/((16/100)-(12/100))) |
| 6 | an article is bought for rs . 823 and sold for... | x=(100-((1000*100)/823)) |
| 7 | 6 workers should finish a job in 8 days . afte... | x=(((6*8)-(3*6))/(6+4)) |
| 8 | j is 25 % less than p and 20 % less than t . t... | x=((25*25)/100) |
| 9 | a student was asked to find 4 / 5 of a number ... | x=((((36*(4/5))*(4/5))/(1-((4/5)*(4/5))))/(4/5)) |
| 10 | the average weight of 8 person ' s increases b... | x=((8*1.5)+75) |
| 11 | a train 125 m long passes a man , running at 1... | x=(((125-((15*0.2778)*15))/15)/0.2778) |
| 12 | the average of 15 result is 60 . average of th... | x=(((10*10)+(10*80))-(15*60)) |
| 13 | a salesman â € ™ s terms were changed from a f... | x=((5*4)-12) |
| 14 | a rectangular floor that measures 15 meters by... | x=(15*15) |
| 15 | a vessel of capacity 2 litre has 30 % of alcoh... | x=(((((30/100)*2)+((40/100)*6))/10)*100) |
| 16 | the total of 324 of 20 paise and 25 paise make... | x=(((324*25)-(70*100))/(25-20)) |
| 17 | in 1970 there were 8,902 women stockbrokers in... | x=((18-947)/8) |
| 18 | what is the number of integers from 1 to 1100 ... | x=(1100-(((1100/11)+(1100/35))-(1100/(11*35)))) |
| 19 | arun makes a popular brand of ice cream in a r... | x=((((6*5)*2)/2)+3) |

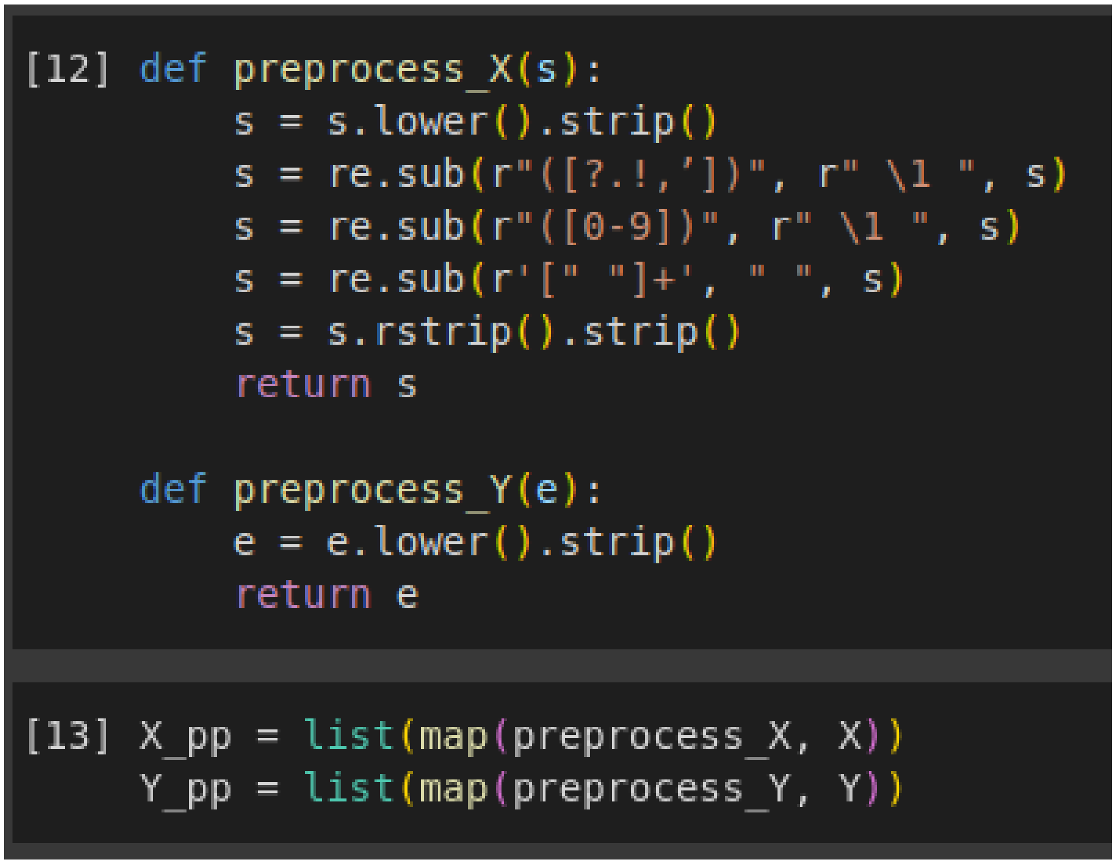
## Syntactic Preprocessing

- **Lowercase Alphabets:**

  Lowercasing is a fundamental step in NLP data preprocessing, ensuring uniformity by converting all characters to lowercase. This prevents the model from treating the same word differently based on its case, improving overall consistency and generalization.

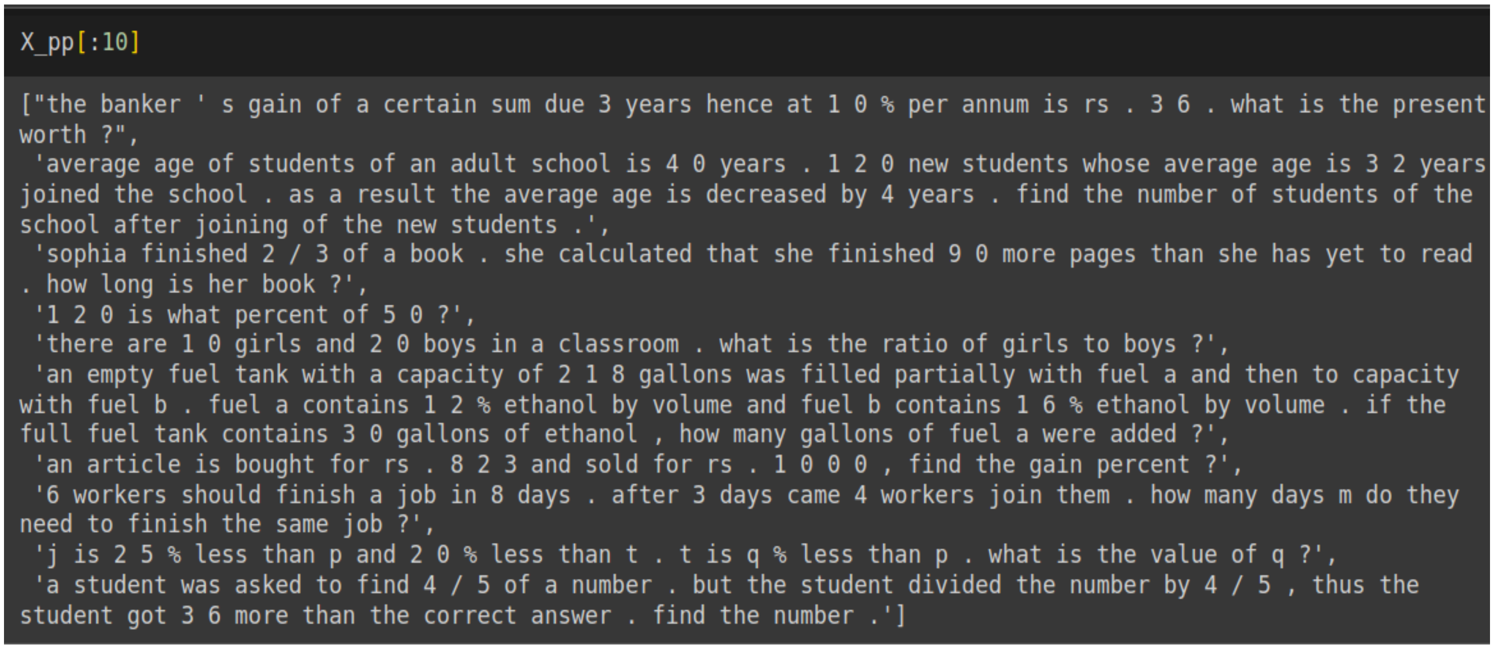
- **Padding Spaces and Removing Extra White Spaces:**

  Proper spacing is crucial for effective tokenization. Padding spaces involve ensuring consistent spacing between words, while removing extra whitespaces enhances data cleanliness. This process maintains the integrity of the text, optimizing subsequent tokenization and analysis.



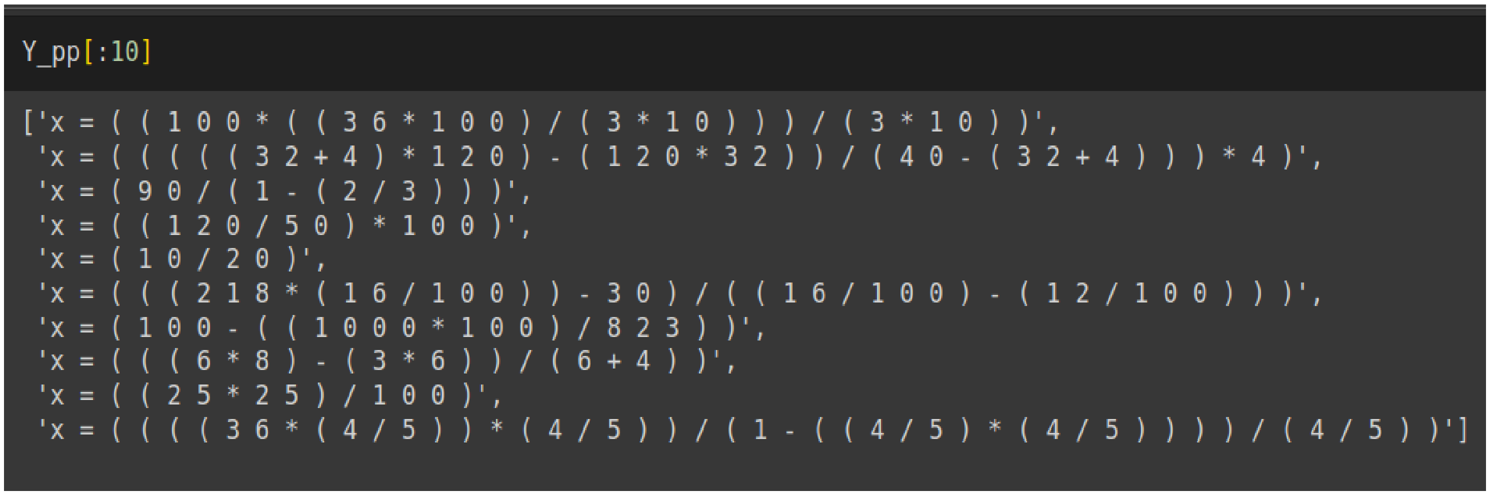
```
[12] def preprocess_X(s):
         s = s.lower().strip()
         s = re.sub(r"([?.!,’])", r" \1 ", s)
         s = re.sub(r"([0-9])", r" \1 ", s)
         s = re.sub(r'[" "]+', " ", s)
         s = s.rstrip().strip()
         return s

     def preprocess_Y(e):
         e = e.lower().strip()
         return e
```

```
[13] X_pp = list(map(preprocess_X, X))
     Y_pp = list(map(preprocess_Y, Y))
```



```
X_pp[:10]
```

```
["the banker ' s gain of a certain sum due 3 years hence at 1 0 % per annum is rs . 3 6 . what is the present
worth ?",
 'average age of students of an adult school is 4 0 years . 1 2 0 new students whose average age is 3 2 years
joined the school . as a result the average age is decreased by 4 years . find the number of students of the
school after joining of the new students .',
 'sophia finished 2 / 3 of a book . she calculated that she finished 9 0 more pages than she has yet to read
. how long is her book ?',
 '1 2 0 is what percent of 5 0 ?',
 'there are 1 0 girls and 2 0 boys in a classroom . what is the ratio of girls to boys ?',
 'an empty fuel tank with a capacity of 2 1 8 gallons was filled partially with fuel a and then to capacity
with fuel b . fuel a contains 1 2 % ethanol by volume and fuel b contains 1 6 % ethanol by volume . if the
full fuel tank contains 3 0 gallons of ethanol , how many gallons of fuel a were added ?',
 'an article is bought for rs . 8 2 3 and sold for rs . 1 0 0 0 , find the gain percent ?',
 '6 workers should finish a job in 8 days . after 3 days came 4 workers join them . how many days m do they
need to finish the same job ?',
 'j is 2 5 % less than p and 2 0 % less than t . t is q % less than p . what is the value of q ?',
 'a student was asked to find 4 / 5 of a number . but the student divided the number by 4 / 5 , thus the
student got 3 6 more than the correct answer . find the number .']
```

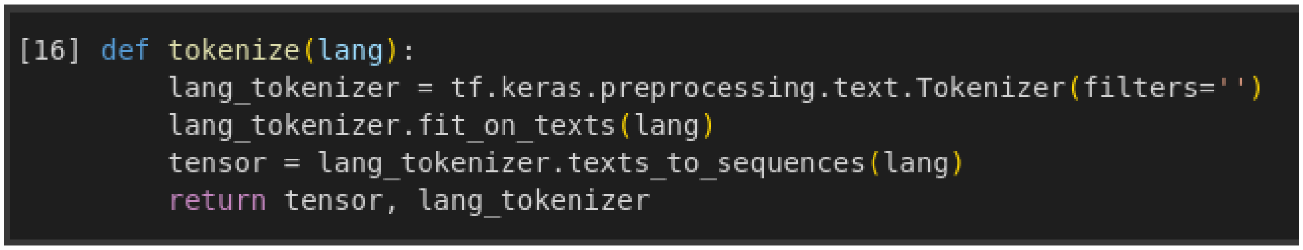


```
Y_pp[:10]
```

```
['x = ( ( 1 0 0 * ( ( 3 6 * 1 0 0 ) / ( 3 * 1 0 ) ) ) / ( 3 * 1 0 ) )',
 'x = ( ( ( ( ( 3 2 + 4 ) * 1 2 0 ) - ( 1 2 0 * 3 2 ) ) / ( 4 0 - ( 3 2 + 4 ) ) ) * 4 )',
 'x = ( 9 0 / ( 1 - ( 2 / 3 ) ) )',
 'x = ( ( 1 2 0 / 5 0 ) * 1 0 0 )',
 'x = ( 1 0 / 2 0 )',
 'x = ( ( ( 2 1 8 * ( 1 6 / 1 0 0 ) ) - 3 0 ) / ( ( 1 6 / 1 0 0 ) - ( 1 2 / 1 0 0 ) ) )',
 'x = ( 1 0 0 - ( ( 1 0 0 0 * 1 0 0 ) / 8 2 3 ) )',
 'x = ( ( ( 6 * 8 ) - ( 3 * 6 ) ) / ( 6 + 4 ) )',
 'x = ( ( 2 5 * 2 5 ) / 1 0 0 )',
 'x = ( ( ( ( 3 6 * ( 4 / 5 ) ) * ( 4 / 5 ) ) / ( 1 - ( ( 4 / 5 ) * ( 4 / 5 ) ) ) ) / ( 4 / 5 ) )']
```

- **Tokenization:**

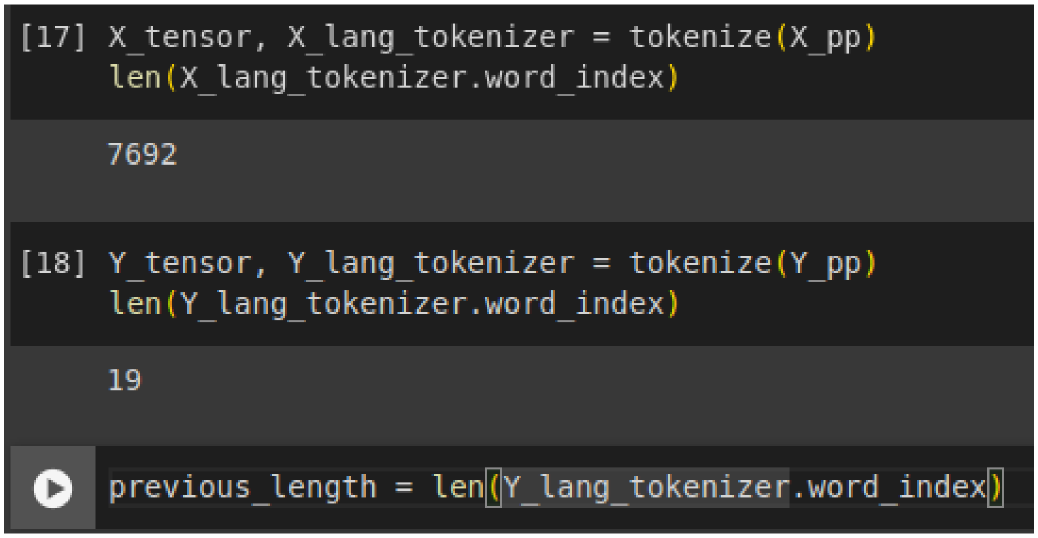
  Tokenization involves breaking down a text into smaller units, typically words or subwords. This step is crucial in NLP as it provides the model with discrete elements to analyze. Tokens serve as the basic building blocks for subsequent processing, facilitating efficient feature extraction and analysis.



```
[16] def tokenize(lang):
         lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
         lang_tokenizer.fit_on_texts(lang)
         tensor = lang_tokenizer.texts_to_sequences(lang)
         return tensor, lang_tokenizer
```

- **Converting to Tensor Sequences:**

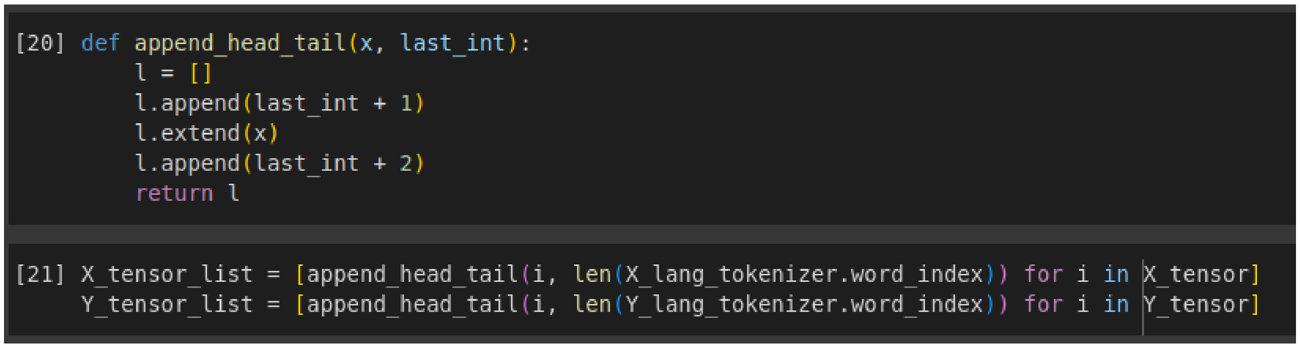
  Converting text data to tensor sequences is essential for numerical representation in NLP models. Tensors are mathematical entities that can be efficiently processed by machine learning algorithms. This conversion enables the model to work with structured numerical data, enhancing its ability to learn and generalize patterns.



```
[17] X_tensor, X_lang_tokenizer = tokenize(X_pp)
     len(X_lang_tokenizer.word_index)

     7692

[18] Y_tensor, Y_lang_tokenizer = tokenize(Y_pp)
     len(Y_lang_tokenizer.word_index)

     19

     previous_length = len(Y_lang_tokenizer.word_index)
```

- **Add integers for < start > and < end > tokens :**

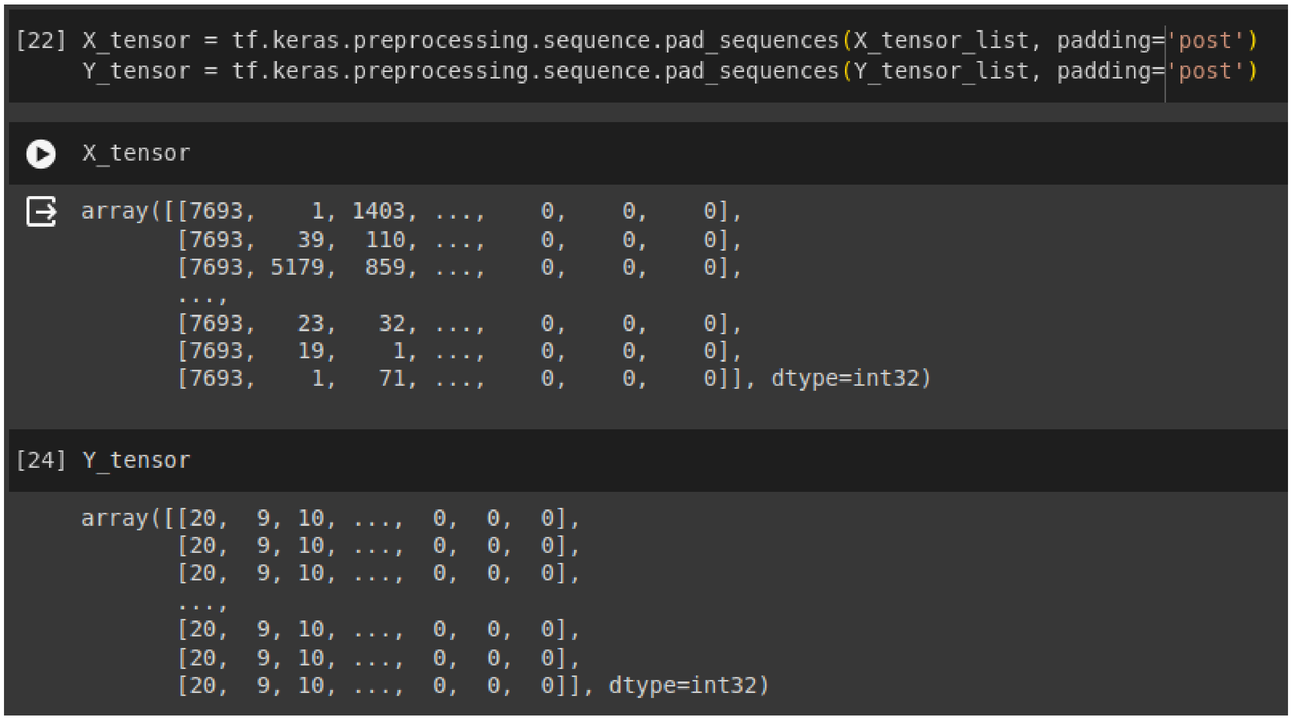
  Adding integer representations of special tokens ("<start>" and "<end>") to both input problems and target math expressions. These tokens serve as markers to indicate the beginning and end of sequences, aiding in sequence generation tasks like machine translation. This ensures proper alignment and interpretation of input-output sequences during model training and inference.



```
[20] def append_head_tail(x, last_int):
         l = []
         l.append(last_int + 1)
         l.extend(x)
         l.append(last_int + 2)
         return l

[21] X_tensor_list = [append_head_tail(i, len(X_lang_tokenizer.word_index)) for i in X_tensor]
     Y_tensor_list = [append_head_tail(i, len(Y_lang_tokenizer.word_index)) for i in Y_tensor]
```

- **Zero Padding for Equalizing Sequence Lengths:**

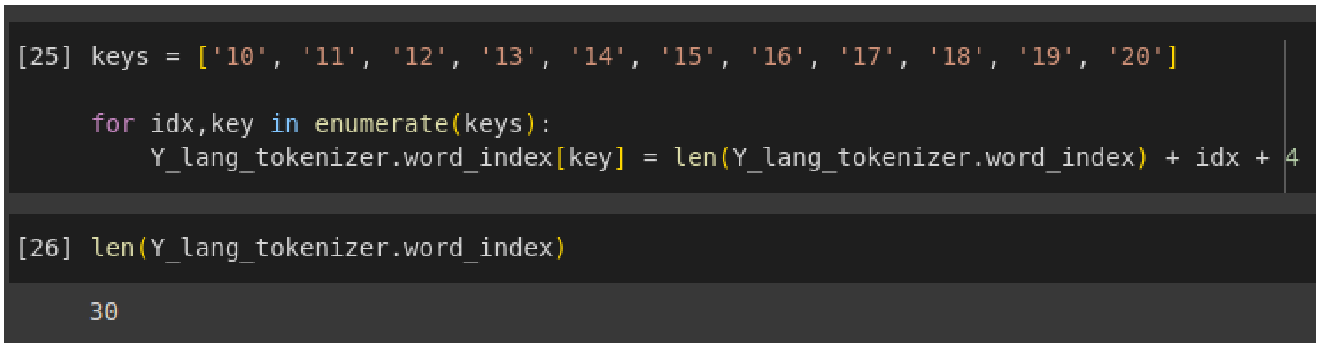
  Pads the sequences with zeros to ensure uniform length, which is crucial for processing batches efficiently during training. It helps maintain consistent input dimensions across data samples.



```
[22] X_tensor = tf.keras.preprocessing.sequence.pad_sequences(X_tensor_list, padding='post')
     Y_tensor = tf.keras.preprocessing.sequence.pad_sequences(Y_tensor_list, padding='post')

     X_tensor

     array([[7693,    1, 1403, ...,    0,    0,    0],
            [7693,   39,  110, ...,    0,    0,    0],
            [7693, 5179,  859, ...,    0,    0,    0],
            ...,
            [7693,   23,   32, ...,    0,    0,    0],
            [7693,   19,    1, ...,    0,    0,    0],
            [7693,    1,   71, ...,    0,    0,    0]], dtype=int32)

[24] Y_tensor

     array([[20,  9, 10, ...,  0,  0,  0],
            [20,  9, 10, ...,  0,  0,  0],
            [20,  9, 10, ...,  0,  0,  0],
            ...,
            [20,  9, 10, ...,  0,  0,  0],
            [20,  9, 10, ...,  0,  0,  0],
            [20,  9, 10, ...,  0,  0,  0]], dtype=int32)
```

- **Increasing the vocabulary size of the target :**

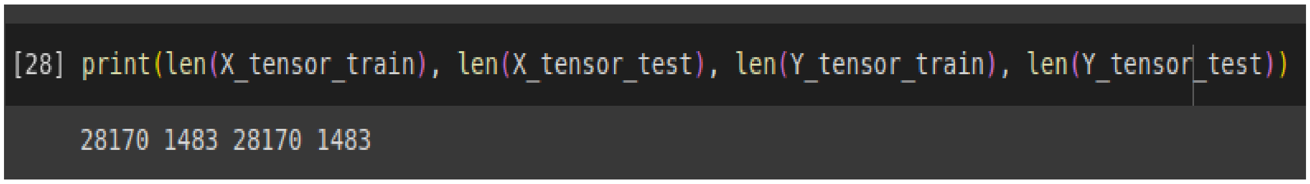
  By including some fodder words which won't be used. This is done to avoid problems later which manifest due to short vocabulary size.



```
[25] keys = ['10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']

     for idx,key in enumerate(keys):
         Y_lang_tokenizer.word_index[key] = len(Y_lang_tokenizer.word_index) + idx + 4

[26] len(Y_lang_tokenizer.word_index)

     30
```

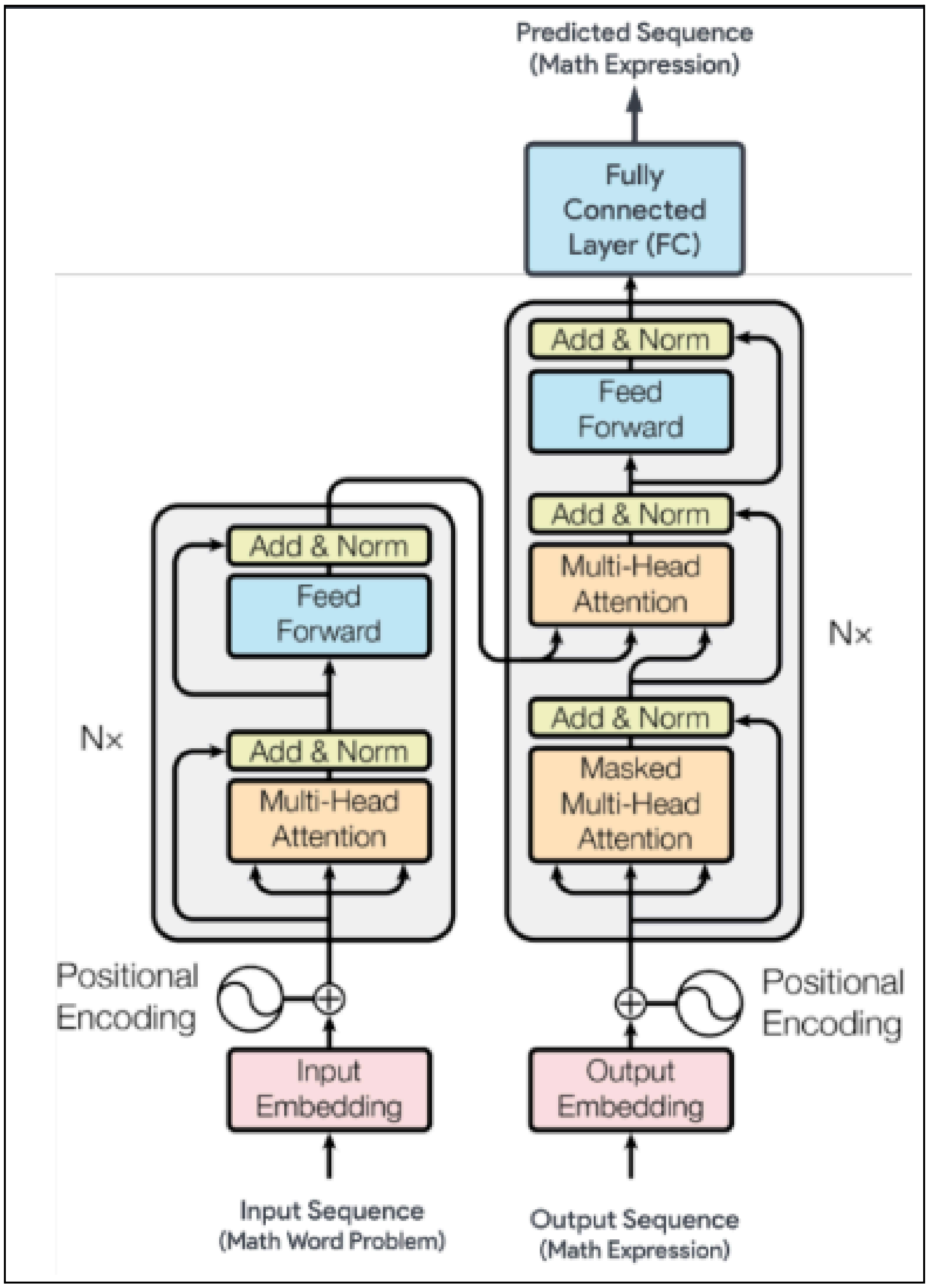
- **Train-Test Split → (95 : 5) :**

X_tensor_train, X_tensor_test, Y_tensor_train, Y_tensor_test = train_test_split(X_tensor, Y_tensor, test_size=0.05, random_state=42)



```
[28] print(len(X_tensor_train), len(X_tensor_test), len(Y_tensor_train), len(Y_tensor_test))

     28170 1483 28170 1483
```

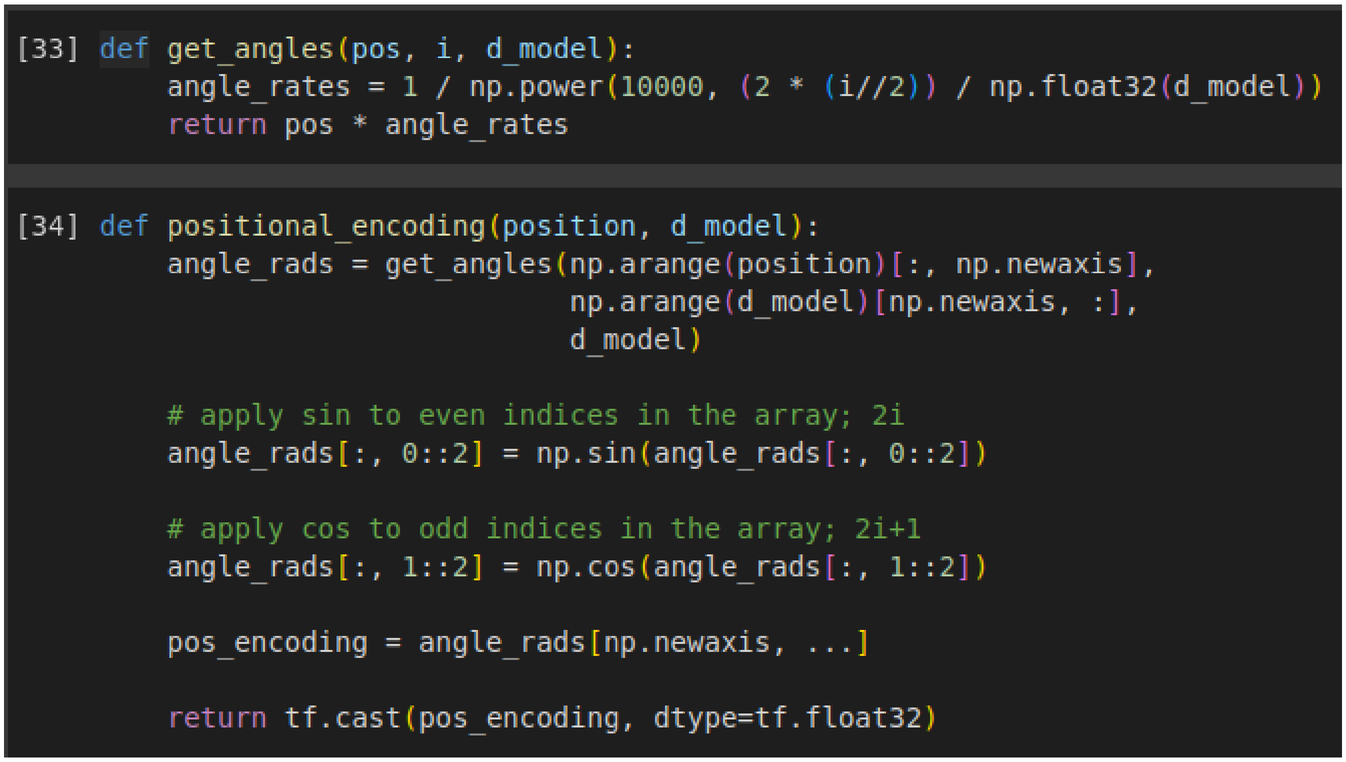
## The Transformer Model behind MathIQ



## Positional Encoding

Since transformers have no recurrence or convolution, we use positional encoding to let the model have an idea about the relative positions of the tokens in a sequence.

$$\mathrm{PE}(pos, 2i) = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$\mathrm{PE}(pos, 2i+1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

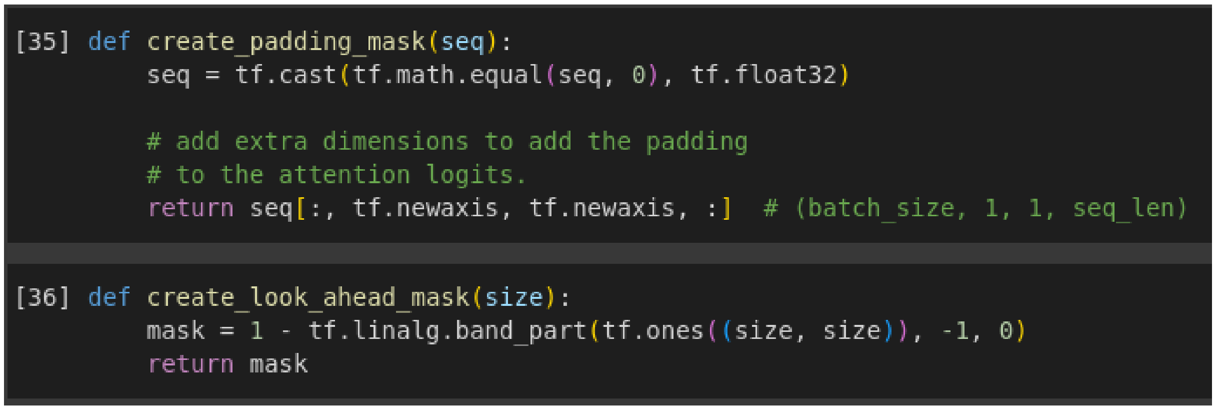


```
[33] def get_angles(pos, i, d_model):
         angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
         return pos * angle_rates
```

```
[34] def positional_encoding(position, d_model):
         angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                                 np.arange(d_model)[np.newaxis, :],
                                 d_model)

         # apply sin to even indices in the array; 2i
         angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

         # apply cos to odd indices in the array; 2i+1
         angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

         pos_encoding = angle_rads[np.newaxis, ...]

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

## Masking

We mask all the padding elements so that they are not considered as input to the model. The position of the pad tokens are the positions at which the mask shows 1 and at the other locations it shows 0. The subsequent tokens in a sequence are masked using the look-ahead mask and this mask indicates the entries that should be avoided.



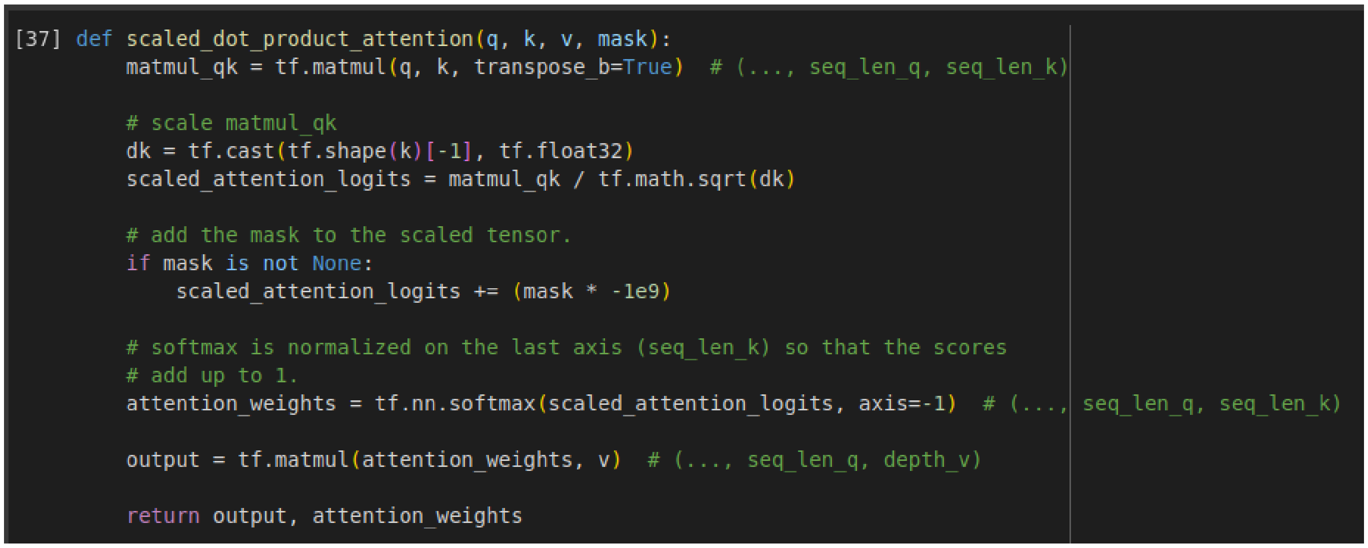
```
[35] def create_padding_mask(seq):
         seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

         # add extra dimensions to add the padding
         # to the attention logits.
         return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)
```

```
[36] def create_look_ahead_mask(size):
         mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
         return mask
```

## Scaled Dot-product Attention

A “soft” dictionary lookup which returns a weighted sum of the values in the corpus. This weight represents the usefulness of a particular token in embedding the query token.

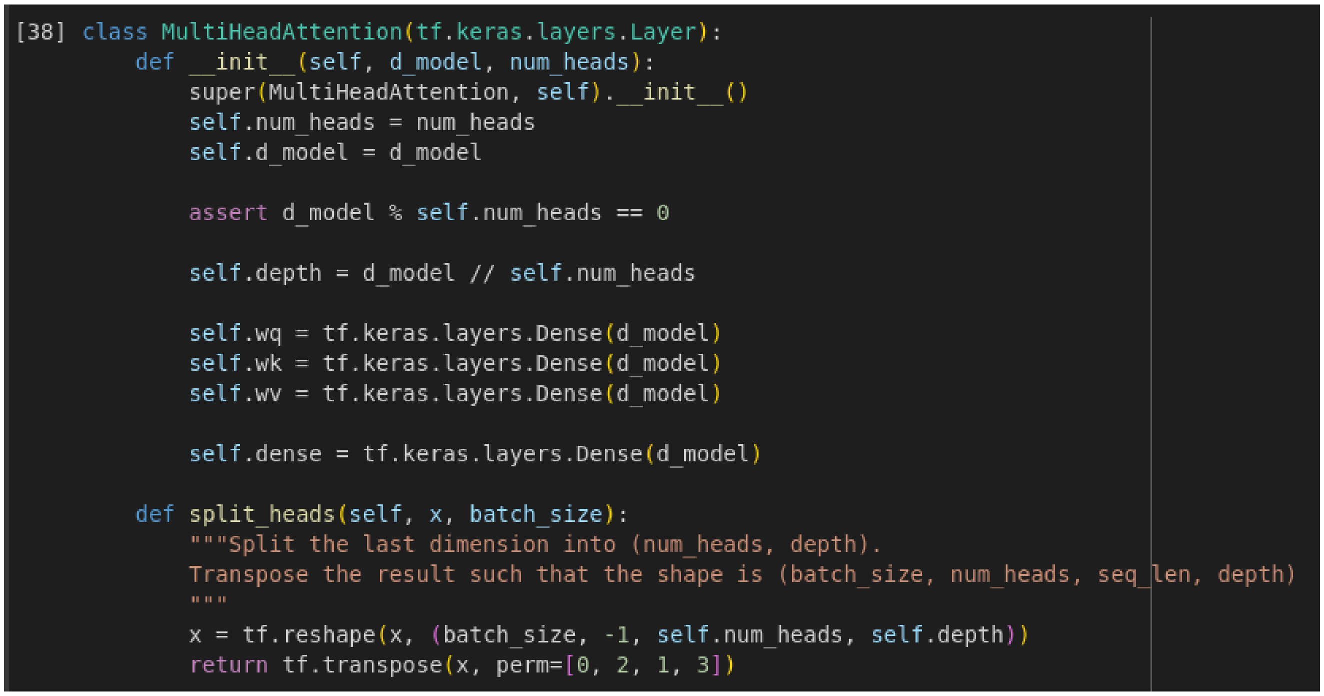
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



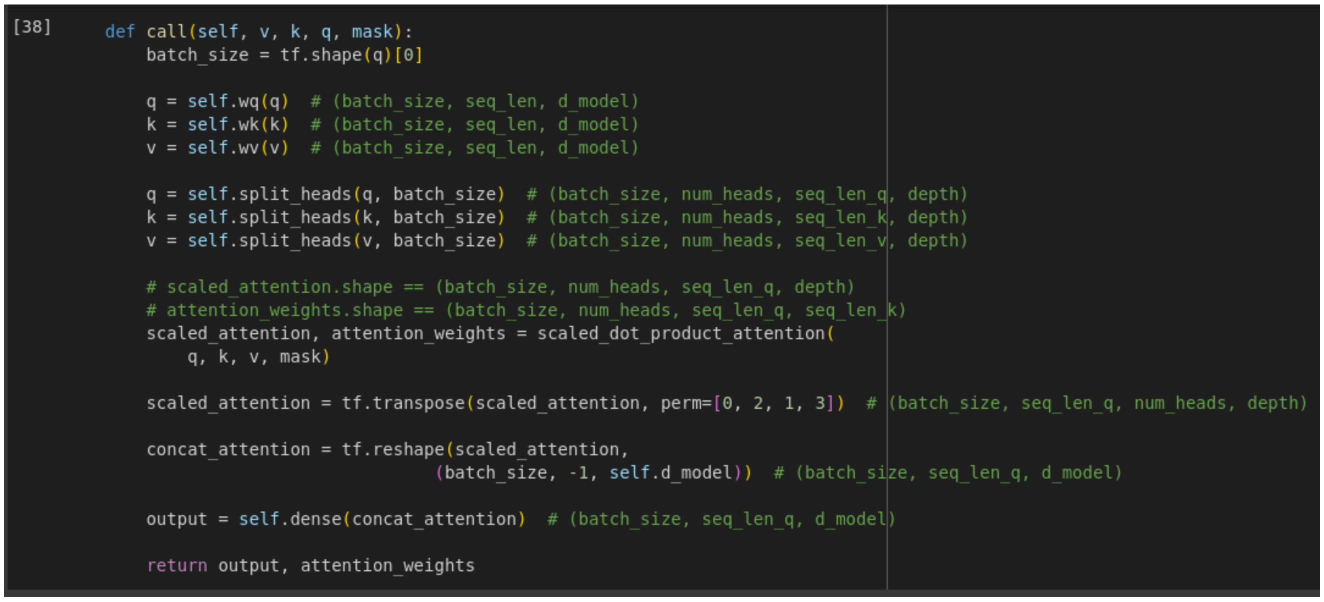
```
[37] def scaled_dot_product_attention(q, k, v, mask):
         matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

         # 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

It is a combination of 'h' self-attention heads, where each head is sandwiched between two linear layers.



```
[38] 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])
```



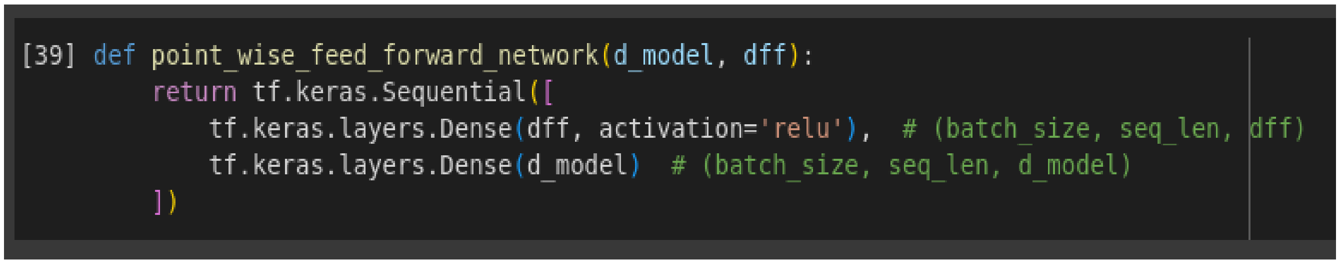
```
[38]     def call(self, v, k, q, mask):
             batch_size = tf.shape(q)[0]

             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
```

## Embedding

- In the Transformer model architecture, embedding layers are employed to convert input tokens into dense vectors.
- Both the encoder and decoder layers utilize embedding layers (‘tf.keras.layers.Embedding’) to convert input tokens (words) into continuous vector representations.
- These embeddings capture semantic information about the tokens, allowing the model to understand the meaning of words in the context of the task.
- **Point-wise Feed Forward Neural Network :**

  Every encoder and decoder layer contains a fully connected feed-forward network. It is composed of two fully-connected layers with a ReLU activation function in between them.

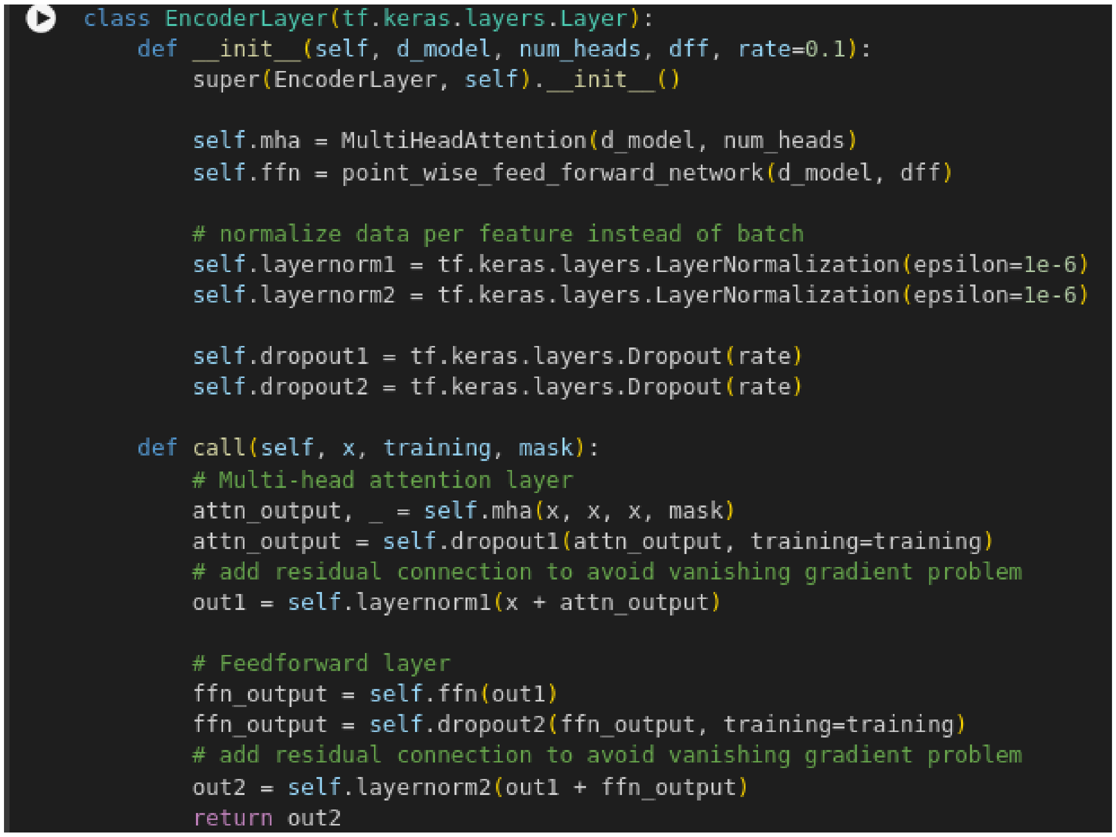
$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



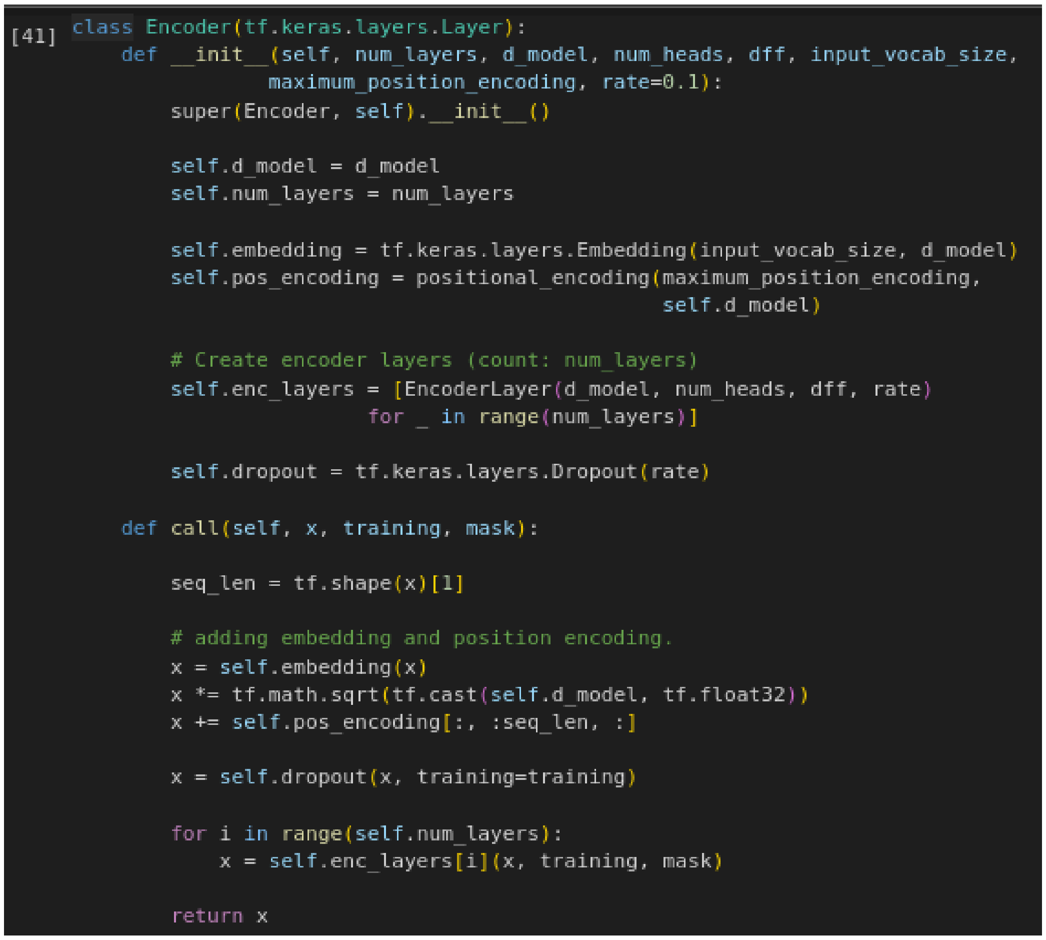
```
[39] def point_wise_feed_forward_network(d_model, dff):
         return tf.keras.Sequential([
             tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
             tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
         ])
```

- **Encoder**

  The encoder consists of N layers and each layer has 2 sub-layers. The first sublayer is a multi-head self attention mechanism layer and the second sub-layer is a position-wise fully connected feed-forward neural network. Each of these sub-layers is preceded by a skip connection or residual connection and succeeded by a layer normalization block. They result in outputs of dimension d-model to facilitate the skip connections. The residual connections are important as they assist in overcoming the vanishing gradient problem in deep network architectures.



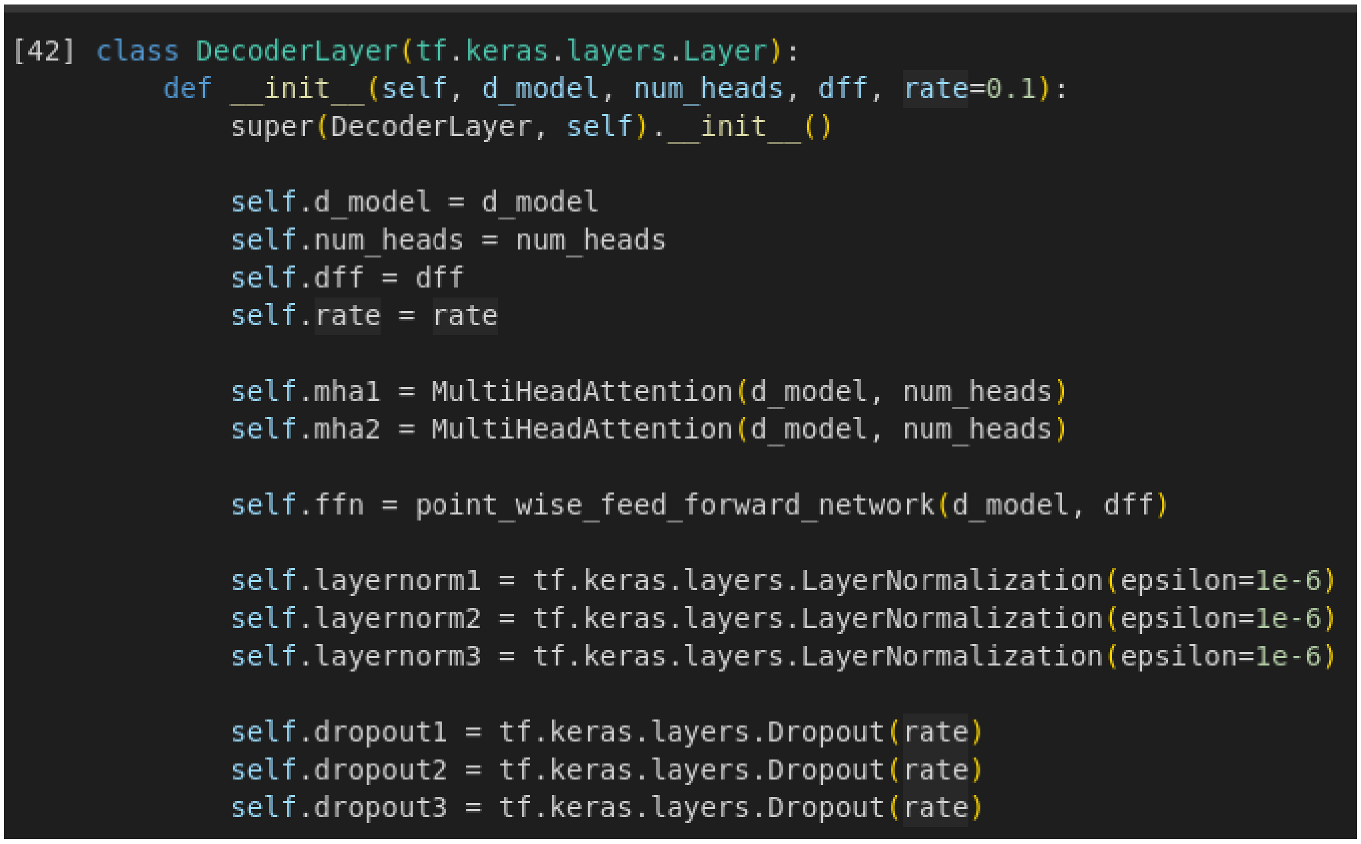
```python
class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(EncoderLayer, self).__init__()

        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = point_wise_feed_forward_network(d_model, dff)

        # normalize data per feature instead of batch
        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):
        # Multi-head attention layer
        attn_output, _ = self.mha(x, x, x, mask)
        attn_output = self.dropout1(attn_output, training=training)
        # add residual connection to avoid vanishing gradient problem
        out1 = self.layernorm1(x + attn_output)

        # Feedforward layer
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        # add residual connection to avoid vanishing gradient problem
        out2 = self.layernorm2(out1 + ffn_output)
        return out2
```



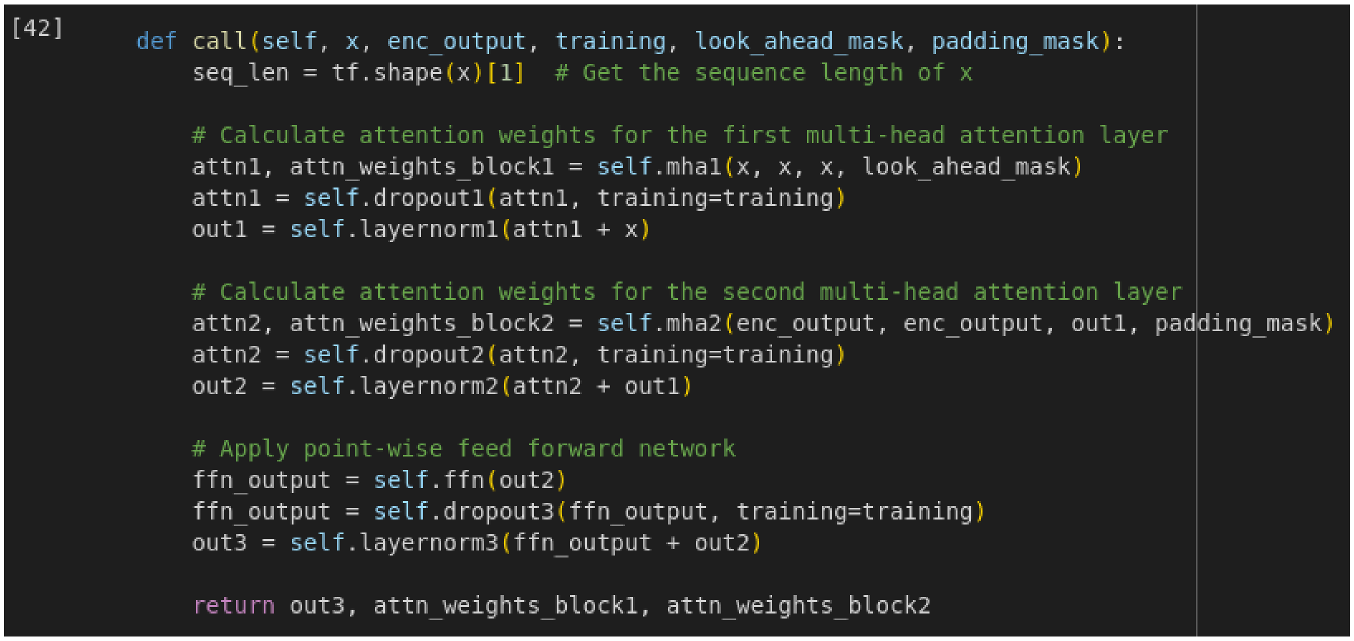
```python
[41] class Encoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
                 maximum_position_encoding, rate=0.1):
        super(Encoder, self).__init__()

        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)

        # Create encoder layers (count: num_layers)
        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)
        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**

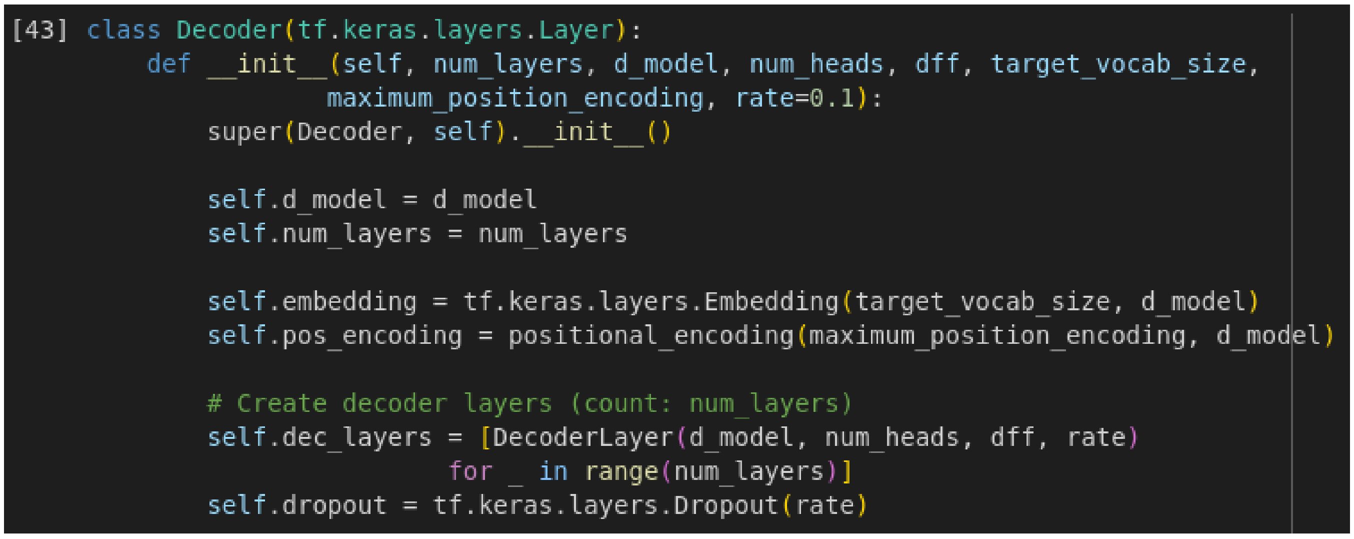
  The decoder consists of N layers and each layer has 3 sub-layers. Two of the sub-layers are the same as those of the encoder. The third sub-layer utilizes the outputs of the encoder stack and performs multi-head attention over them. Decoder feeds on two types of inputs which are, the outputs of the encoder and positionally encoded target output embeddings. And just like the encoder, decoder also has subsequent layer normalization blocks and incorporation of skip connections after and before each sub-layer respectively. The principal task of the decoder is to predict a token at position n by looking at all the preceding n-1 tokens using the look-ahead mask. The predicted sequence is then passed through a fully-connected neural network layer to generate the final math expression.



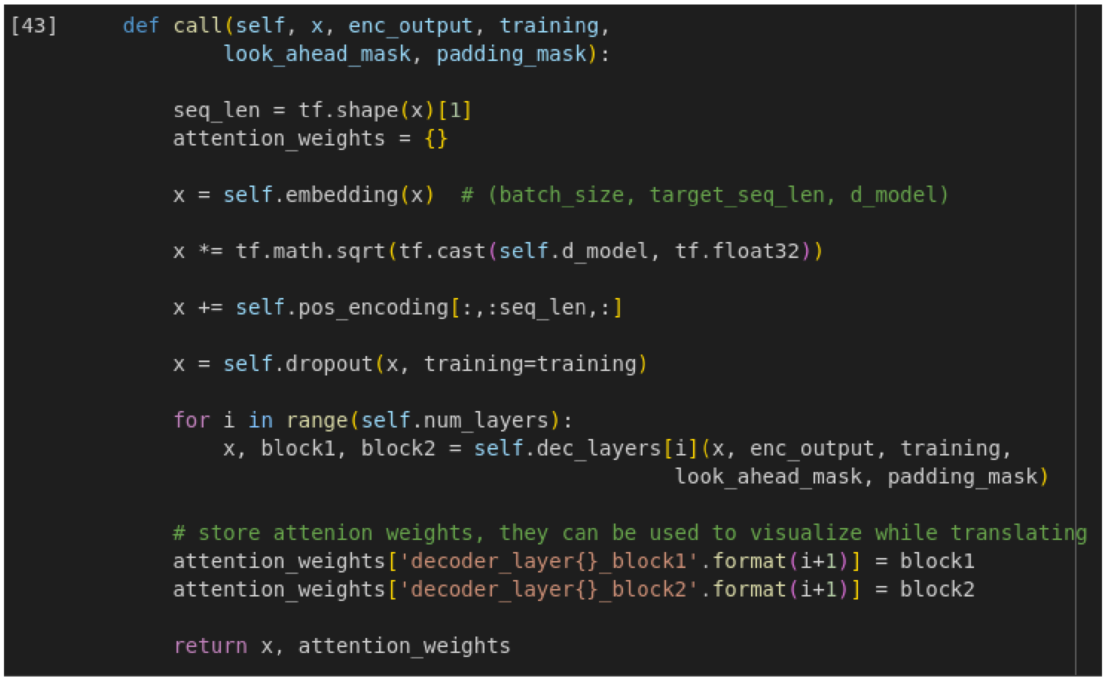
```
[42] class DecoderLayer(tf.keras.layers.Layer):
         def __init__(self, d_model, num_heads, dff, rate=0.1):
             super(DecoderLayer, self).__init__()

             self.d_model = d_model
             self.num_heads = num_heads
             self.dff = dff
             self.rate = rate

             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)
```



```
[42]    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
            seq_len = tf.shape(x)[1]  # Get the sequence length of x

            # Calculate attention weights for the first multi-head attention layer
            attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)
            attn1 = self.dropout1(attn1, training=training)
            out1 = self.layernorm1(attn1 + x)

            # Calculate attention weights for the second multi-head attention layer
            attn2, attn_weights_block2 = self.mha2(enc_output, enc_output, out1, padding_mask)
            attn2 = self.dropout2(attn2, training=training)
            out2 = self.layernorm2(attn2 + out1)

            # Apply point-wise feed forward network
            ffn_output = self.ffn(out2)
            ffn_output = self.dropout3(ffn_output, training=training)
            out3 = self.layernorm3(ffn_output + out2)

            return out3, attn_weights_block1, attn_weights_block2
```

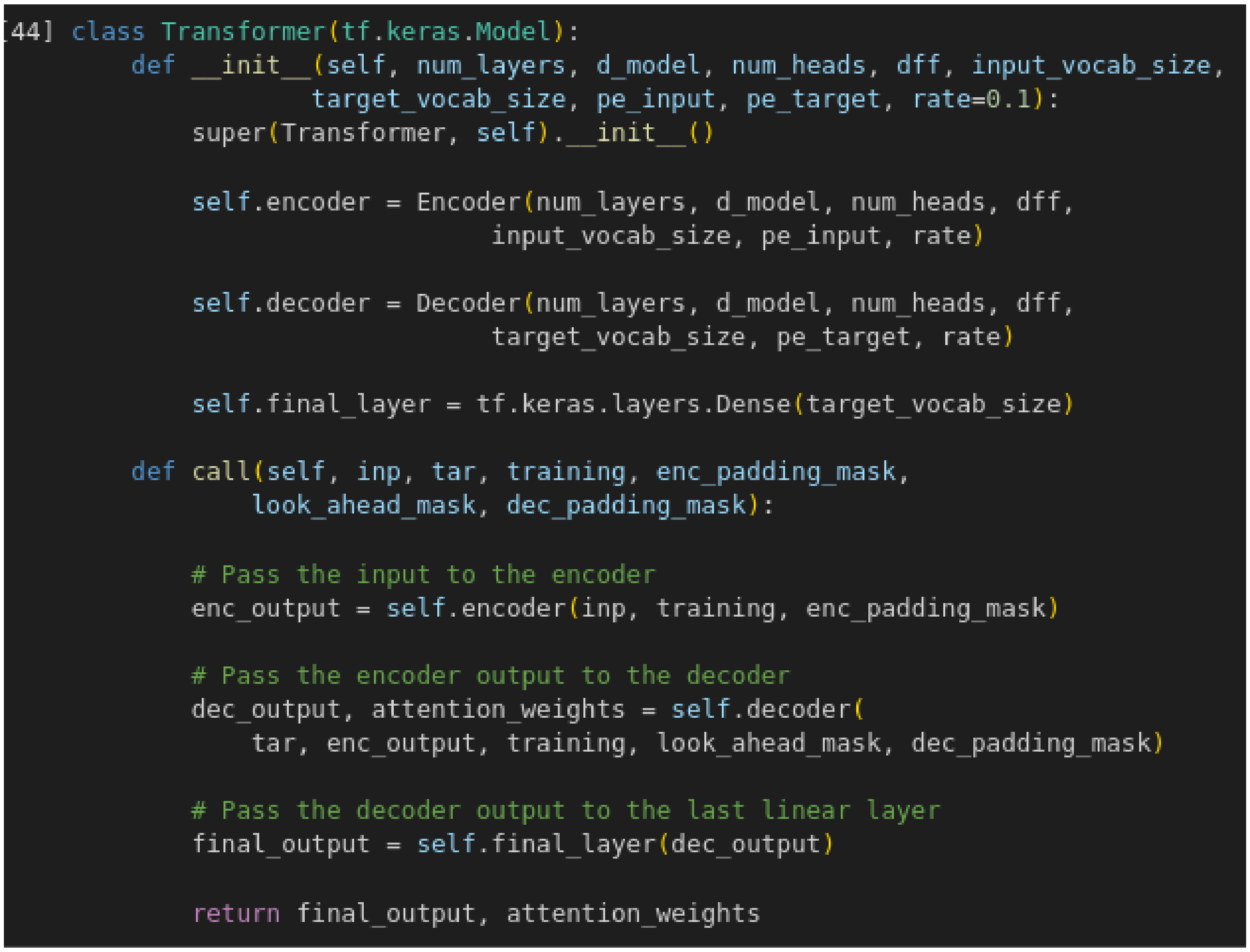


```
[43] class Decoder(tf.keras.layers.Layer):
         def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
                      maximum_position_encoding, rate=0.1):
             super(Decoder, self).__init__()

             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)

             # Create decoder layers (count: num_layers)
             self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                                for _ in range(num_layers)]
             self.dropout = tf.keras.layers.Dropout(rate)
```



```
[43]    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)

            # store attenion weights, they can be used to visualize while translating
            attention_weights['decoder_layer{}_block1'.format(i+1)] = block1
            attention_weights['decoder_layer{}_block2'.format(i+1)] = block2

            return x, attention_weights
```

- **Transformer**



```
[44] 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.encoder = 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):

             # Pass the input to the encoder
             enc_output = self.encoder(inp, training, enc_padding_mask)

             # Pass the encoder output to the decoder
             dec_output, attention_weights = self.decoder(
                 tar, enc_output, training, look_ahead_mask, dec_padding_mask)

             # Pass the decoder output to the last linear layer
             final_output = self.final_layer(dec_output)

             return final_output, attention_weights
```

## Resources

### Tutorials:

TensorFlow tutorial on Transformers

Transfer learning and Transformer models (ML Tech Talks)

### Inspirations:

We were inspired by similar research works and projects like:

Attention Is All You Need

Graph2Tree

MWP-BERT

Deep Learning-based MWP solving

## Milestones To Achieve In Future Mandates

**Mandate 4:** Implement Word Sense Disambiguation (WSD) techniques to accurately understand the meanings of ambiguous words in specific mathematical contexts. Integrate Named Entity Recognition (NER) to identify and categorize entities like mathematical terms, symbols, and units in given math problems. Apply Topic Modeling to capture the underlying mathematical terms in user queries. Employ fine-tuning methods and design a user-friendly interface to optimize the performance of MathIQ.