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

Introduction

Dataset

Data Preprocessing

Hyperparameters

The Transformer Model behind MathIQ

Positional Encoding

Masking

Scaled Dot-product Attention

Multi-Head Attention

Point-wise Feed Forward Neural Network

Encoder

Decoder

Encoder Layer and Decoder Layer

Transformer

Optimizer

Performance Metric

Screenshot of Implementation

Drive Link

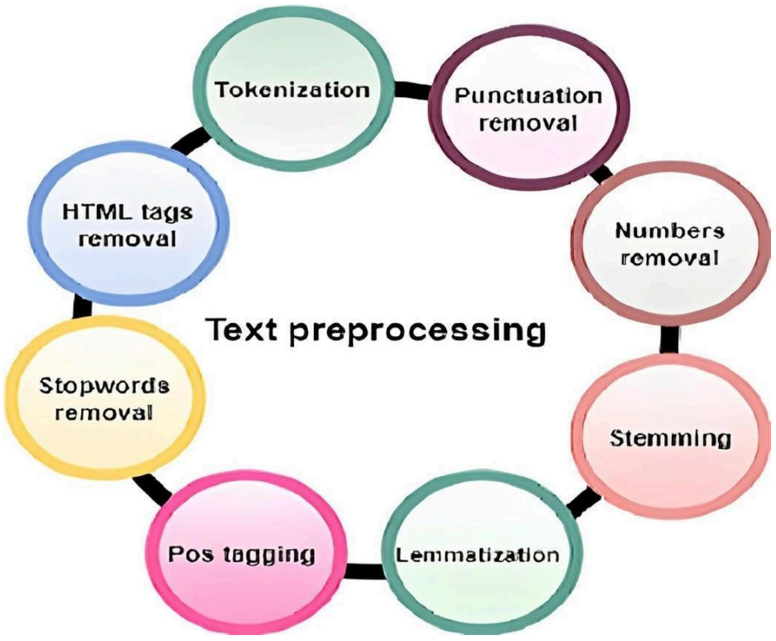
Problems Faced

Resources

Introduction

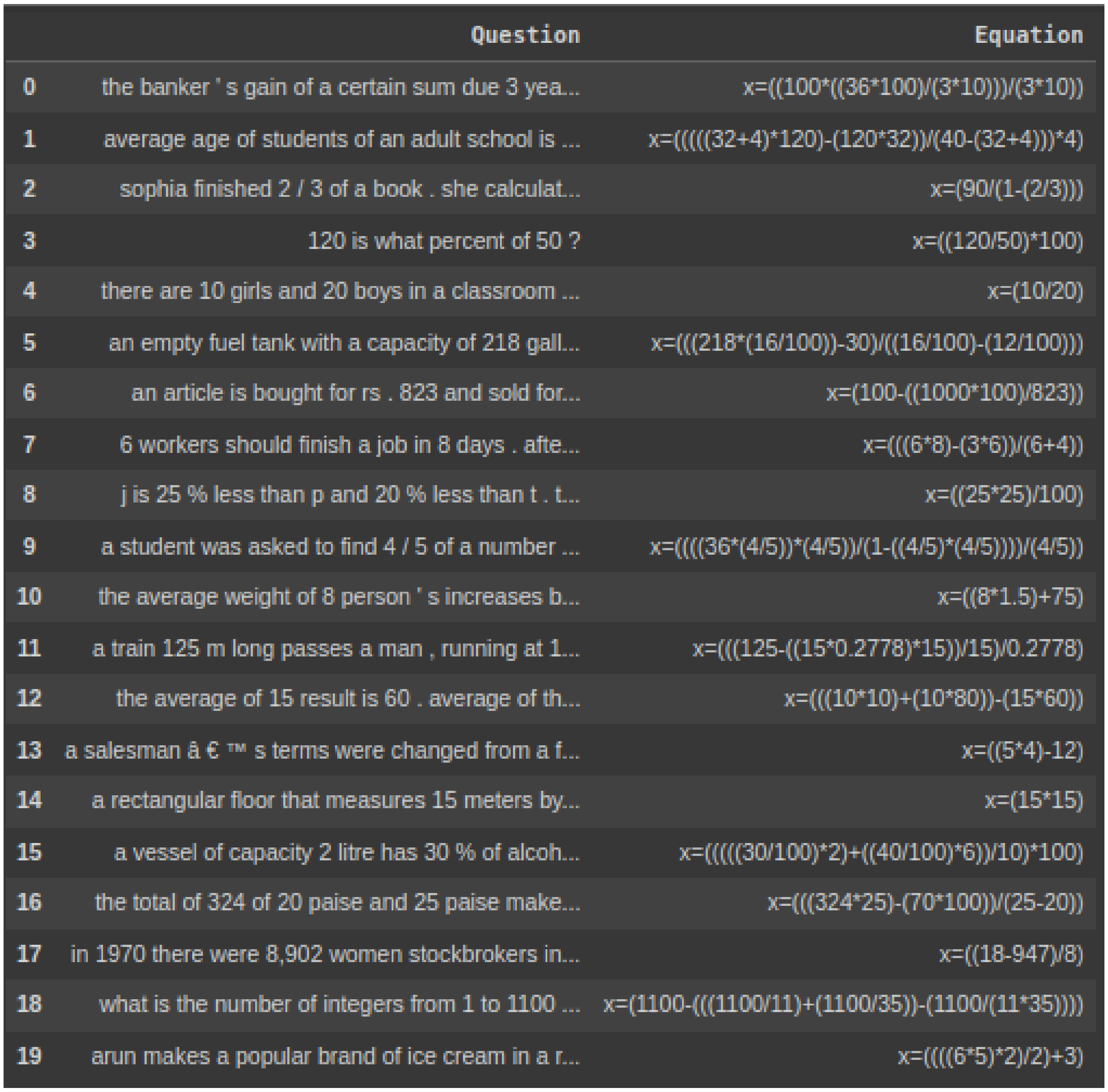
This project tackles the challenge of converting math word problems into their corresponding equations. We leverage the power of deep learning, specifically a Transformer model, to bridge the gap between natural and mathematical language. However, before diving into equation generation, we delve into the world of conceptual modeling fundamentals. This involves techniques like word sense disambiguation and Named Entity Recognition to ensure the model understands the true meaning of words and identifies key elements within the problem.

Spectral models for latent semantics, such as LSA and word embeddings, play a crucial role in capturing the underlying relationships between words and concepts. Additionally, topic modeling helps uncover hidden thematic structures within the data, further enriching the model's understanding. Finally, the Transformer itself incorporates Masked Language Models (MLM), enabling it to predict the next element in the sequence, ultimately building the equation step-by-step. Through this comprehensive approach, we aim to empower users with a powerful tool to translate the complexities of text-based math problems into clear and concise equations.



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)

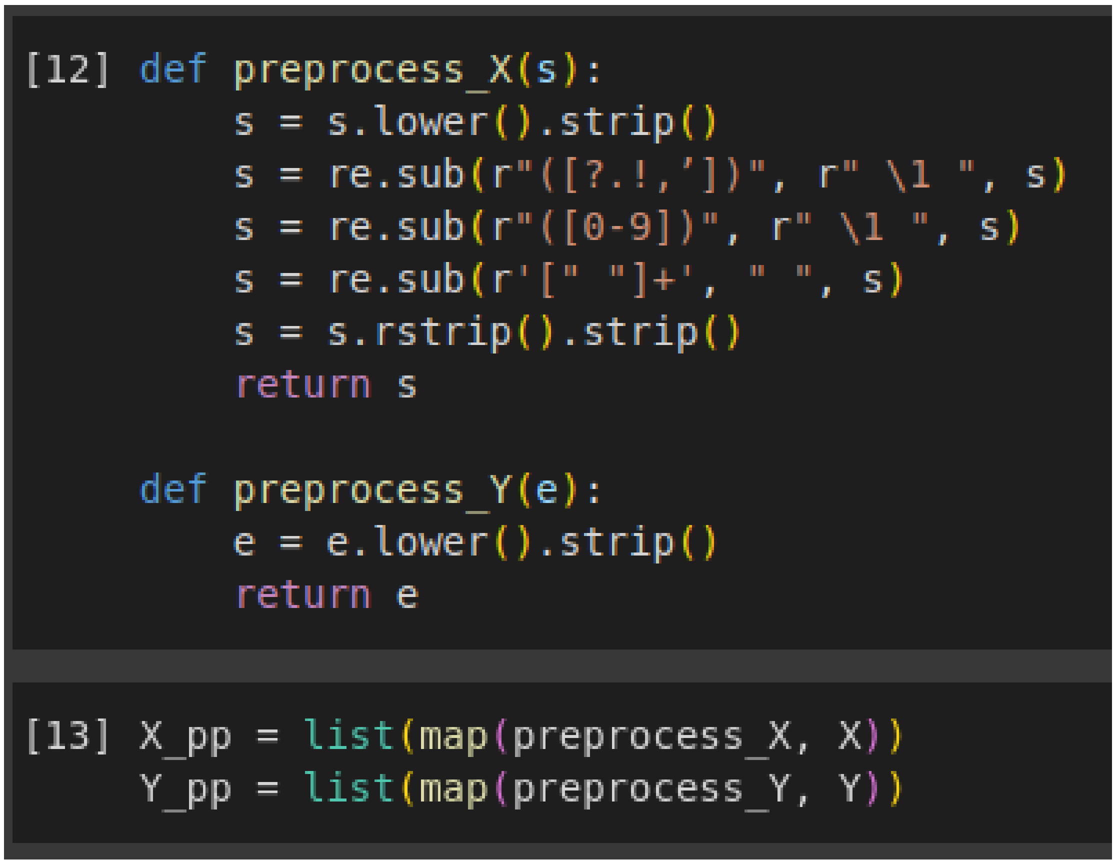
Data Preprocessing

- **Lowercase Alphabets:**

  To ensure uniformity by converting all characters to lowercase. This prevents inconsistencies in word treatment based on case, improving model generalization.

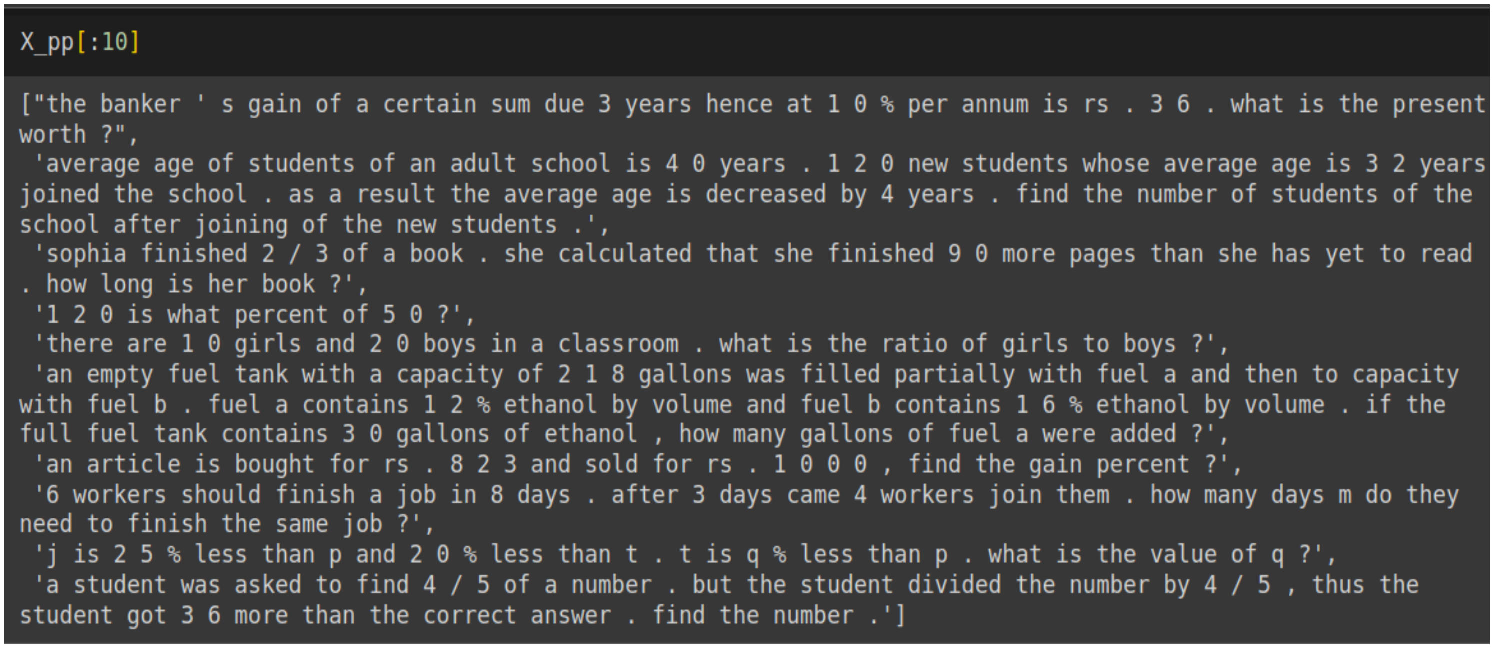
- **Padding Spaces and Removing Extra White Spaces:**

  This ensures consistent spacing between words for effective tokenization, and enhances data cleanliness by removing unnecessary whitespaces.



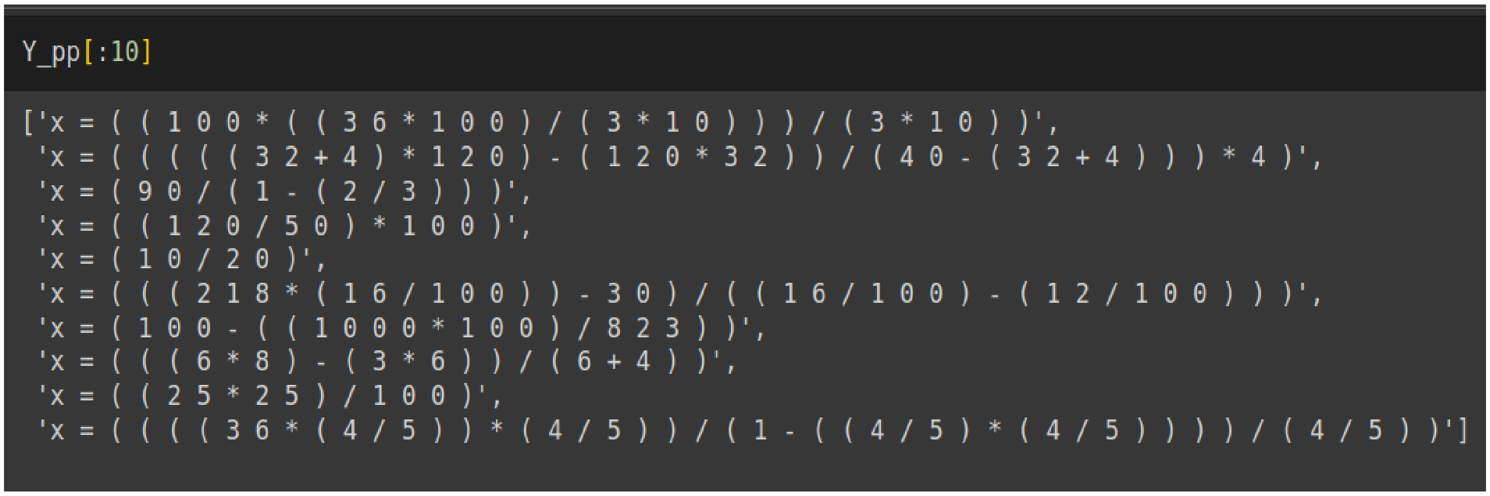
```
[12] def preprocess_X(s):
         s = s.lower().strip()
         s = re.sub(r"([?.!,’])", r" \1 ", s)
         s = re.sub(r"([0-9])", r" \1 ", s)
         s = re.sub(r'[" "]+', " ", s)
         s = s.rstrip().strip()
         return s

     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 .']
```

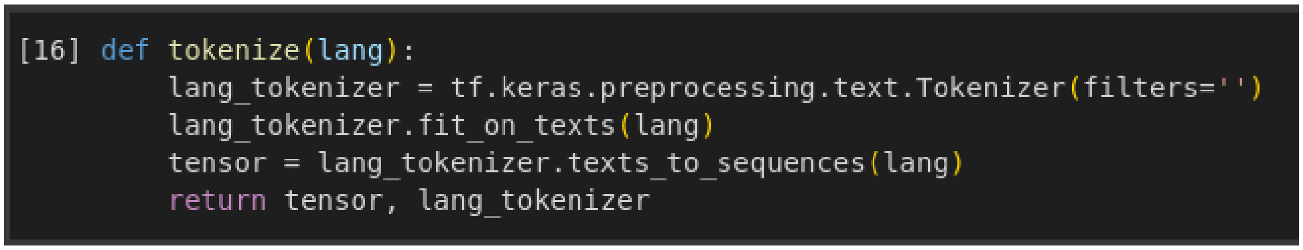


```
Y_pp[:10]
```

```
['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:**

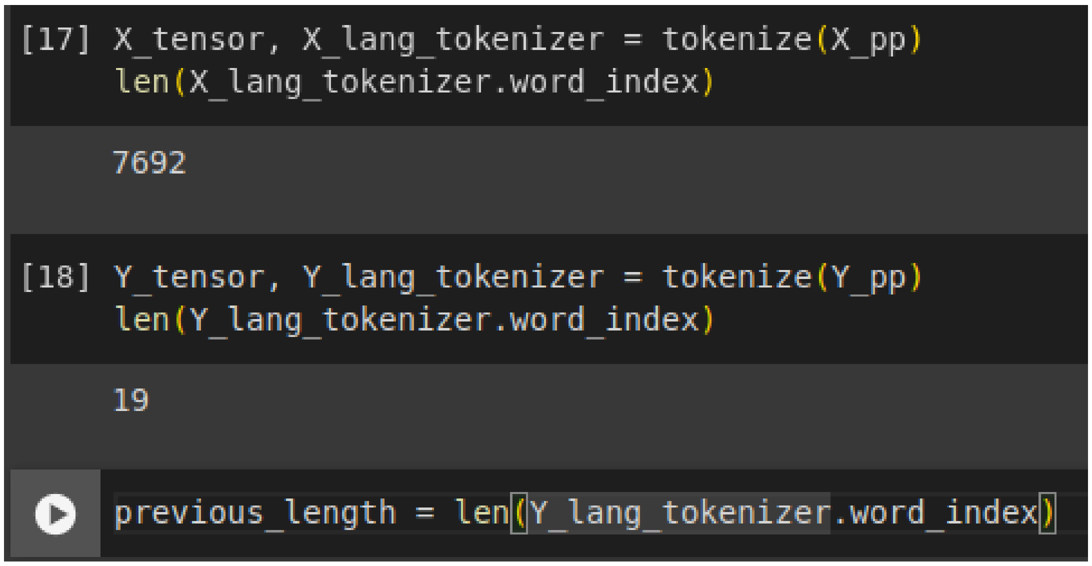
  It breaks down text into smaller units (words or subwords) for analysis. It also provides discrete elements for subsequent processing, facilitating feature extraction.



```
[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:**

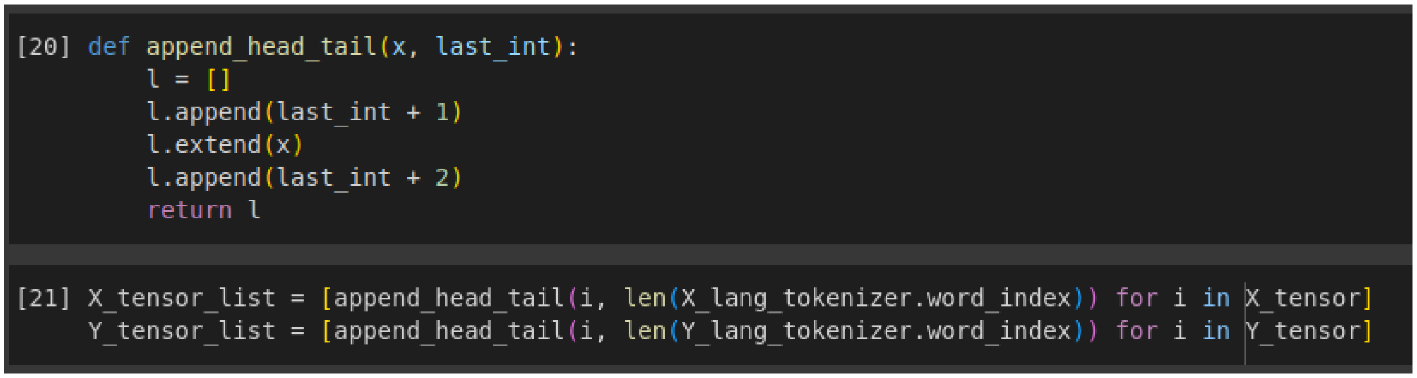
  This is essential for numerical representation in NLP models. It enables efficient processing by machine learning algorithms.



```
[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 :**

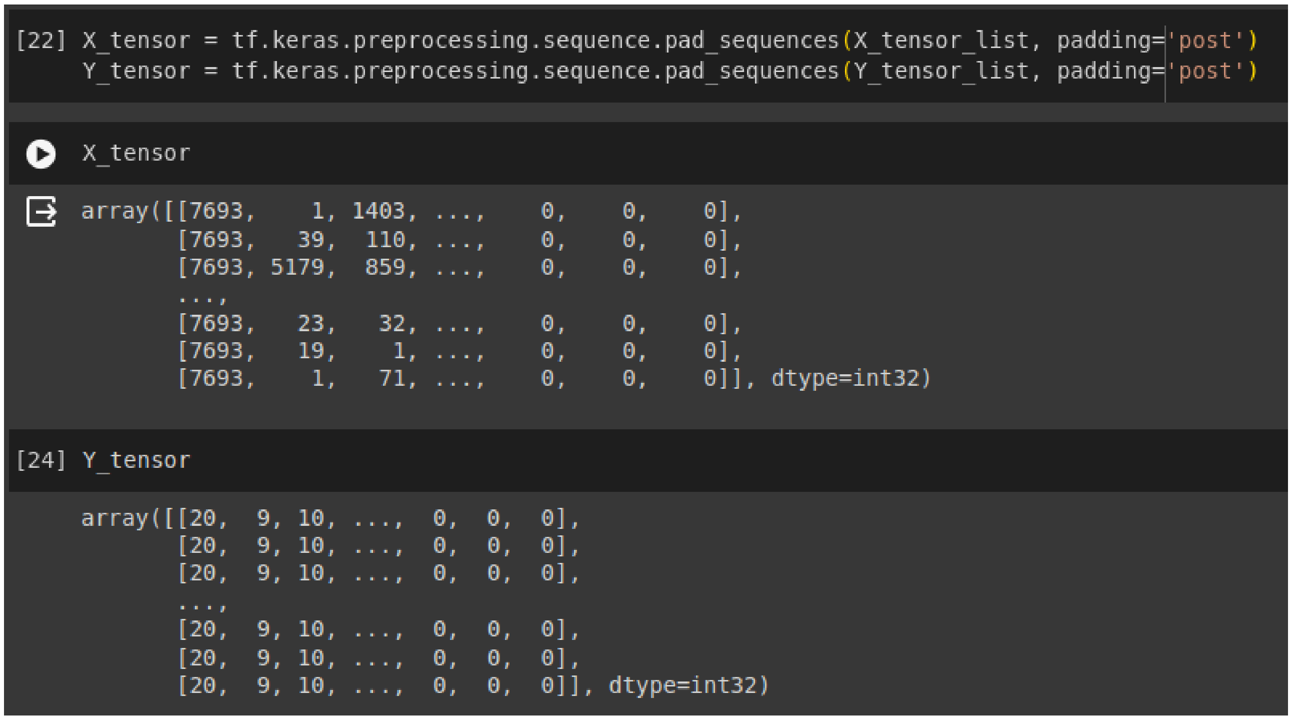
  Markers to indicate the beginning and end of sequences, aiding in sequence generation tasks. This ensures proper alignment and interpretation of input-output sequences during model training and inference.



```
[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:**

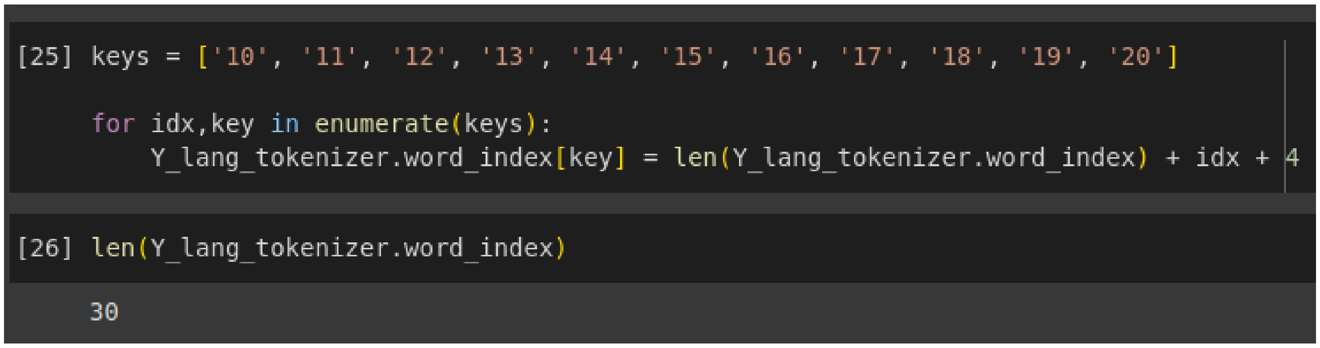
  Addresses challenges posed by sequences of varying lengths in model training, and ensures consistent input dimensions for neural networks by adding.



```
[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 :**

  This will include additional words to avoid problems due to a short vocabulary size.

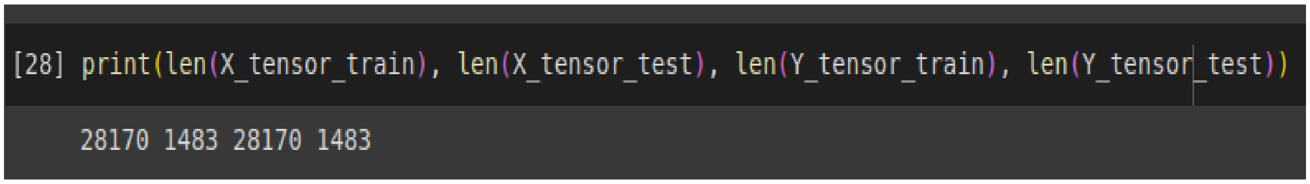


```
[25] keys = ['10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20']

     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) :**

  Splits the dataset into training and testing sets with a ratio of 95:5, respectively, for model evaluation.

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
```

Hyperparameters

• Number of Encoder-Decoder layers, N=4

• Embedding dimension of input/output, *dmodel*=128

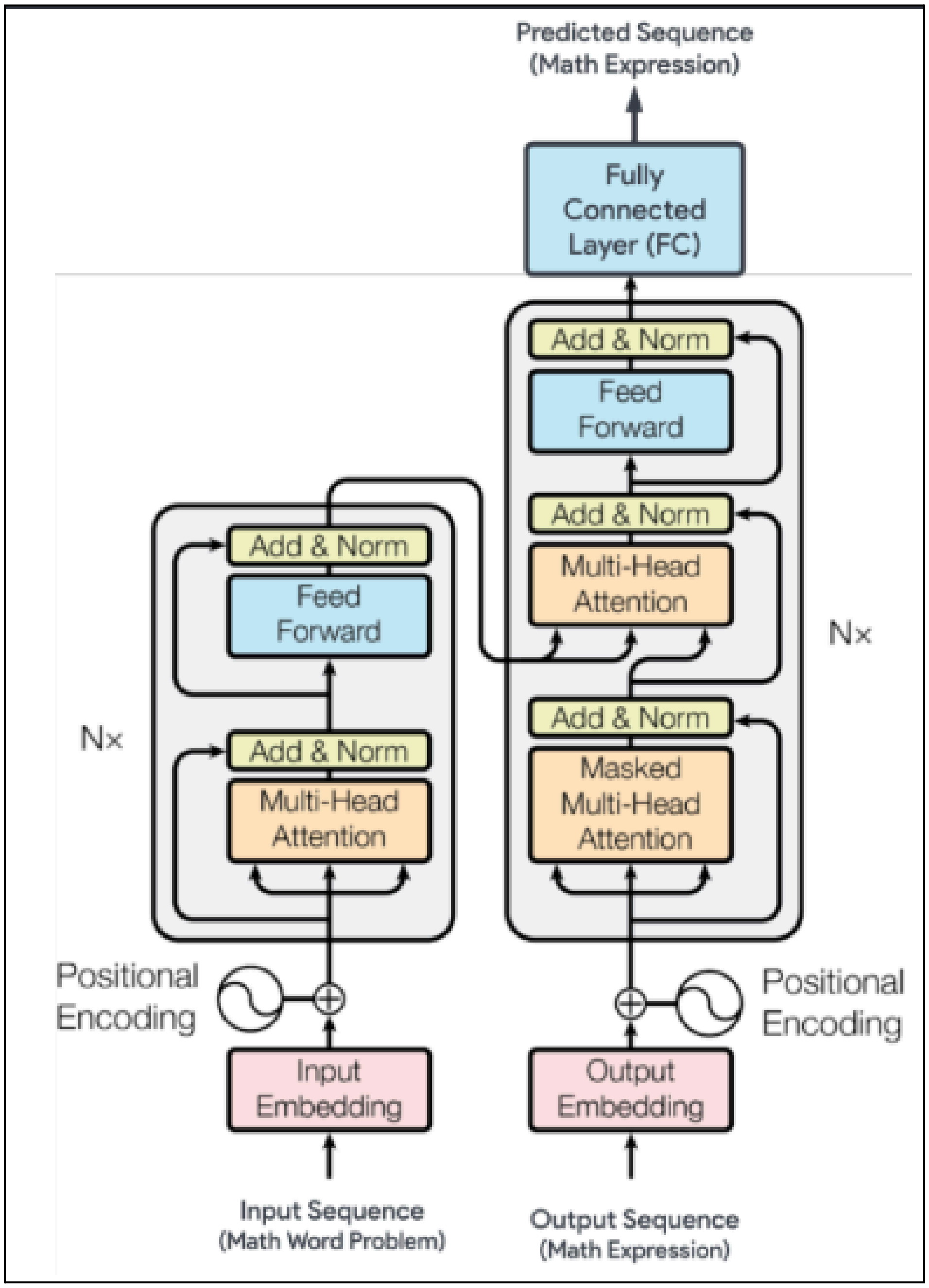
• Number of Self-Attention heads, h=8

• Dropout Rate, *Pdrop*=0.1

• Size of each mini-batch = 64

• Epochs = 17

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.

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

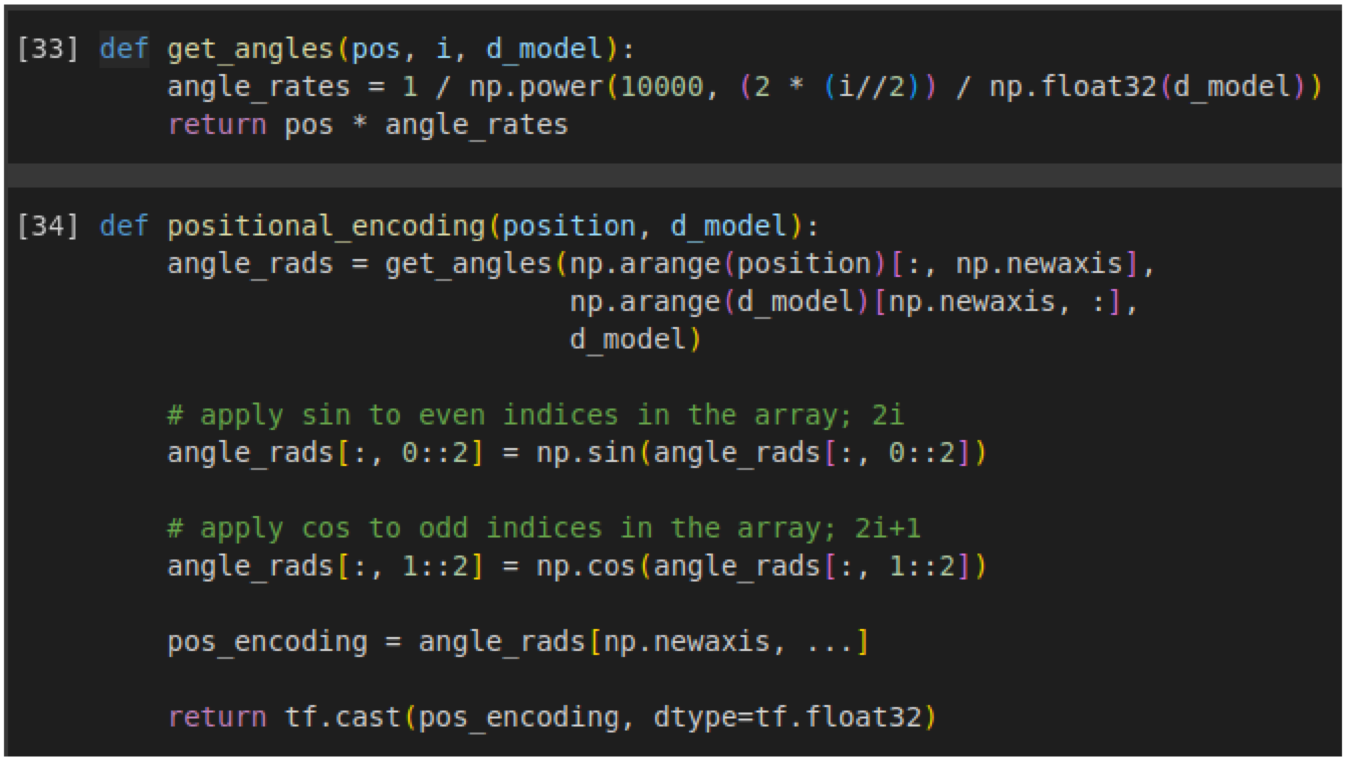
Encoding Function: The code likely uses a mathematical function (e.g., sine or cosine) to generate positional encoding vectors. These vectors have the same dimensionality as the word embeddings.

Positional Information: The function takes the position of a word within the sequence as input and uses it to calculate the encoding vector. Different positions will have distinct encodings.

Embedding Addition: The positional encoding vector is then added element-wise to the word embedding for each word in the sequence. This injects positional information into the word's representation.

In math word problems, word order is crucial. **For example,** "2 / 3" has a different meaning than "3 / 2."

Positional encoding allows the model to differentiate between these cases by altering the word embeddings based on their positions. The model can then learn the **importance of order** when generating the expression.



```
[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
```

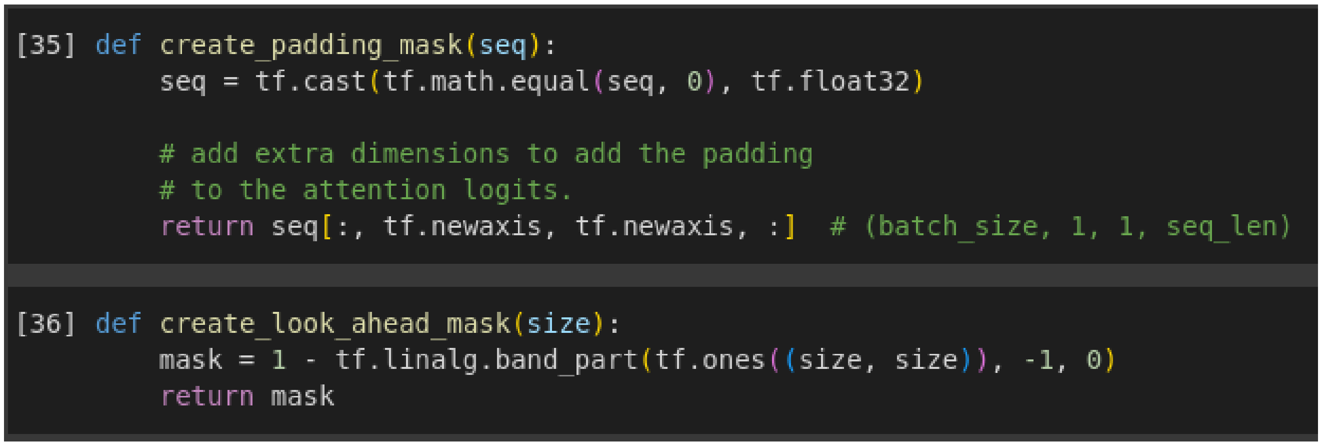
```
[34] def positional_encoding(position, d_model):
         angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                                 np.arange(d_model)[np.newaxis, :],
                                 d_model)

         # 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 is the position 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.

Padding Mask: The padding mask addresses the issue of variable sequence lengths in the data. In real-world scenarios, math word problems can have different lengths. To train the model efficiently, sequences are often padded with special tokens (e.g., zeros) to ensure a fixed length.

Look-Ahead Mask (Decoder Only): The look-ahead mask is specific to the decoder part of the Transformer model. In math word problems, the **solution (expression) depends on the problem itself, not on future words** in the solution being generated. This structure prevents the decoder from attending to future words in the expression during generation.



```
[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)
```

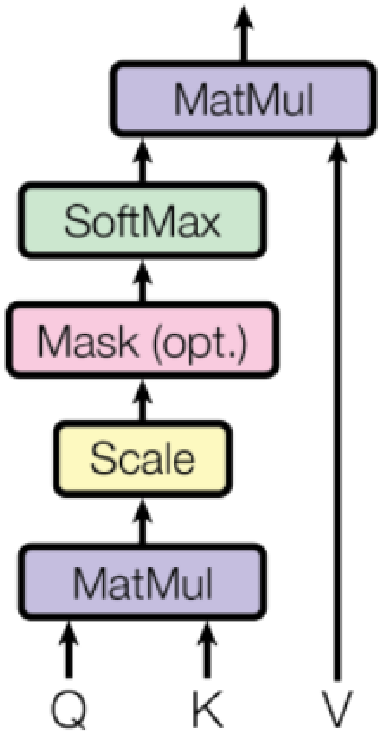
```
[36] def create_look_ahead_mask(size):
         mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
         return mask
```

Scaled Dot-product Attention

Input Preparation: The Transformer blocks receive queries (encoder) or queries and keys (decoder) as input. These are vectors representing individual words or concepts.

Dot Product: The scaled dot-product attention calculates the compatibility score **between each query and every key** using a dot-product operation. This score indicates how relevant a particular key (word in the problem or encoder output) is to the current query (the word being processed in the decoder or encoder).

Scaling: The raw dot product values are scaled down by the square root of the key vector dimension. This helps prevent exploding gradients during training and improves the stability of the model.

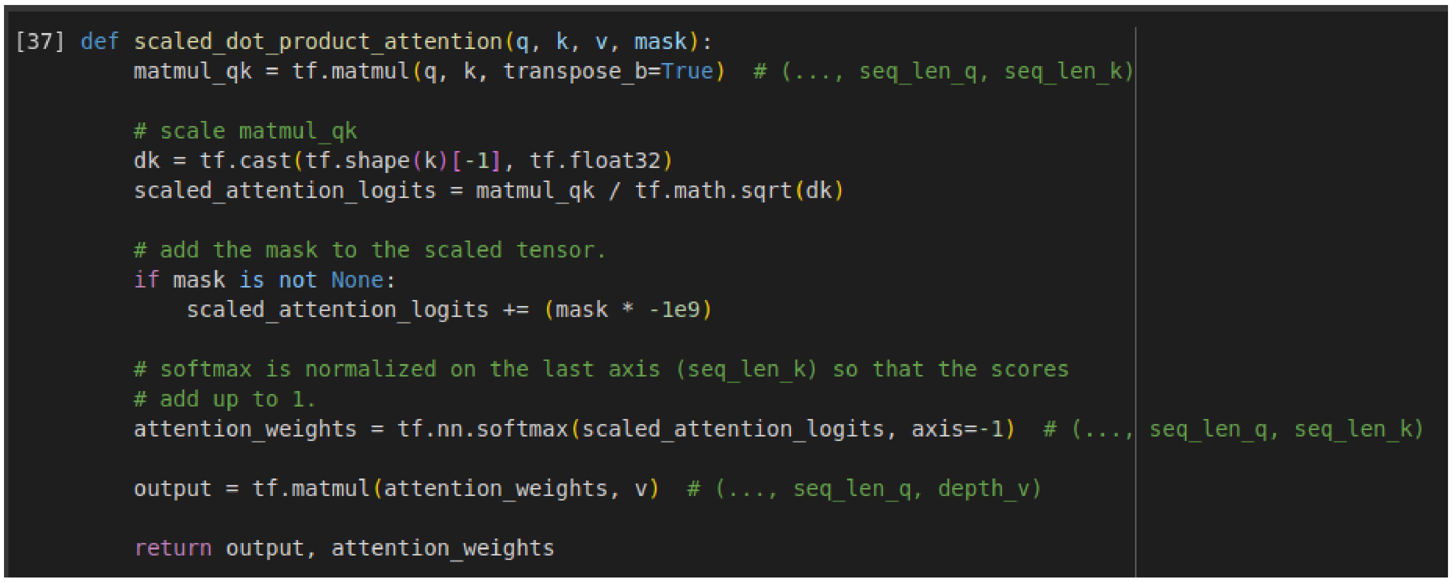


In the context of math word problems, the query might represent the current word in the problem being processed by the decoder. The keys would represent words from the encoder's output (the encoded problem representation).

The scaled dot-product attention allows the model to focus on the most relevant parts of the problem (keys) for generating the mathematical expression (based on the current query word in the decoder).

For example, when translating the phrase "five apples plus three oranges," the attention might assign higher weights to "five" and "apples" for the query word "plus," reflecting their importance in calculating the sum.

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

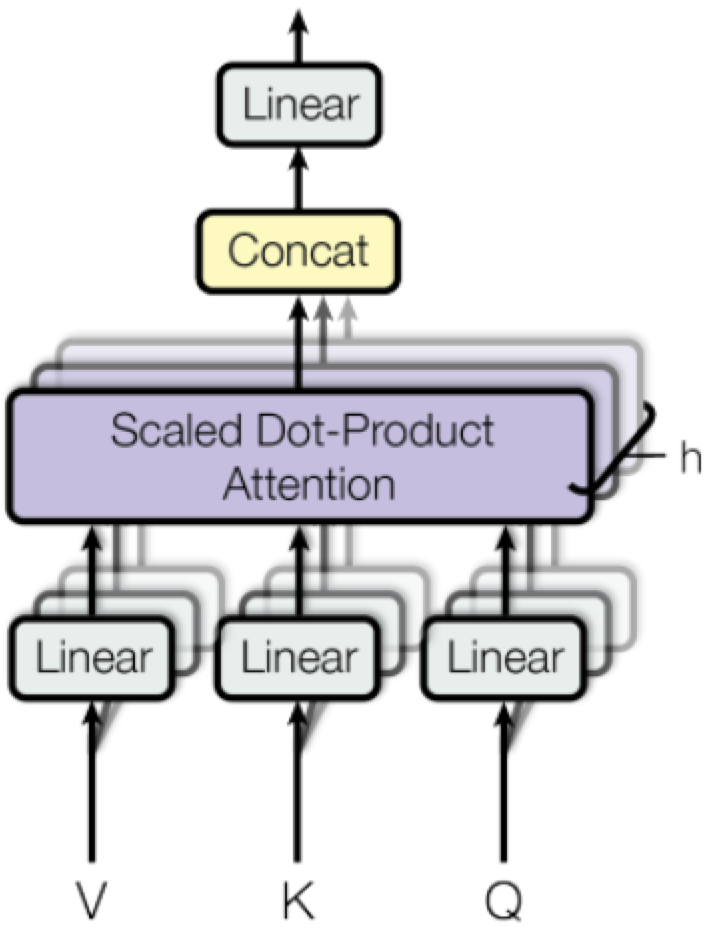


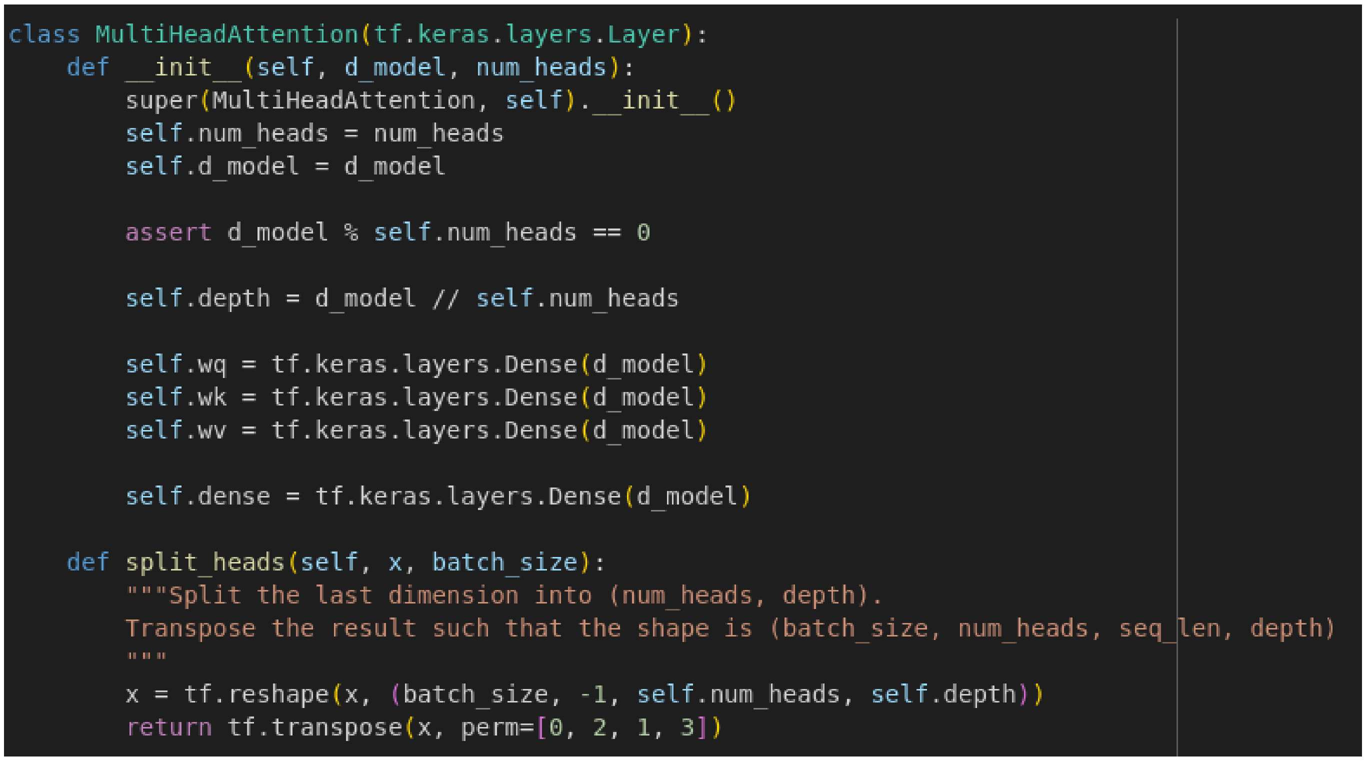
```
[37] def scaled_dot_product_attention(q, k, v, mask):
         matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

         # 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
```

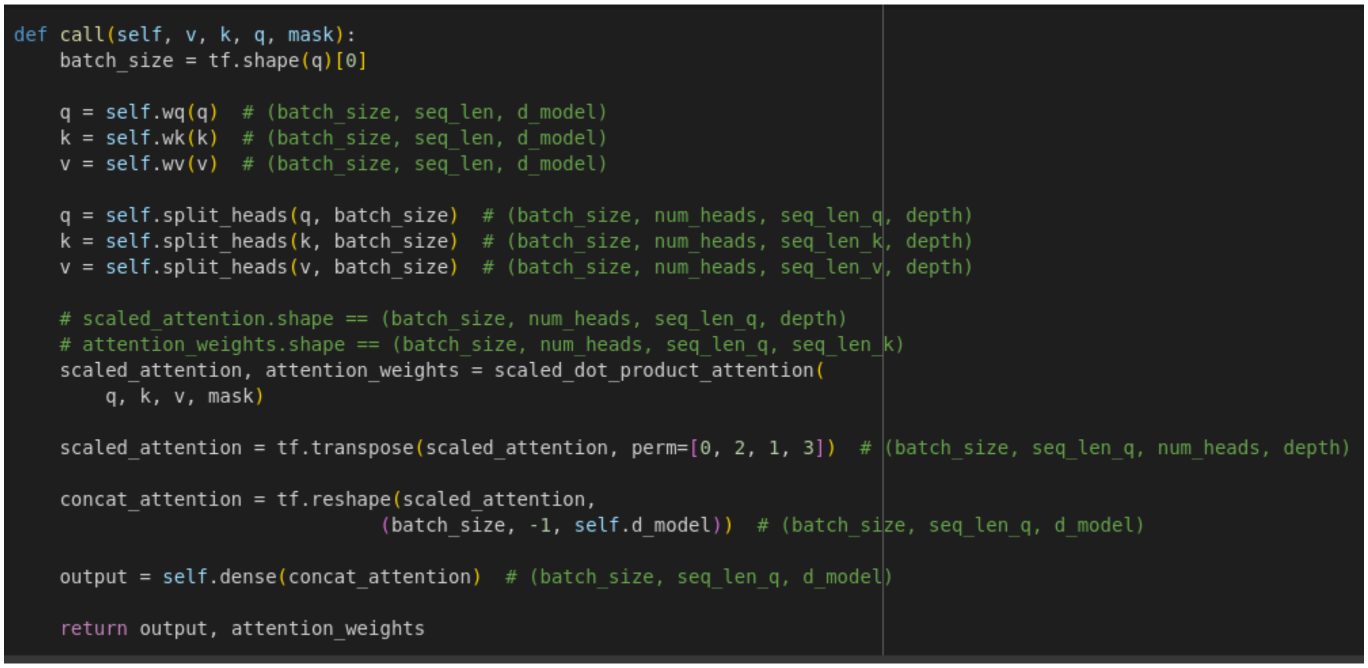
Multi-Head Attention

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

Analyzes **relationships between all tokens within the current sequence (encoder) or between the decoder's input and the encoder's output.** This allows the model to understand how different parts of the problem and expression relate to each other.




```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])
```


```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
```

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.

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

Transformers and FFNs: Transformer blocks, the core building blocks of the encoder and decoder in the model, rely on two key mechanisms:

- **Multi-head Self-Attention (MHSA):** Captures relationships between words within a sequence.
- **Feed Forward Neural Network (FFN):** Introduces non-linearity and allows the model to learn more complex patterns beyond just attention.

Functionality of FFNs in Transformers:

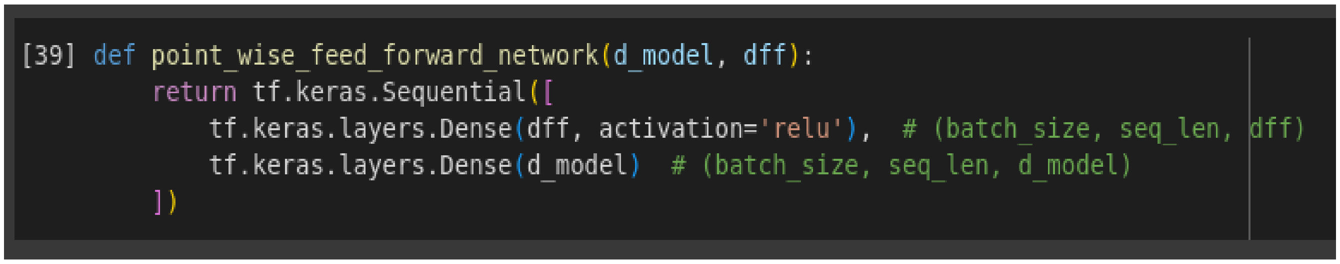
- **Structure:** The FFN is typically a simple multi-layer perceptron (MLP) with one or two hidden layers.
- **Input:** The FFN takes the output from the MHSA layer as input. This output is a context vector that summarizes the relevant information from surrounding words.
- **Hidden Layers:** The input vector is passed through one or two hidden layers with non-linear activation functions (e.g., ReLU). These layers allow the model to learn complex, non-linear relationships between the words in the sequence.
- **Output:** The final layer of the FFN produces a new vector with the same dimensionality as the input. This vector can be interpreted as a refined representation of the context, potentially capturing higher-level features or interactions not solely captured by attention.

Benefits in Math Word Problem Translation:

In math word problems, the FFN within the Transformer block can help the model learn intricate relationships between words beyond just their positional context.

For example, the FFN might learn that "plus" and "minus" have opposite meanings, even though they might be close together in the sentence.

This capability complements the MHSA mechanism, allowing the model to capture both **long-range dependencies (through attention)** and **non-linear interactions between words (through FFNs)**.



```
[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

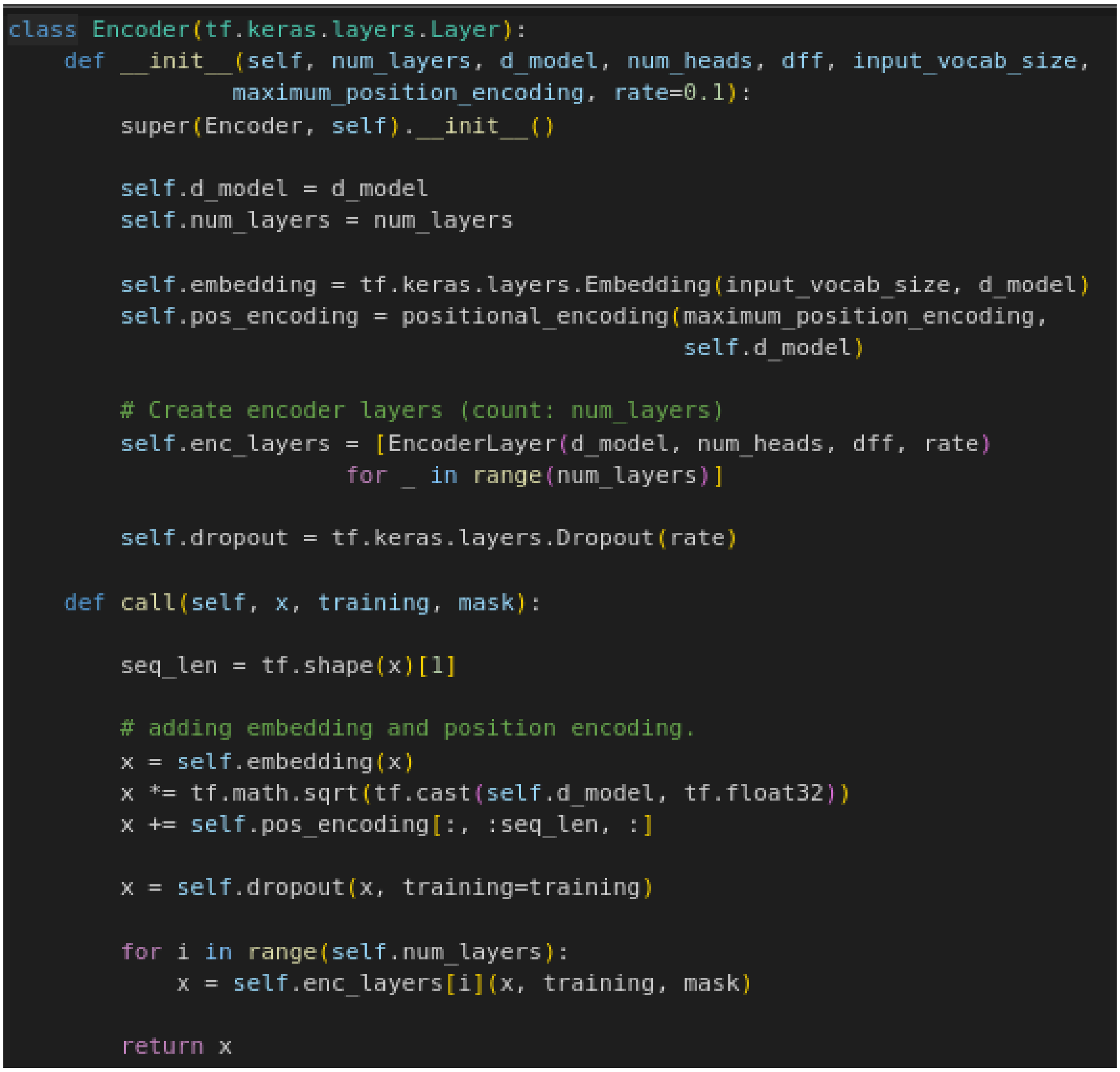
The encoder consists of N layers and each layer has 2 sub-layers. The first sublayer is a multi-head self-attention mechanism layer and the second sub-layer is a position-wise fully connected feed-forward neural network. Each of these sub-layers is preceded by a skip connection or residual connection 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.

$$Encoder(x) = LayerNorm(x + Sublayer(x))$$

This is the first part of the Transformer model. It takes the input sequence (the math worded problem) as a series of tokens. It stacks multiple TransformerEncoderLayer objects, where each layer performs:

- **Multi-head self-attention:** Analyzes relationships between all words within the problem sentence to understand the context of each word.
- **Feed-forward neural network (FFN):** Introduces non-linearity to capture more complex relationships beyond just attention.

The final output of the encoder is a contextual representation of each word in the problem, capturing its **meaning and relationships with other words**.



```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)

        # 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

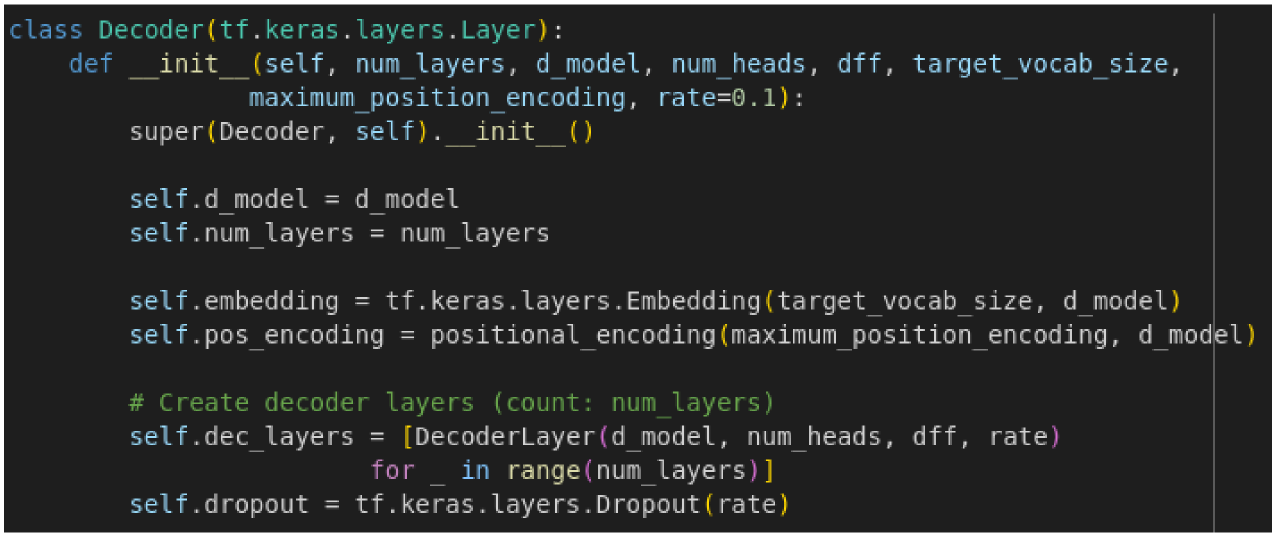
The decoder consists of N layers and each layer has 3 sub-layers. Two of the sub-layers are the same as those of the encoder. The third sub-layer utilizes the outputs of the encoder stack and performs multi-head attention over them. Decoder feeds on two types of inputs which are, the outputs of the encoder and positionally encoded target output embeddings. And just like the encoder, the 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**.

It takes two inputs:

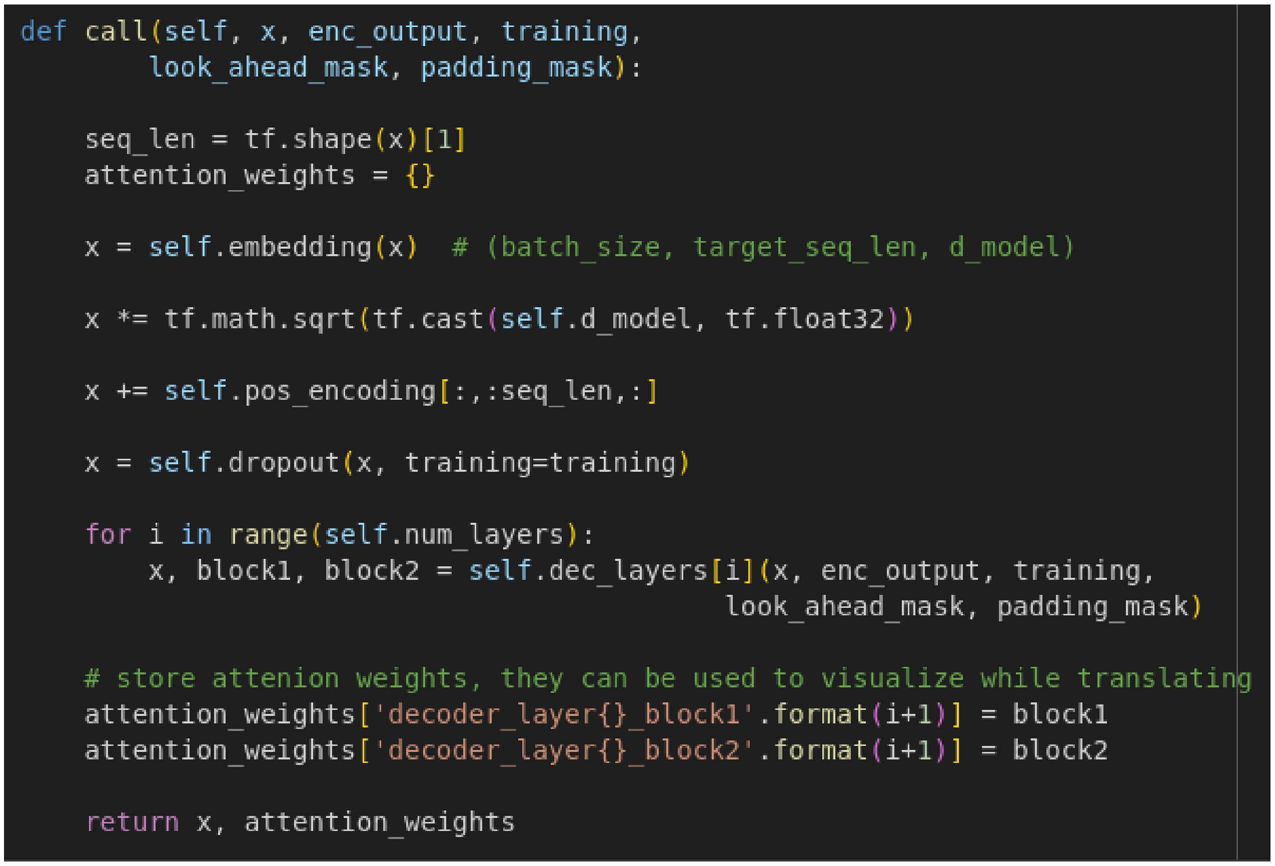
- The encoded representation of the problem from the encoder.
- The target sequence (the mathematical expression) is one token at a time (during training) or starts with a special start token (during prediction).

It stacks multiple TransformerDecoderLayer objects, where each layer performs:

- **Masked multi-head attention:** Similar to self-attention but prevents the decoder from attending to future words in the generated expression (important for correct order).
- **Multi-head attention to the encoder output:** Allows the decoder to attend to relevant parts of the encoded problem representation for generating each token in the expression.
- **FFN:** Similar to the encoder's FFN.



```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)

        # 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)
```



```python
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
```

Encoder Layer and Decoder Layer:

These are the fundamental building blocks of the encoder and decoder, respectively. As mentioned above, they encapsulate the core functionalities:

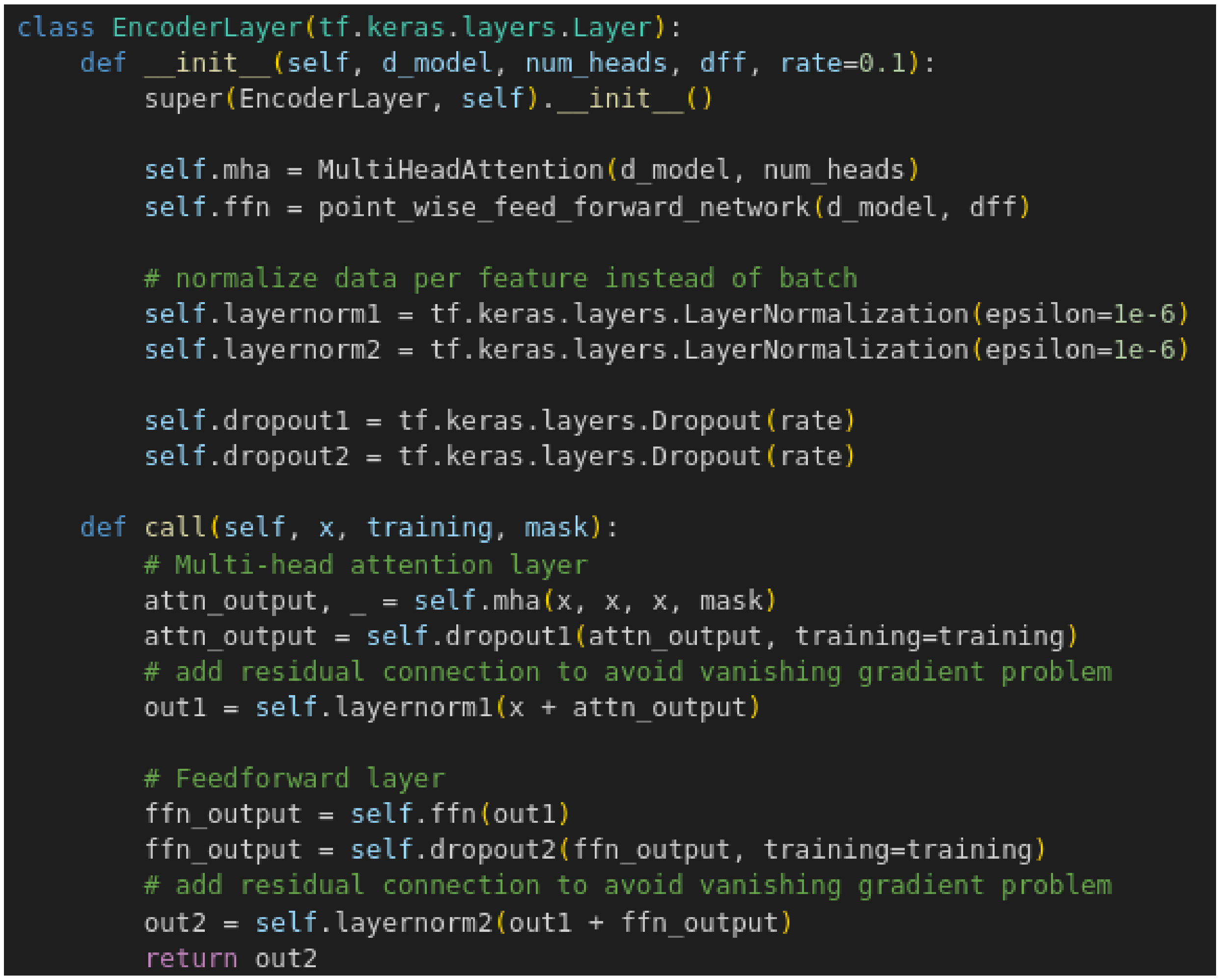
Multi-head self-attention (encoder) or masked multi-head attention (decoder): Analyzes relationships between words.

Feed-forward neural network (FFN): Introduces non-linearity for complex pattern learning.

Residual connections and layer normalization: Techniques to improve training stability and gradient flow.

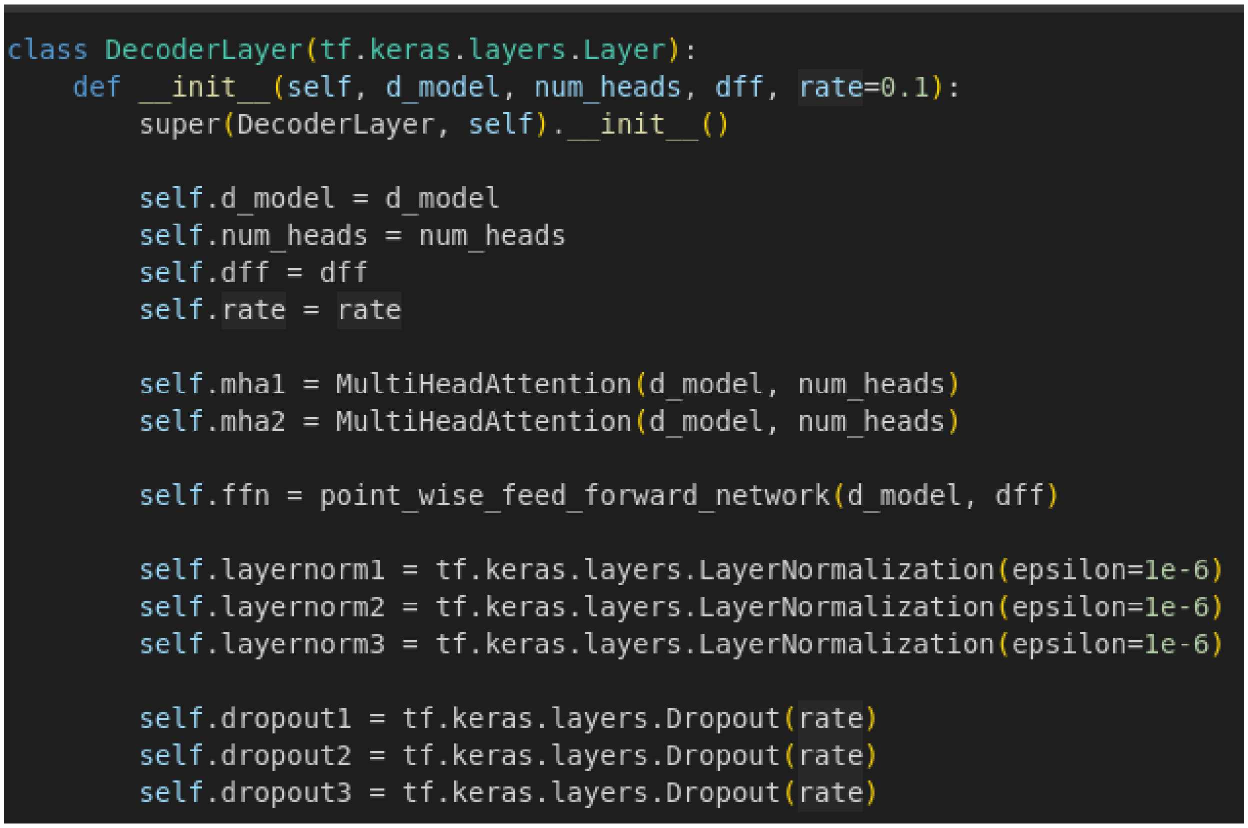
Overall, these components work together in the Transformer model to achieve the task of translating math-worded problems into mathematical expressions. The **encoder captures the context of the problem**, and the **decoder uses that context along with its attention mechanism to generate the expression** one step at a time.

Encoder Layer:

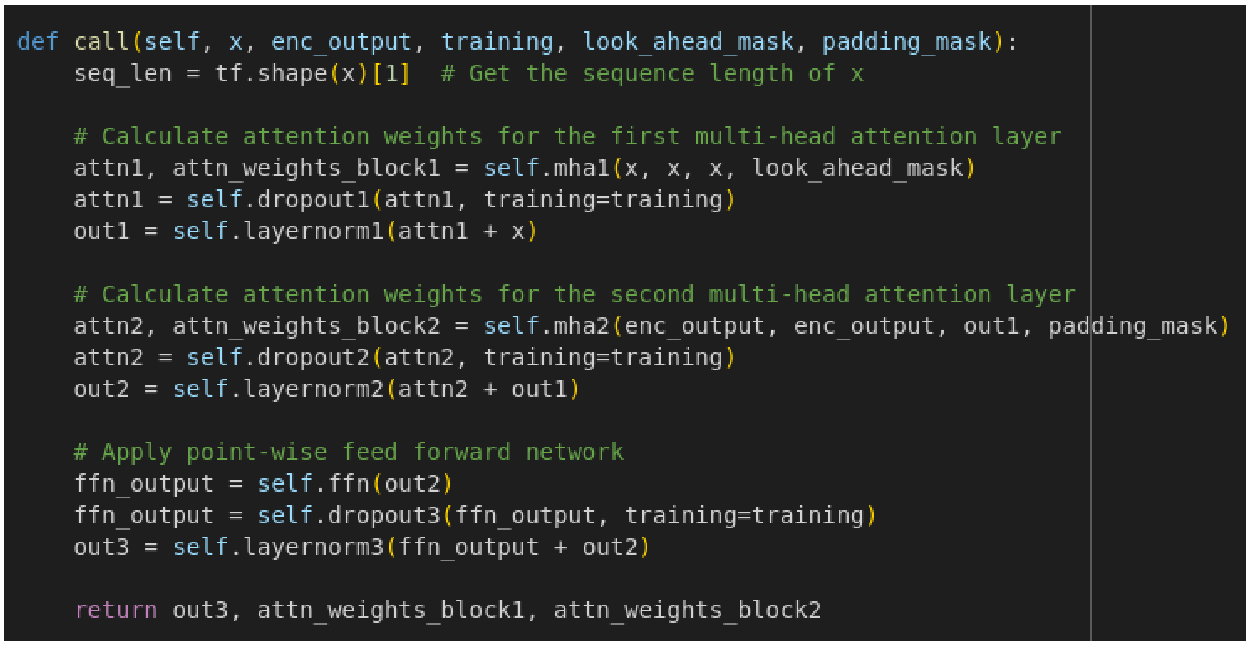


```python
class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(EncoderLayer, self).__init__()

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

        # 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
```

Decoder Layer:



```python
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)
```



```python
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
```

Transformer

The intuition behind the transformer model for converting math word problems to equations relies on two key concepts:

Sequential Encoding and Attention:

- The model treats the word problem as a **sequence of tokens**, similar to a sentence. The encoder processes this sequence to capture the **meaning and relationships between words**. This creates a condensed representation encoding the problem's structure.
- The decoder, **during equation generation**, uses an attention mechanism. This allows it to focus on specific parts of the encoded-word problem (encoder output) that are relevant to predicting the next token in the equation sequence.
- As the decoder generates each token (like a variable or operation symbol), it attends to the most relevant parts of the encoded problem representation, effectively using context to build the equation step-by-step.

Learning from Examples:

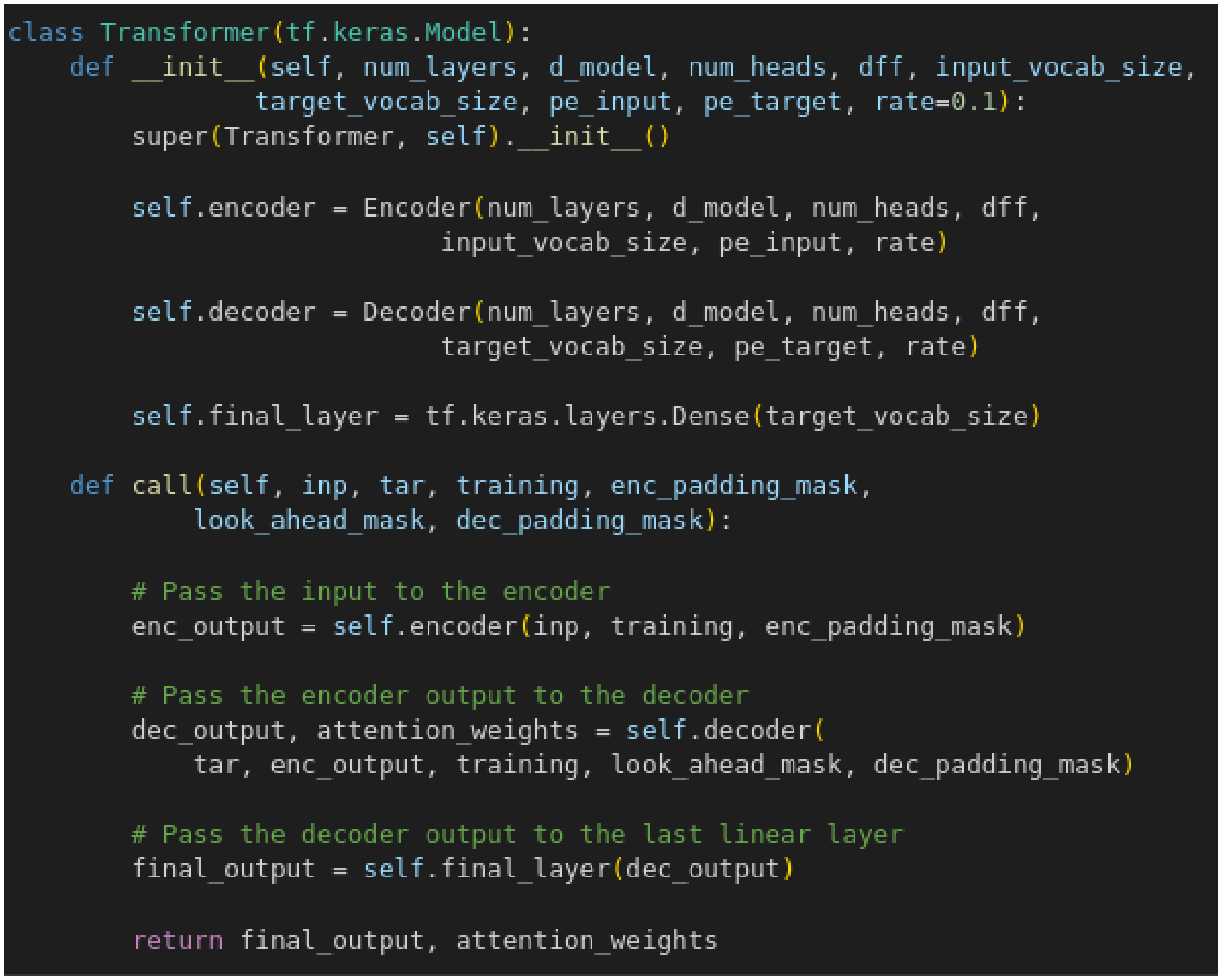
- The transformer model **doesn't have pre-programmed knowledge** of math or equation structures. It **learns the mapping between word problems and their corresponding equations** by training on a large dataset of paired examples. Each example consists of a word problem and its corresponding equation.
- During training, the model adjusts its internal weights and biases to improve its ability to predict the correct equation token sequence based on the encoded-word problem representation.
- Essentially, the model leverages the power of sequential processing and attention to mimic how humans might approach a word problem. We read and understand the problem, focusing on relevant parts (like numbers and keywords), and then use that understanding to construct the equation step-by-step. The transformer model replicates this process through its architecture and learning from vast amounts of data.

Here are some additional points to consider:

The model doesn't have a symbolic understanding of math. It learns the relationships and patterns between word problem representations and equations based on the training data.

The quality of the generated equations depends on the quality and size of the training dataset. The more comprehensive the training data, the better the model can generalize to unseen problems.

I hope this explanation provides a clearer picture of the intuition behind the transformer model for math word problem to equation conversion.



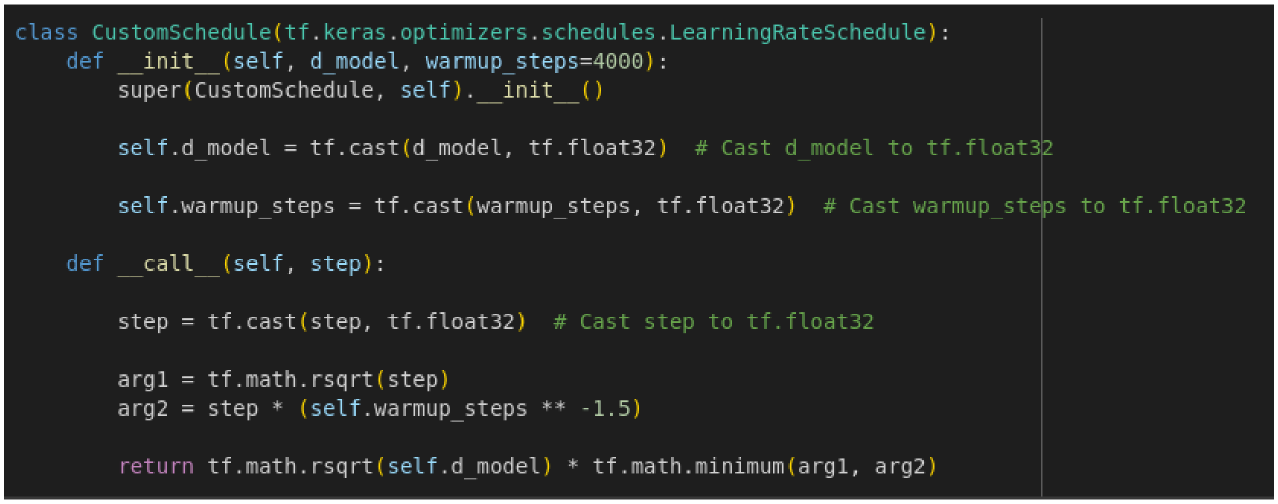
```python
class Transformer(tf.keras.Model):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
                 target_vocab_size, pe_input, pe_target, rate=0.1):
        super(Transformer, self).__init__()

        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
```

Optimizer

We used Adam with a custom learning rate scheduler as our optimizer. The learning rate was set according to,

$$lrate = d_{model}^{-0.5} \cdot \min(n^{-0.5}, n \cdot w^{-1.5})$$

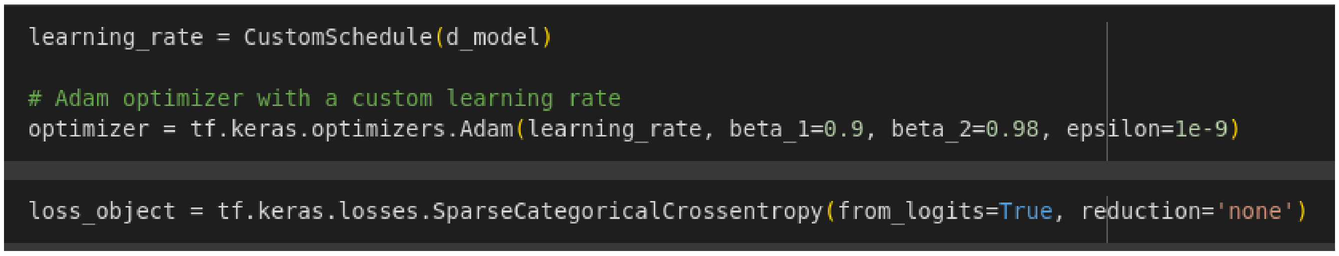
Where ‘d-model’ is the embedding dimension, ‘n’ is the step number and ‘w’ is the number of warm-up steps. Here, warm-up steps ‘w’ simply insinuates that the learning rate rises linearly for the initial ‘w’ training steps.



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

        self.d_model = tf.cast(d_model, tf.float32)  # Cast d_model to tf.float32

        self.warmup_steps = tf.cast(warmup_steps, tf.float32)  # Cast warmup_steps to tf.float32

    def __call__(self, step):

        step = tf.cast(step, tf.float32)  # Cast step to 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)
```

Learning Rate: This hyperparameter controls the step size taken by the optimizer when updating the weights. A custom schedule defines how this learning rate changes.

Loss Object: This function calculates the difference between the model's predictions and the actual target expressions (mathematical expressions).



```
learning_rate = CustomSchedule(d_model)

# Adam optimizer with a custom learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
```

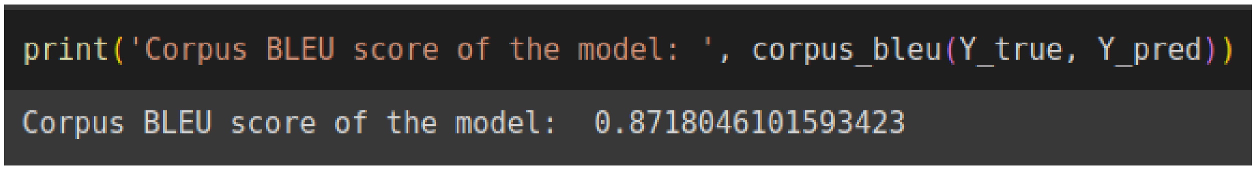
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')
```

Performance Metric

As output sequences are padded, it is important to apply a padding mask when calculating the loss. We used Sparse Categorical Cross-entropy for loss and Mean for accuracy. In the test-phase we also calcuated the **BLEU (BiLingual Evaluation Understudy) score** of our model to assess its translation quality. The BLEU score is computed as,

$$\text{BLEU} = \underbrace{\min\left(1, e^{1-\frac{l_{reference}}{l_{output}}}\right)}_{\text{brevity penalty}} \cdot \underbrace{\left(\prod_{i=1}^{4} precision_i\right)^{\frac{1}{4}}}_{\text{n-gram overlap}}$$

$$precision_i = \frac{\sum_{s\in\text{Cand-Corpus}} \sum_{i\in s} \min(m^i_{cand}, m^i_{ref})}{w^i_t}$$

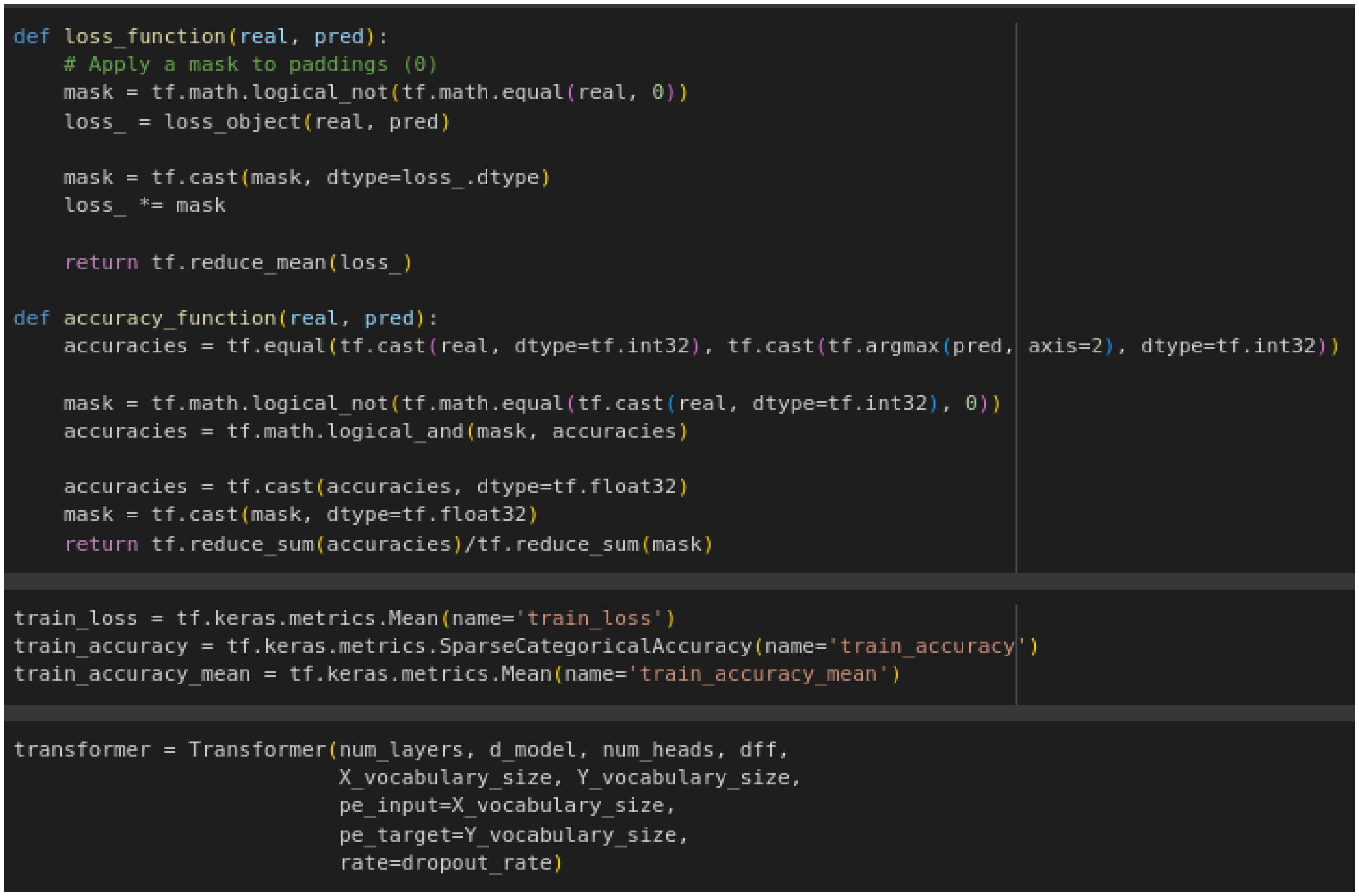


```
print('Corpus BLEU score of the model: ', corpus_bleu(Y_true, Y_pred))
```

```
Corpus BLEU score of the model:  0.8718046101593423
```

Loss Function: This function, used during training, measures the discrepancy between the model's predicted expressions and the actual expressions.

Accuracy Function: While not essential for this specific task, the code might include an accuracy function to evaluate the model's performance on a separate validation set. Accuracy can be measured in terms of correctly generated expressions or character-level accuracy.

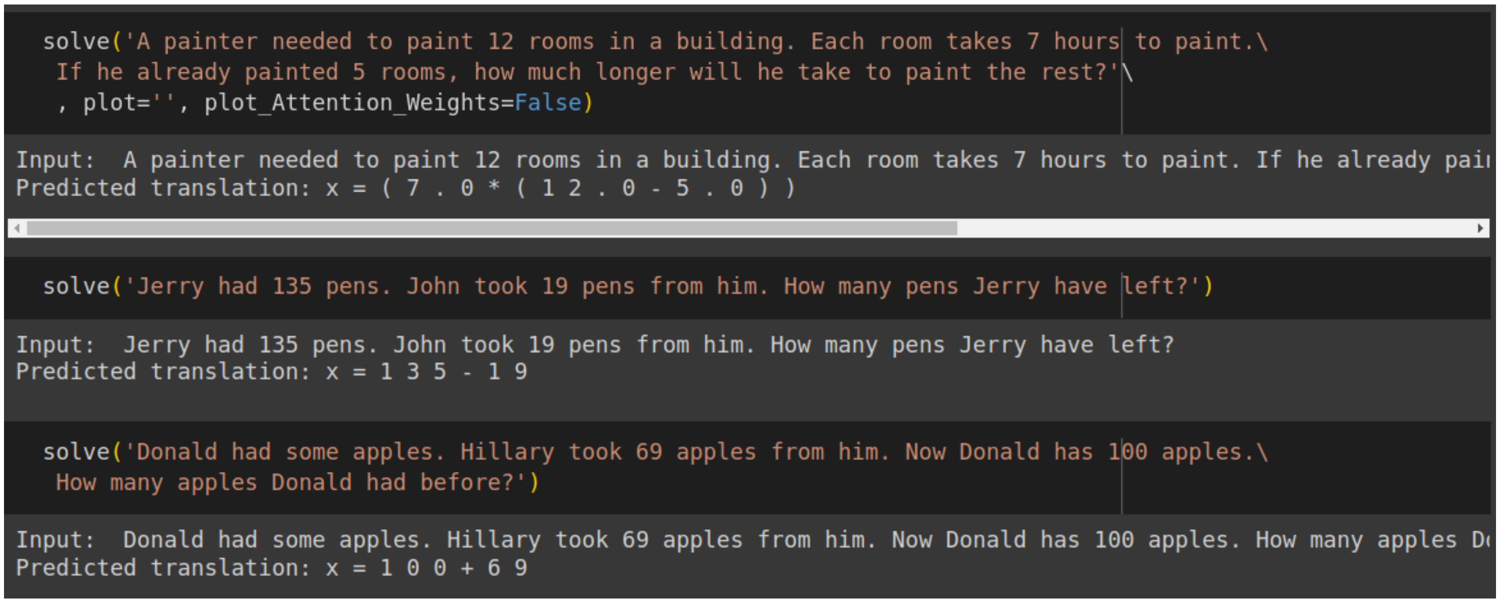


```python
def loss_function(real, pred):
    # Apply a mask to paddings (0)
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

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

    return tf.reduce_mean(loss_)

def accuracy_function(real, pred):
    accuracies = tf.equal(tf.cast(real, dtype=tf.int32), tf.cast(tf.argmax(pred, axis=2), dtype=tf.int32))

    mask = tf.math.logical_not(tf.math.equal(tf.cast(real, dtype=tf.int32), 0))
    accuracies = tf.math.logical_and(mask, accuracies)

    accuracies = tf.cast(accuracies, dtype=tf.float32)
    mask = tf.cast(mask, dtype=tf.float32)
    return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)
```

```python
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')
train_accuracy_mean = tf.keras.metrics.Mean(name='train_accuracy_mean')
```

```python
transformer = Transformer(num_layers, d_model, num_heads, dff,
                          X_vocabulary_size, Y_vocabulary_size,
                          pe_input=X_vocabulary_size,
                          pe_target=Y_vocabulary_size,
                          rate=dropout_rate)
```

Screenshot of Implementation



```
solve('A painter needed to paint 12 rooms in a building. Each room takes 7 hours to paint.\
 If he already painted 5 rooms, how much longer will he take to paint the rest?'\
 , plot='', plot_Attention_Weights=False)
```

```
Input:  A painter needed to paint 12 rooms in a building. Each room takes 7 hours to paint. If he already pai
Predicted translation: x = ( 7 . 0 * ( 1 2 . 0 - 5 . 0 ) )
```

```
solve('Jerry had 135 pens. John took 19 pens from him. How many pens Jerry have left?')
```

```
Input:  Jerry had 135 pens. John took 19 pens from him. How many pens Jerry have left?
Predicted translation: x = 1 3 5 - 1 9
```

```
solve('Donald had some apples. Hillary took 69 apples from him. Now Donald has 100 apples.\
 How many apples Donald had before?')
```

```
Input:  Donald had some apples. Hillary took 69 apples from him. Now Donald has 100 apples. How many apples Do
Predicted translation: x = 1 0 0 + 6 9
```

Drive Link

https://drive.google.com/drive/folders/1uzedq3q9kZBOnrxs3PmgtggxGje1mHZt?usp=sharing

This drive folder link contains code for our project named "mathiq.ipynb" with the dataset named "MathIQ_Dataset - Sheet1.csv" and all four mandates.

Problems Faced

- Trained on small dataset.
- Struggles in problems that require multiple steps and > 2 operators.
- Uses tokens of digits, not whole numbers. Output can dramatically change for only changing a number in whole problem.
- Can produce erroneous outputs if statement's grammar is slightly changed or if the given problem statement deviates too much from the structure of the problems in the training set.

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