Transformers

Presented by Allison Buck

What are they?

- Transformers are a type of neural network architecture that transforms or changes an input sequence into an output sequence
- They do this by learning context and tracking relationships between sequence components
- Consider this input sequence: **"What is the color of the sky?"** The transformer model uses an internal mathematical representation that identifies the relevancy and relationship between the words color, sky, and blue. It uses that knowledge to generate the output: **"The sky is blue."**

Real World Applications

1

Autocomplete feature in an iPhone that suggests words based on what you normally type

EX: if you frequently type "I am fine," your phone autosuggests fine after you type am.

2

Translation applications that use transformers to provide immediate, accurate translations between languages

EX: asking Google Translate how to say “hello my name is” in Spanish

3

DNA sequence analysis can be done using transformers that predict the effects of genetic mutations, understand patterns, and identify regions of DNA responsible for certain diseases

Why use it over other models?

LSTMs and RNNs drawbacks:

- Difficult to parallelize when processing sentences, since you have to process word by word
- No modeling of long and short range dependencies
- When sentences are long, these models tend to forget the content of more distant positions in the sequence
  - There are gaps between relevant information and the point where it is needed
  - The longer the sentence is, the easier it is to lose information from the beginning of it

Why use it over other models?

CNNs:

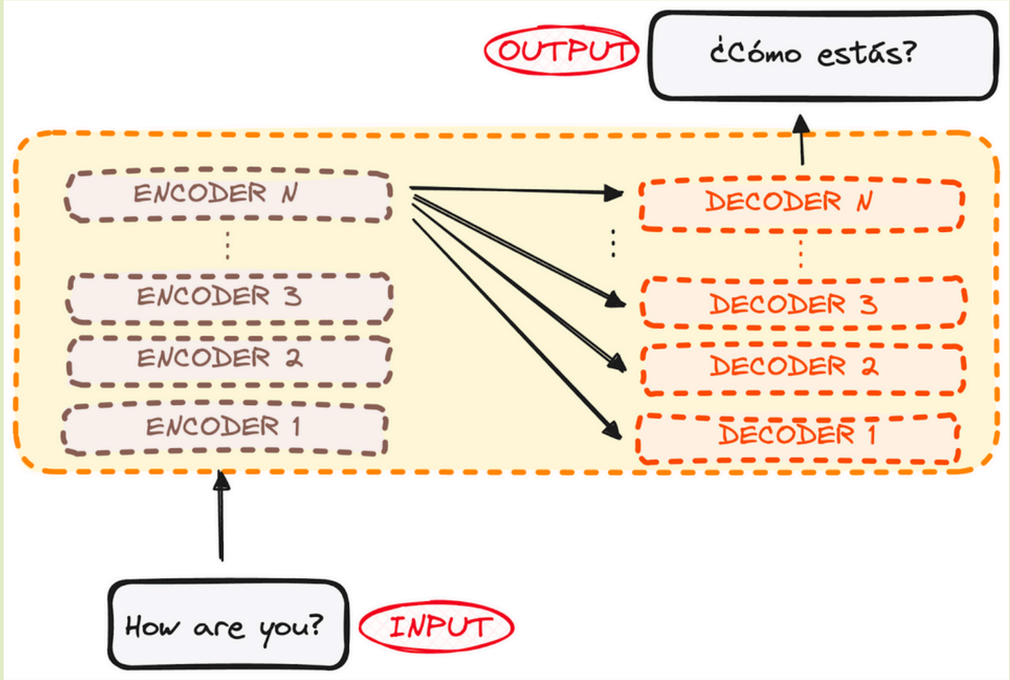
- Can work in parallel, each word in the input can be processed at the same time and does not have to depend on the previous words to be translated
- Distance between positions is logarithmic (using a tree with layers rather than linear)
- Do **NOT** help with the problem of dependencies when translating

Transformers:

- Combination of CNNs with attention

Encoders and Decoders

- The encoder takes in our input and outputs a matrix representation of that input. For instance, the English sentence “How are you?”
- The decoder takes in that encoded representation and iteratively generates an output. In our example, the translated sentence “¿Cómo estás?”
- Multiple layers for both



Attention

- Attention boosts the speed of how fast the model can translate from one sequence to another
- It helps the encoder look at other words in the input sentence as it encodes a specific word
- having an attention layer first helps the decoder then focus on relevant parts of the input sentence
- For instance, in a given example, the model might learn to connect the word “are” with “you”

Steps for Encoding

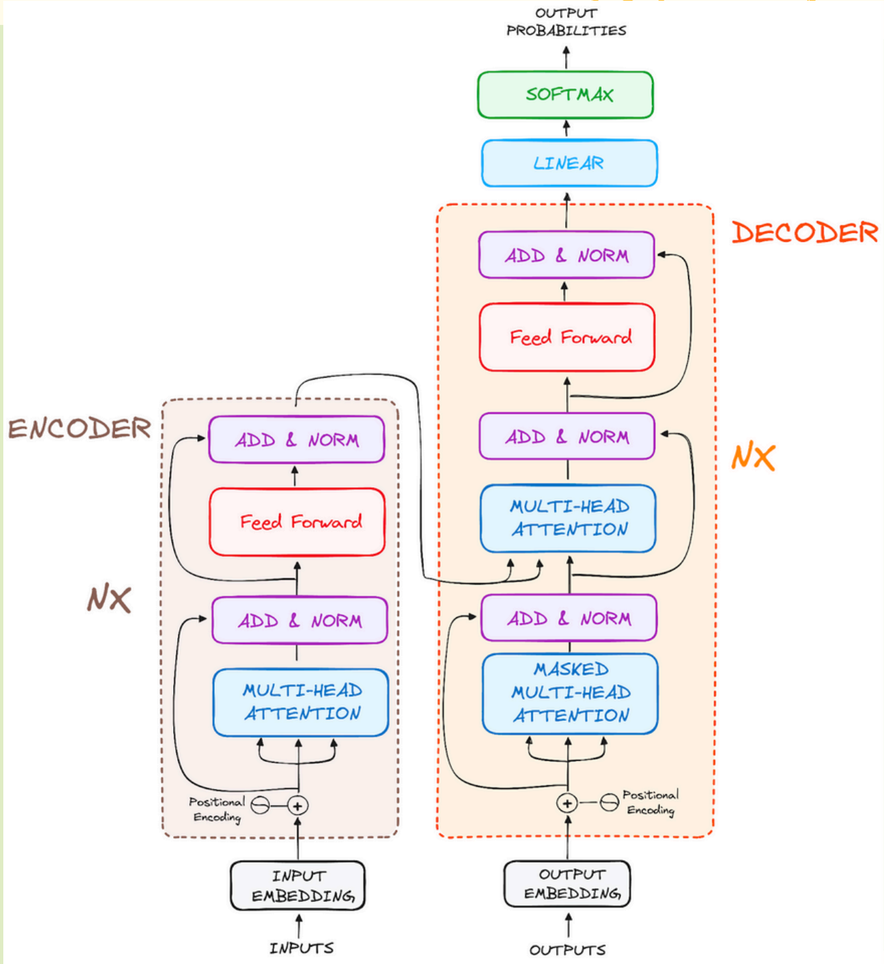
- Embed each word in a vector with positional context
- Create query, key, and value vector for each word in input sentence
- Once they are passed through a linear layer, a dot product matrix multiplication is performed between the queries and keys, resulting in the creation of a score matrix
    - The score matrix establishes how much emphasis each word should place on other words
- Scale the scores by dividing by the dimension of the query and key vectors
- Normalize them using a softmax operation
- Multiply these scores by the value vector (helps drown out irrelevant words)
- In this process, only the words that present high softmax scores are preserved
- Send to feed forward layer
- This encoding paves the way for the decoder, guiding it to pay attention to the right words in the input when it's time to decode

Steps for Decoding

- The decoder uses a list of previously generated outputs as its inputs alongside the outputs from the encoder that consider attention information from the initial input
- First attention layer:
  - Score matrix is formed similar to in encoding but with a crucial difference: it prevents words from looking to those in positions ahead
    - Ensures that each word in the sequence isn't influenced by future words
- Cross attention layer:
  - The outputs from the encoder take on the roles of both queries and keys, while the outputs from the first attention layer of the decoder serve as values
- Send to feed forward layer
- The final of the decoder's process involves a linear layer, serving as a classifier, topped off with a softmax function to calculate the probabilities of different words
  - The classes are each of the words contained in the vocabulary

Steps for Decoding

- The highest of these probability scores is key, its corresponding index directly points to the word that the model predicts as the next in the sequence
- Such a multi-layered approach can significantly enhance the model's ability to predict, as it develops a more nuanced understanding of different attention combinations.



Data

I used the alice in wonderland text from a previous RNN model example that we did in class to predict on

I sampled 3000 words from the text (to make the training go faster) and pulled out sequences of 20 words to use as the inputs

```
# load text and covert to lowercase
filename = "alice_in_wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
raw_text = raw_text[:3000]
```

Setting up data

```
# create tokenizer to convert from string to integers
tokenizer = tf.keras.preprocessing.text.Tokenizer()
tokenizer.fit_on_texts([raw_text])

# convert text to sequences
sequences = tokenizer.texts_to_sequences([raw_text])[0]

# define parameters
vocab_size = len(tokenizer.word_index) + 1
seq_length = 20
```

```
# prepare the dataset of input to output pairs encoded as integers
input = []
output = []

for i in range(seq_length, len(sequences)):
    input.append(sequences[i-seq_length:i])
    output.append(sequences[i])
```

```
# convert to arrays
X = np.array(input)
y = np.array(output)
```

- change all strings to integers
- pull out sequences for input
- find the vocab size and establish sequence length
- put input and output data into arrays

Multi-Head Self-Attention

```python
class MultiHeadSelfAttention(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        self.embed_dim = embed_dim # model's dimension
        self.num_heads = num_heads # number of attention heads
        self.dept = embed_dim // num_heads # dimension of each head's vectors

        # layers for query, key, and value vectors
        self.wq = tf.keras.layers.Dense(embed_dim)
        self.wk = tf.keras.layers.Dense(embed_dim)
        self.wv = tf.keras.layers.Dense(embed_dim)
        self.dense = tf.keras.layers.Dense(embed_dim) # final classification layer

    # reshape input to have num_heads for multi-head attention
    def split_heads(self, x, batch_size):
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.dept))
        return tf.transpose(x, perm=[0, 2, 1, 3])
```

Multi-Head Self-Attention

```python
# finding attention scores
def call(self, inputs):
    batch_size = tf.shape(inputs)[0]

    # create vectors from input sequences
    q = self.wq(inputs)
    k = self.wk(inputs)
    v = self.wv(inputs)

    # split into multiple heads
    q = self.split_heads(q, batch_size)
    k = self.split_heads(k, batch_size)
    v = self.split_heads(v, batch_size)

    # calculate attention scores from vectors above
    scaled_attention_logits = tf.matmul(q, k, transpose_b=True) / tf.sqrt(tf.cast(self.dept, tf.float32)) # scaling
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1) # normalize

    # attention weights x values
    output = tf.matmul(attention_weights, v)
    output = tf.transpose(output, perm=[0, 2, 1, 3])
    output = tf.reshape(output, (batch_size, -1, self.embed_dim))
    return self.dense(output) # return classification layer
```

Transformer Block

```python
class TransformerBlock(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim):
        super(TransformerBlock, self).__init__()

        # attention layer and feed forward network
        self.attention = MultiHeadSelfAttention(embed_dim, num_heads)
        self.ffn = tf.keras.Sequential([
            tf.keras.layers.Dense(ff_dim, activation='relu'),
            tf.keras.layers.Dense(embed_dim)
        ])

        # normalize and dropout layers
        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = tf.keras.layers.Dropout(0.1)
        self.dropout2 = tf.keras.layers.Dropout(0.1)

    # get outputs for attention and feed forward layers
    def call(self, inputs, training):
        attn_output = self.attention(inputs)
        out1 = self.layernorm1(inputs + self.dropout1(attn_output, training=training))
        ffn_output = self.ffn(out1)
        return self.layernorm2(out1 + self.dropout2(ffn_output, training=training))
```

- send inputs through all layers
  - Attention
  - Feed Forward Network
  - Layer Normalization
  - Dropout
- will allow model to learn information about the input data

The Model

```python
class TransformerModel(tf.keras.Model):
    def __init__(self, input_dim, embed_dim, num_heads, num_layers, output_dim):
        super(TransformerModel, self).__init__()
        self.embedding = tf.keras.layers.Embedding(input_dim, embed_dim) # embedding layer
        self.transformer_blocks = [TransformerBlock(embed_dim, num_heads, embed_dim * 4) for _ in range(num_layers)] # transformer blocks
        self.fc = tf.keras.layers.Dense(output_dim) # output layer

    def call(self, x, training=False):
        embedded = self.embedding(x) # convert input to embeddings
        for transformer in self.transformer_blocks:
            embedded = transformer(embedded, training=training) # pass through blocks
        output = tf.reduce_mean(embedded, axis=1) # pooling over sequence length
        return self.fc(output) # final output
```

Build, Compile, Train

```
# model
model = TransformerModel(vocab_size, embed_dim, num_heads, num_layers, vocab_size)
```

```
# compile model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# train model
model.fit(X, y, epochs=100, batch_size=32, validation_split=0.2)
```

Example

```
# example prediction
input_example = np.array(X[110:111])
predicted = model.predict(input_example)
predicted_word_index = np.argmax(predicted, axis=-1)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step
```

```
# get the actual
predicted_word = tokenizer.index_word[predicted_word_index[0]]
input_indices = input_example[0]
actual_next_index = y[110]
actual_next_word = tokenizer.index_word[actual_next_index]

# Show results
print("Input sequence:", [tokenizer.index_word.get(index) for index in input_indices if index > 0])
print("Predicted word:", predicted_word)
print("Actual next word:", actual_next_word)
```

```
Input sequence: ['a', 'daisy', 'chain', 'would', 'be', 'worth', 'the', 'trouble', 'of', 'getting', 'up', 'and', 'picking', 'the', 'daisies',
'when', 'suddenly', 'a', 'white', 'rabbit']
Predicted word: dear
Actual next word: with
```

Transformer Model Example

- This model is a very simple Transformer model that could use some improvement
- Low accuracies:
  - Not enough epochs, dense layers, etc.
  - Only 3000 words from the text were included
  - Lack of training and testing sets

Conclusion

- By effectively managing sequential data through a unique self-attention mechanism, these models have outperformed traditional RNNs
- Their ability to handle long sequences more efficiently and parallelize data processing significantly accelerates training
- Transformers use positional encoders to keep track of data coming in and out of the network
    - Attention units follow these, calculating a kind of algebraic map of how each element relates to the others
- Self-attention finds meaning in the sequences and pays attention to what it needs to in order to predict

Sources

- https://towardsdatascience.com/transformers-141e32e69591
- https://www.datacamp.com/tutorial/how-transformers-work
- https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/
- https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch
- https://www.geeksforgeeks.org/transformer-model-from-scratch-using-tensorflow/