Transformer Practice

컴퓨터소프트웨어학부 심승현

Contents

https://www.tensorflow.org/text/tutorials/transformer

Tensorflow.org, transformer tutorial

translating Portuguese to English using transformer model

Review : Positional Encoding

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

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)
```

input parameters are both int type

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Review : Positional Encoding

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

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)
```

$$PE_{(pos,2i)} = \boxed{\sin}(\boxed{pos/10000^{2i/d_{model}}})$$

$$PE_{(pos,2i+1)} = \boxed{\cos}(\boxed{pos/10000^{2i/d_{model}}})$$

Review : Positional Encoding

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

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)
```

$$PE_{(pos,2i)} = \boxed{\sin}\left(\boxed{pos/10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \boxed{\cos}\left(\boxed{pos/10000^{2i/d_{model}}}\right)$$

dimension of return matrix is
(1, 10000, 512)

Review : Multi Head Attention

▶ First, get Query, Key, Value matrix from input matrix X

$$\boxed{\begin{aligned} Q &= X \,@\, W_Q \\ K &= X \,@\, W_K \\ V &= X \,@\, W_V \end{aligned}}$$

▶ Then, derive attention matrix Z through the formula behind

$$\boxed{Z = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right) V}$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

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

        attn_output, _ = self.mha(x, x, x, mask)
```

```python
class MultiHeadAttention(tf.keras.layers.Layer):
    def call(self, v, k, q, mask):
        batch_size = tf.sh

        q = self.wq(q)  #
        k = self.wk(k)
        v = self.wv(v)

        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)

        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)
```

q.shape
(batch_size, seq_len, d_model)

▶ First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

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

        attn_output, _ = self.mha(x, x, x, mask)
```

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

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

        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)

        scaled_attention, attention_weights = scaled_dot_product_attention(
        q, k, v, mask)
```

```python
self.wq = tf.keras.layers.Dense(d_model)
self.wk = tf.keras.layers.Dense(d_model)
self.wv = tf.keras.layers.Dense(d_model)
```

v, k, q passes different dense layers,
which have different weight matrix and bias.
In this step, Q, K, V are defined.

▶ First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{Q \cdot K^T}{\quad}\right) V$$

is dimension of matrix K.

Review : Multi Head Attention

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

        attn_output, _ = self.mha(x, x, x, mask)
```

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

        q = self.wq(q)  # (ba
        k = self.wk(k)
        v = self.wv(v)

        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)

        scaled_attention, attention_weights =
            q, k, v, mask)
```

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

q.shape
(batch_size, seq_len, d_model)
->
(batch_size, self.num_heads, seq_len, self.depth)

- First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

perm of transpose
tf.transpose(x, perm=[0, 2, 1, 3]) moves
x[a][b][c][d] to x[a][c][b][d]

d_K is dimension of matrix K.

Review : Multi Head Attention

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

        attn_output, _ = self.mha(x, x, x, mask)
```

```python
class MultiHeadAttention(tf.keras.layers.Layer):
    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)
        v = self.wv(v)

        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)

        scaled_attention, attention_weights = scaled_dot_product_attention(
        q, k, v, mask)
```

let's go to scaled dot product attention!

▶ First, get Query, Key, Value matrix from input matrix X

$$Q = X \mathbin{@} W_Q$$
$$K = X \mathbin{@} W_K$$
$$V = X \mathbin{@} W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

q.shape
(batch_size, self.num_heads, seq_len, self.depth)

```
def scaled_dot_product_attention(q, k, v, mask):

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

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

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

▶ First, get Query, Key, Value matrix
...atrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

q.shape
(batch_size, self.num_heads, seq_len, self.depth)

```
def scaled_dot_product_attention(q, k, v, mask):

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

    dk = tf.cast(tf.shape(k)
    scaled_attention_logits

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

matmul_qk.shape
(batch_size, self.num_heads, seq_len, seq_len)

- First, get Query, Key, Value matrix
  atrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{\boxed{Q \cdot K^T}}{\sqrt{d_K}}\right) V$$

matmul_qk

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

q.shape
(batch_size, self.num_heads, seq_len, self.depth)

```
def scaled_dot_product_attention(q, k, v, mask):

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

    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logi

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

dk is a tf.float32 number

- First, get Query, Key, Value matrix
  atrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$Z = softmax\left(\frac{\underbrace{\boxed{Q \cdot K^T}}_{\text{matmul\_qk}}}{\sqrt{\underbrace{\boxed{d_K}}_{\text{dk}}}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```python
def scaled_dot_product_at

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

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

    if mask is not None:
        scaled_attention_
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

matmul_qk.shape
(batch_size, self.num_heads, seq_len, seq_len)

scaled_attention_logits.shape
(batch_size, self.num_heads, seq_len, seq_len)

- First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$Z = softmax\left(\underbrace{\frac{Q \cdot K^T}{\sqrt{d_K}}}_{\text{scaled\_attention\_logits}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```
def scaled_dot_product_attention(q, k, v, mask):

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

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

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

▶ Padding mask matrix consists of 1 and 0
1 means mask

seq is token ID matrix, and its size is (batch_size, seq_len)

```
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)
```

▶ First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\left(\boxed{\frac{Q \cdot K^T}{\sqrt{d_K}}}\right) V$$

scaled_attention_logits

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```
def scaled_dot_product_attention(q, k, v, mask):

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

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

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

▶ Padding mask matrix consists of 1 and 0
1 means mask

```
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, :]
```

seq is token ID matrix, and its size is (batch_size, seq_len)

seq is now mask matrix, and its size is (batch_size, 1, 1, seq_len)

▶ First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

▶ Then, derive attention matrix through the formula behind

$$Z = softmax\underbrace{\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right)}_{\text{scaled\_attention\_logits}} V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```
def scaled_dot_product_attention(q, k, v, mask):

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

    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.
    output = tf.matmul(atte
    return output, attentic
```

scaled_attention_logits.shape
(batch_size, self.num_heads, seq_len, seq_len)

mask.shape
(batch_size, 1, 1, seq_len)

- Padding mask matrix consists of 1 and 0
  1 means mask
- In mask matrix, mask elements are -10^9 and otherwise 0
- Add mask matrix to scaled_attention_logits

- First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$Z = softmax\left(\underbrace{\frac{Q \cdot K^T}{\sqrt{d_K}}}_{\text{scaled\_attention\_logits}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```
def scaled_dot_product_attention(q, k, v, mask):

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

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

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights =
    output = tf.matmul(a
    return output, atten
```

for the result of masking scaled_attention_logits will have -10^9 for the pad token.

- Padding mask matrix consists of 1 and 0
  1 means mask

- In mask matrix, mask elements are -10^9 and otherwise 0

- Add mask matrix to scaled_attention_logits

- First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$Z = softmax\left(\underbrace{\frac{Q \cdot K^T}{\sqrt{d_K}}}_{\text{scaled\_attention\_logits}}\right) V$$

※ Z is attention matrix, d_K is dimension of matrix K.

Review : Multi Head Attention

```python
def scaled_dot_product_attention(q, k, v, mask):

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

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

    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

- First, get Query, Key, Value matrix from input matrix X

$$Q = X @ W_Q$$
$$K = X @ W_K$$
$$V = X @ W_V$$

- Then, derive attention matrix through the formula behind

$$\underbrace{\boxed{Z}}_{\text{output}} = \underbrace{\boxed{softmax\left(\frac{Q \cdot K^T}{\sqrt{d_K}}\right)}}_{\text{attention\_weights}} V$$

※ Z is attention matrix, d_K is dimension of matrix K.