그룹명 : 66기 비전팀
(8)차 주간보고서

활동 현황

작성자	여윤기	장소	온라인
모임일자	2022년 11월 13일 일요일	모임시간	22:00 ~ 23:50 (총110분)
참석자	여윤기, 이근호, 정성실	결석자	없음

학습 내용

학습주제 및 목표

Transformer - Attention is all you need 논문에서 제시한 모델구조를 참고하여 파트별로 코딩을 통해 모델을 직접 구현하며 Transformer 원리를 학습하였음.

학습내용

◇ 트랜스포머 Encoder - Decoder 코드 구현

• 기본적으로 트랜스포머 논문은 인코더 layer와 디코더 layer가 중첩되어 순서대로 연결되어 있는 Seq2Seq구조로 구성

• 인코더에서는 source 문장을 input으로 받아 embedding, 디코더에서는 인코더의 output을 참조하고 디코더에서 input으로 받은 정답 문장과 상응하는 문장을 출력

• Transformer 또한 입력문장과 출력문장 두가지 병렬 구조로 된 데이터셋을 학습하게 됨

• 학습 데이터에 따라 번역모델 외 챗봇 등 학습을 통해 다양한 모델을 생성할 수 있다.

< Positional Embedding >

• 텍스트 문장을 input으로 받기 위해 컴퓨터가 인식할 수 있는 숫자값으로 이뤄진 Embedding Vector로 변환하는 과정을 거쳐야함.

• 문장을 Embedding Vector로 변환시키는 방법은 여러가지가 있음(단어의 출현 빈도, 예측기반방법, 토픽기반방법 등...)

• RNN 모델에서는 모델 구조상 문장의 순서대로 input을 넣어주기 때문에 별도로 문장의 순서에 대한 정보를 모델에 입력시켜 줄 필요가 없었지만, 트랜스포머에서는 문장을 구성하는 모든 단어를 한꺼번에 input으로 받기 때문에 단어의 Embedding Vecotr에 위치정보를 가진 positional embedding을 더해준 값을 모델의 input으로 한다.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# positional embedding 구현
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(짝수에는 sin 함수 적용)
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # apply cos to odd indices in the array; 2i+1(홀수에는 cos함수 적용)
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    pos_encoding = angle_rads[np.newaxis, ...]
    return tf.cast(pos_encoding, dtype=tf.float32)
```

* 지면이 모자랄 경우 별지에 작성하세요

< Masking >

- 특정 값들을 가려서 실제 연산에 방해가 되지 않도록 하는 기법
- Transformer에서는 Attention을 위해 크게 두가지 마스킹 기법을 사용함

1. Padding Masking

- 입력 문장의 길이가 서로 다를때, 모든 문장의 길이를 동일하게 해주기 위해서 정해진 길이보다 짧은 문장의 경우 0값을 패딩으로 채워서 문장의 길이를 맞춰주는 자연어 전처리 기법. 이때, 0은 실제로 의미있는 값은 아니므로 Attention 연산 수행할 때는 마스킹을 통해 제외시켜 주어야함.

- Look Ahead Masking : RNN 모델에서는 hidden state 값을 차례로 갱신하면서 문장 출력 시 다음 단어를 만들어 갈 때 항상 이전 출력 단어를 참고하게 된다. 그러나 Transformer 모델에서는 문장 구성하는 단어 전체가 한번에 입력으로 들어가므로 문장 위치와 상관없이 전체 문장을 참고하여 다음 단어를 예측할 수 있다. 모델 학습 목적이 이전 단어를 바탕으로 다음 출력 단어를 예측하는 것이므로, 출력 단어 이후에 나와야할 단어들을 참고하지 않도록 Masking 하는 기법이 Look-Ahead Masking 이다.(즉, 예를 들면 3번째 단어를 출력하기 위해 문장 순사 상 앞선 1,2번째 단어들만 참고하는 방식)

```
# 패딩 마스크

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_l

# Look-Ahead 마스크
def create_look_ahead_mask(size):
  mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0
  return mask  # (seq_len, seq_len)
```

< Attention >

- Attention 함수는 주어진 쿼리에 대해 모든 키와의 유사도를 각각 구한다.
- 구해진 유사도를 키와 매핑된 각각의 값에 반영한다.
- 유사도가 반영된 값을 모두 더해서 뭉쳐주면 최종 결과인 Attention Value가 도출된다.
- Transformer에 사용된 Attention

1. Encoder self attention(인코더 part) : 인코더에 입력으로 들어간 문장 단어들의 유사도 구함

- 단어들 간 유사도를 구하는 수식 : Scaled dot product attention

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

- 수식에서 Query와 Key 행렬의 내적(dot product) 를 통해 구한 Query와 Key의 유사도를 dk(분산) 값으로 나누어서 scale 해주었기 때문에 Scaled dot product attention 이라고 함.

2. Masked Decoder self attention(디코더 part) : 단어를 1개씩 생성하는 디코더에서 이미 생성된 앞 단어들과의 유사도를 구함(Masking 기법이 적용된 것 뿐이지, Encoder self attention과 작동방식이 동일함)

3. Encoder-Decoder attention(디코더 part) : 디코더가 잘 예측하기 위해서 인코더에 입력된 단어들과의 유사도를 구함

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

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

      Returns:
        output, attention_weights
      """

      matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)
      # scale matmul_qk
      dk = tf.cast(tf.shape(k)[-1], tf.float32)
      scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

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

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

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

      return output, attention_weights

    def print_out(q, k, v):
      temp_out, temp_attn = scaled_dot_product_attention(
          q, k, v, None)
      print('Attention weights are:')
      print(temp_attn)
      print('Output is:')
      print(temp_out)
```

< Multi-Head Attention >

• 트랜스포머에서 사용되는 위 3개의 Attention은 모두 Multi Head Attention에 속한다.

• Multi-Head Attention 간단히 이야기하면, 위 Attention 계산을 입력문장이 들어왔을 때 한번만 수행하는 것이 아닌 동시에 Attention 연산을 여러번(n번) 수행하는 Attention 방식임.

-> Attention을 한번만 수행할 때에는 나타나지 않았던 다른 단어들과의 유사도가 Multi-Head Attention 에서는 나타날 수 있다.

• num_heads = 병렬적으로 몇개의 attention 연산을 수행할 것인가? Attention 수행 횟수 나타냄

• 입력 문장의 길이를 행으로 하고, embedding vector 차원을 열으로 하는 행렬을 num_heads 수만큼 쪼개서 Attention 연산을 수행. 이후 num_heads 수만큼 얻은 attention 값 행렬을 다시 하나로 합친다.

```
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])

  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)
```

< Feed Forward Network >

• 2개의 완전히 연결된 Dense Layer로 구성. 활성화 함수는 Relu 사용

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

< 인코더 layer 구성 >

• 인코더 layer는 아래의 하위 계층으로 구성된다.

1. Multi-Head Attention(Encoder self attention)
2. Feed Forward Network

• 각 하위 계층은 학습 효율을 높여주고 gradient vanishing 문제를 해결하기 위해 residual connection으로 연결되어 있다.

• 이후 residual connection 값 + layer 출력을 더한 값을 정규화(Normalization)

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

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

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)

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

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

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

        return out2
```

< 인코더 구성 >

• 인코더는 아래와 같이 구성된다.

1. 입력 문장에 대한 Embedding
2. Positional Embedding
3. 1번 + 2번을 입력값으로 받아, N개의 인코더 layer 적층
   * 이때, 각각의 인코더 layer는 서로 다른 파라미터를 가짐

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

        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
        self.pos_encoding = positional_encoding(maximum_position_encoding,
                                                self.d_model)

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

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

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

        seq_len = tf.shape(x)[1]

        # adding embedding and position encoding.
        x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]

        x = self.dropout(x, training=training)

        # for문을 활용하여 EncoderLayer를 정해준 num_layer만큼 쌓아준다.
        for i in range(self.num_layers):
          x = self.enc_layers[i](x, training, mask)

        return x  # (batch_size, input_seq_len, d_model)
```

<.디코더 layer 구성 >

- 디코더 layer는 아래 하위 계층으로 구성된다.
  1. Maksed Multi Head Attention(Self attention)
  2. Multi-Head Attention(Encoder-Decoder attention)
  3. Feed Forward Network
- 이후 1,2,3번 layer를 거친 출력값 + residual connection 더한 후 Normalization
- 즉, 디코더에서는 정답 문장의 embedding 벡터를 self-attention 하여 단어들 간 유사도를 구하고, 인코더의 출력인 Query, Key 값을 Encoder-Decoder Attention으로 참고하여 생성해야 할 예측 단어를 만들어낸다.

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

         self.mha1 = MultiHeadAttention(d_model, num_heads)
         self.mha2 = MultiHeadAttention(d_model, num_heads)

         self.ffn = point_wise_feed_forward_network(d_model, dff)

         self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
         self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
         self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

         self.dropout1 = tf.keras.layers.Dropout(rate)
         self.dropout2 = tf.keras.layers.Dropout(rate)
         self.dropout3 = tf.keras.layers.Dropout(rate)

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

         # 첫번째 Attention : Look Ahead Mask 적용된 Self-Attention
         attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_size, target_seq_len, d_model)
         attn1 = self.dropout1(attn1, training=training)
         out1 = self.layernorm1(attn1 + x)

         # 두번째 Attention : 인코더 출력 참고하여 Encoder-Decoder Attention 수행
         attn2, attn_weights_block2 = self.mha2(
             enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len, d_model)
         attn2 = self.dropout2(attn2, training=training)
         out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)

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

         return out3, attn_weights_block1, attn_weights_block2
```

<.디코더 구성 >

- 디코더는 아래와 같이 구성된다.
  1. output 문장에 대한 Embedding
  2. Positional Embedding
  3. N개의 디코더 layer 적층

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

         self.d_model = d_model
         self.num_layers = num_layers

         self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
         self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

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

       def call(self, x, enc_output, training,
                look_ahead_mask, padding_mask):

         seq_len = tf.shape(x)[1]
         attention_weights = {}

         x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
         x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
         x += self.pos_encoding[:, :seq_len, :]

         x = self.dropout(x, training=training)

         for i in range(self.num_layers):
           x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                                  look_ahead_mask, padding_mask)

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

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

◇ 트랜스포머 Encoder - Decoder 파트를 코드를 통해 구현하였으며, 이를 조합하여 Transformer 모델을 생성할 수 있다. 이후 학습에서는 생성한 Transformer 모델을 특정 데이터셋을 학습시켜 어떤 output을 출력하는지 확인해 볼 예정이다.

그룹 운영 기록사항

-

다음 모임 계획

모임일자	2022년 11월 17일 목요일	모임시간	21:00~23:00 (총 120 분)
역할분담	데이터셋을 활용하여 Transformer 모델학습 및 결과 비교	장 소	오프라인