exam 개인학습

Lec 3.2

AI summary ✨ Try on this page

Type something...

01) Image Captioning with Transformers
01_Multi-Headed Attention
02_MultiHeadAttention(nn.Module):
03_Positional Encoding
1. Inline Question 1
04_CaptioningTransformer

01) Image Captioning with Transformers

01_Multi-Headed Attention

- **Scaled dot-Product Attention**

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

- **Self-Attention**

$$v_i = Vx_i \; i \in \{1,...,l\}$$
$$k_i = Kx_i \; i \in \{1,...,l\}$$
$$q_i = Qx_i \; i \in \{1,...,l\}$$
$$X \in \mathbb{R}^{l \times d}$$
$$V,K,Q \in \mathbb{R}^{d \times d}$$

- Multi-Headed Scaled Dot-Product Attention

$$Y_i = softmax(\frac{(XQ_i)(XK_i)^T}{\sqrt{d/h}})(XV_i)$$
$$Q_i, K_i, V_i \in \mathbb{R}^{d \times d/h}$$
$$Y_i \in \mathbb{R}^{l \times d/h}$$
$$XQ_i \in \mathbb{R}^{l \times d/h}$$

→ h: number of heads, attention을 여러개로 쪼갬.

$$Y_i = dropout(softmax(\frac{(XQ_i)(XK_i)^T}{\sqrt{d/h}}))(XV_i)$$

→ Dropout까지 사용

$$Y = [Y_1; ...Y_h]A$$

→ Concatenation 이후 linear transformation 사용

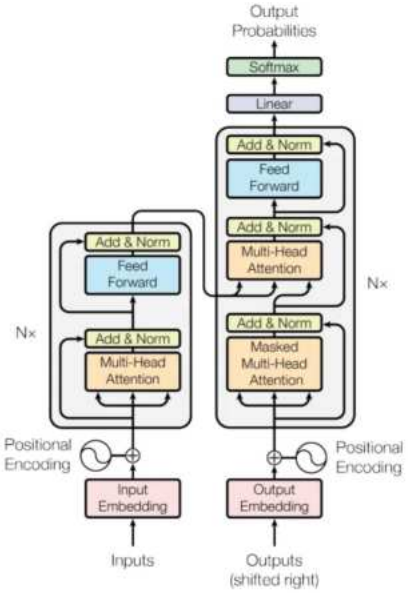


Figure 1: The Transformer - model architecture.

<요약>

- Multi-Headed Attention
  1. Multi-Headed Scaled Dot-Product Attention(Dropout 포함, ratio = 0.1)
  2. Concatenation
  3. Linear transformation
- Residual connection + Layer Normalization
- Feed Foward Network(FFN)
  1. Linear
  2. ReLU
  3. Linear

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b2$$

- Residual connection + Layer Normalization

02_MultiHeadAttention(nn.Module):

- `masked_fill` : tensor의 특정 값을 다른 값으로 바꾸는데 사용된다.

```
# Q, K, V Split first
H = self.n_head
query = torch.reshape(self.query(query), (N, S, H, E// H))
query = torch.swapaxes(query, 1, 2) # N, H, S, E // H
key = torch.reshape(self.key(key), (N, T, H, E// H))
key = torch.swapaxes(key, 1, 2) # N, H, T, E // H
value = torch.reshape(self.value(value), (N, T, H, E// H))
value = torch.swapaxes(value, 1, 2) # N, H, T, E // H

# matmul 진행
xqxk = torch.matmul(query, torch.swapaxes(key, 2, 3)) # 전치행렬
xqxk /= torch.sqrt(torch.tensor(self.head_dim))

# mask
if attn_mask != None:
  xqxk = xqxk.masked_fill(attn_mask == 0, float('-inf'))

# softmax
out = self.attn_drop(F.softmax(xqxk, dim = -1))
out = torch.matmul(out, value)

# reshape
out = torch.swapaxes(out, 1, 2)
out = torch.reshape(out, (N, S, E))
output = self.proj(out)
```

- (N, S, E) → (N, S, H, E/H) , (N, T, E) → (N, T, H, E/H) 로 shape를 바꿔주기
- Key 값을 transpose해주고 matmul 진행
- mask 가 있다면 해당 인덱스 값을 `-inf` 으로 바꿔주기
- softmax → dropout 진행, softmax는 마지막 차원을 기준으로 진행
- 마지막에 out 모양을 (N, S, E)원래 모양으로 바꿔주기 → 모양이 시작과 끝이 같아야 함.
- proj(Linear)을 통과시키기.

03_Positional Encoding

$$P \in \mathbb{R}^{l \times d} \; where$$
$$P_{ij} = sin(i \cdot 10000^{\frac{j}{d}}) \; (if \; j \; is \; even)$$
$$p_{ij} = cos(i \cdot 10000^{\frac{-(j-1)}{d}}) \; (otherwise)$$

- 인덱스가 짝수일 때는 sin함수에, 인덱스가 홀수일 때는 cos함수에 넣어서 Positional encoding을 해준다.
- Input X 대신에 X + P를 통과시킬 것이다.
- **Positional Encoding 후 Dropout을 진행한다.**

```
# max_len = L, embed_dim = d
row = torch.arange(0, max_len).reshape(-1, 1) # 5000 x 1
col = torch.pow(10000, -torch.arange(0, embed_dim, 2) / embed_dim) # 3
pe[0, :, 0::2] = torch.sin(row * col)
pe[0, :, 1::2] = torch.cos(row * col)
```

- row는 연산을 위해 `reshape(-1, 1)` 를 해주었다.
- 짝수는 sin, 홀수는 cos을 씌워준다.
- col는 절반씩 사용될 거기 때문에 사이즈가 절반이다. (0, 0, 2, 2, 4, 4...)

```
# x shape: (1, 2, 6)
# self.pe shape: (1, 5000, 6)
# pe의 max_length = 5000
P = self.pe[:, :S, :] # (1, 2, 6)
output = self.dropout(x + P)
```

- `self.pe` 는 max_length까지 계산한거기 때문에 그 중에서 S만큼만 뽑아서 더해준다.
- 그 후 잊지말고 Dropout을 해주자.

1. Inline Question 1

Inline Question 1

Several key design decisions were made in designing the scaled dot product attention we introduced above. Explain why the following choices were beneficial:

1. Using multiple attention heads as opposed to one.
2. Dividing by $\sqrt{d/h}$ before applying the softmax function. Recall that d is the feature dimension and h is the number of heads.
3. Adding a linear transformation to the output of the attention operation.

Only one or two sentences per choice is necessary, but be sure to be specific in addressing what would have happened without each given implementation detail, why such a situation would be suboptimal, and how the proposed implementation improves the situation.

Your Answer:

1. 하나의 attention 대신 여러개의 attention을 사용하면 sequence의 다양한 부분에 focus를 할 수 있게 되어 flexbility해진다.
2. 차원이 너무 커지면 dot product값이 매우 커지기 때문에 softmax함수도 큰 숫자를 반영해 gradient가 커질 수 있기 때문에 scaling을 해주어야 한다.
3. Attention operation이후 linear을 넣게 되면 concate 부분을 잘 mixing해줄 수 있다.

04_CaptioningTransformer

```
    - word_to_idx: A dictionary giving the vocabulary. It contains V entries,
      and maps each string to a unique integer in the range [0, V).
    - input_dim: Dimension D of input image feature vectors.
    - wordvec_dim: Dimension W of word vectors.
    - num_heads: Number of attention heads.
    - num_layers: Number of transformer layers.
    - max_length: Max possible sequence length.
    """
    super().__init__()

    vocab_size = len(word_to_idx)
    self.vocab_size = vocab_size
    self._null = word_to_idx["<NULL>"]
    self._start = word_to_idx.get("<START>", None)
    self._end = word_to_idx.get("<END>", None)

    self.visual_projection = nn.Linear(input_dim, wordvec_dim)
    self.embedding = nn.Embedding(vocab_size, wordvec_dim, padding_idx=self._null)
    self.positional_encoding = PositionalEncoding(wordvec_dim, max_len=max_length)

    decoder_layer = TransformerDecoderLayer(input_dim=wordvec_dim, num_heads=num_heads)
    self.transformer = TransformerDecoder(decoder_layer, num_layers=num_layers)
    self.apply(self._init_weights)

    self.output = nn.Linear(wordvec_dim, vocab_size)
```

- word_to_idx: 각 숫자가 뜻하는 word 저장

- input_dim: feature의 차원
- wordvec_dim: wordvector의 차원
- num_heads: head의 개수
- num_layer: transformer layer의 개수
- max_length: output caption의 최대 길이
- visual_projection: feature → wordvec
- embedding: caption → wordvec
- positional_encoding: 위치정보 추가

```
class TransformerDecoderLayer(nn.Module):
    """
    A single layer of a Transformer decoder, to be used with TransformerDecoder.
    """
    def __init__(self, input_dim, num_heads, dim_feedforward=2048, dropout=0.1):
        """
        Construct a TransformerDecoderLayer instance.

        Inputs:
        - input_dim: Number of expected features in the input.
        - num_heads: Number of attention heads
        - dim_feedforward: Dimension of the feedforward network model.
        - dropout: The dropout value.
        """
        super().__init__()
        self.self_attn = MultiHeadAttention(input_dim, num_heads, dropout)
        self.multihead_attn = MultiHeadAttention(input_dim, num_heads, dropout)
        self.linear1 = nn.Linear(input_dim, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, input_dim)

        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.norm3 = nn.LayerNorm(input_dim)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

        self.activation = nn.ReLU()
```

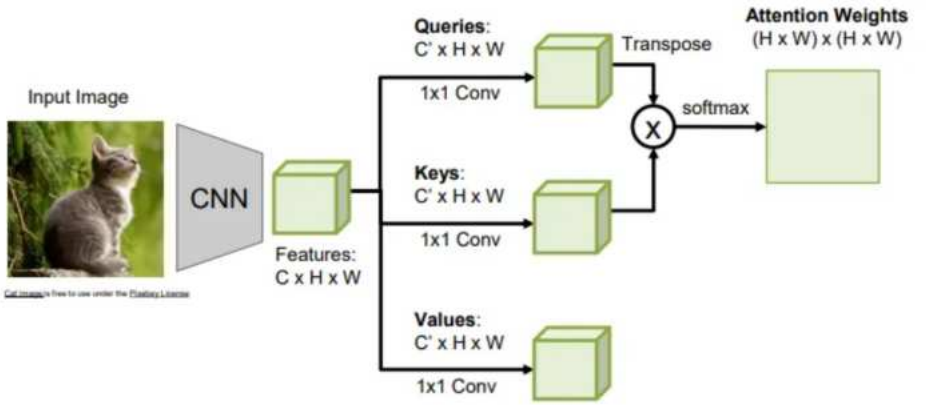
- multiHeadAttention사용
- linear + dropout 사용

```
def forward(self, tgt, memory, tgt_mask=None):
    """
    Pass the inputs (and mask) through the decoder layer.

    Inputs:
    - tgt: the sequence to the decoder layer, of shape (N, T, W)
    - memory: the sequence from the last layer of the encoder, of shape (N, S, D)
    - tgt_mask: the parts of the target sequence to mask, of shape (T, T)

    Returns:
    - out: the Transformer features, of shape (N, T, W)
    """
    # Perform self-attention on the target sequence (along with dropout and
    # layer norm).
    tgt2 = self.self_attn(query=tgt, key=tgt, value=tgt, attn_mask=tgt_mask)
    tgt = tgt + self.dropout1(tgt2)
    tgt = self.norm1(tgt)

    # Attend to both the target sequence and the sequence from the last
    # encoder layer.
    tgt2 = self.multihead_attn(query=tgt, key=memory, value=memory)
    tgt = tgt + self.dropout2(tgt2)
    tgt = self.norm2(tgt)

    # Pass
    tgt2 = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
    tgt = tgt + self.dropout3(tgt2)
    tgt = self.norm3(tgt)
    return tgt
```

1. multihead self attention → dropout → residual connection → layer norm
2. memory(encoder sequence)를 key, value로 받고 다시 multihead self attention → dropout → residual connection → layer norm
3. Feed Forward Network(linear → relu → dropout → linear) → dropout + residual connection → layernorm

- 우리가 구현할 것은 Captioning Transformer의 forward 함수이다.

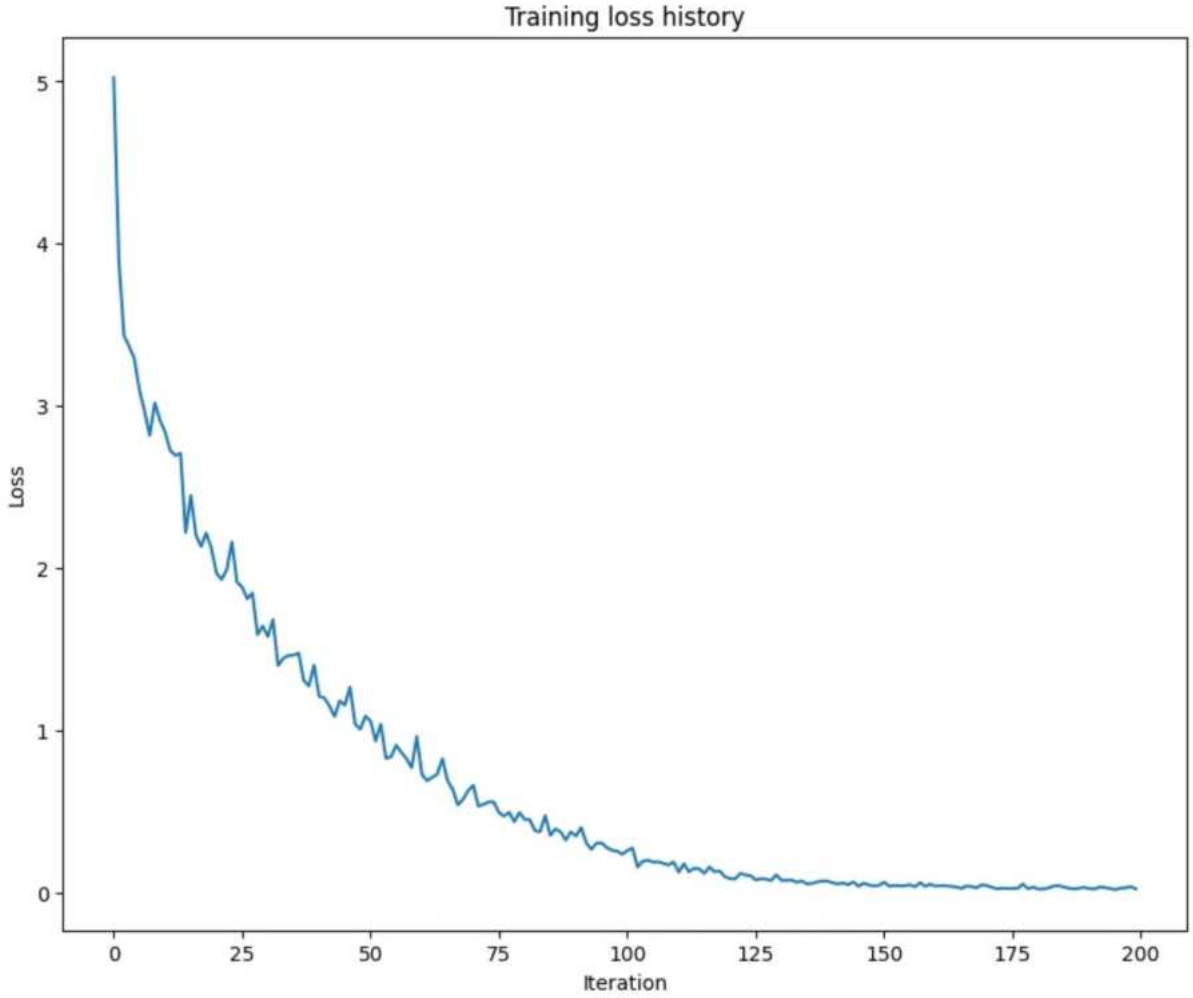
Example: CNN with Self-Attention



- Image Feature들 visual project해서 Query, Key, Value 값을 만든다.

```
caption_embed = self.embedding(captions) # (N, T, D)
caption_embed = self.positional_encoding(caption_embed) # (N, T, D)
feature_proj = self.visual_projection(features).view(N, 1, -1) # (N, 1, D)

tgt_mask = torch.tril(torch.ones(T, T)) # 하삼각행렬
out = self.transformer(caption_embed, feature_proj, tgt_mask)
scores = self.output(out)
```

- 이미지를 처리하는 encoder sequence는 visual_projectiong layer를 통해 구현해준다.
- decoder에 사용할 것이기 때문에 차원을 맞춰준다.
- tgt_mask는 아래 삼각형은 0, 위 삼각형은 1로 절반 정도 mask를 해준다.
- encoder는 사진을 보기 때문에 features를 이용하고 Decoder는 글자를 출력할 것이기 때문에 Caption을 만든다.
- out: (N, T, D), self.out: (D, V) → scores : (N, T, V) 완성



- 모델이 잘 overfit되는 것을 확인할 수 있다.

train

a <UNK> decorated living room with a big tv in it <END>
GT:<START> a <UNK> decorated living room with a big tv in it <END>

val

a man is <UNK> in a bottles his food while <END>
GT:<START> a group of people <UNK> outside by a wall <END>

- validation set에서도 RNN보다 뛰어난 성능을 보여준다.