Assignment 2. Attention is All You Need

120220225 조유빈

1. 논문 내용 요약

   A. 제안하는 내용

      i. Recurrence와 convolution을 완전히 배제하고 attention mechanism만을 기반으로 새로운 architecture인 Transformer를 제안

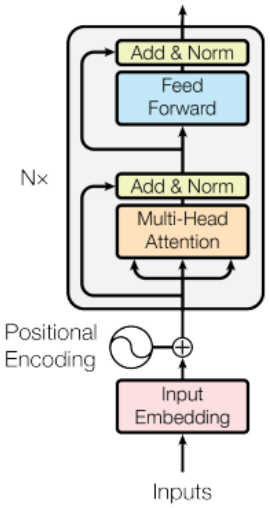
         ✓ 병렬적으로 동시 연산 가능

         ✓ 멀리 떨어진 원소들 간의 path length 감소

            ▪ Long-term dependency problem 해결

2. 수식 설명 및 코드 분석

   A. Transformer Encoder



      i. 먼저, 입력 문장의 각 단어들을 embedding하여 token화 함

```
self.tok_embedding = nn.Embedding(input_dim, hid_dim)
```

      ii. Token embedding은 positional embedding과 summation되어 모든 단어의 위치 정보를 부여

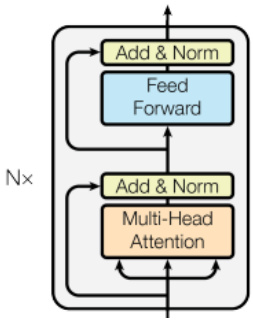
```
src = self.dropout((self.tok_embedding(src) * self.scale) + self.pos_embedding(pos))
```

✓ 같은 단어여도 문장 내 위치에 따라 해석되는 의미가 다를 수 있음

iii. 임베딩 된 토큰들은 encoder layers를 통과함

```
for layer in self.layers:
    src = layer(src, src_mask) # [batch size, src len, hid dim]

return src
```

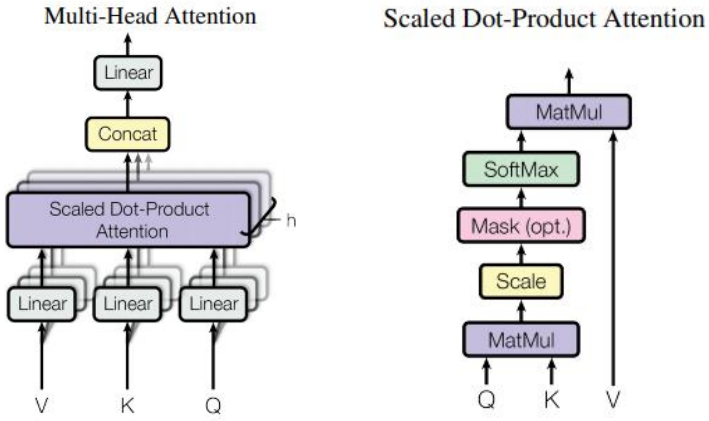
B. Encoder Layer



i. Multi-Head Attention module과 Feed Forward module로 구성되며 residual connection을 수행함

```
# Multi-Head Self Attention 수행
# 하나의 sentence 내의 서로 다른 token들을 관련시켜 각 token들의 representation을 구함
_src, _ = self.self_attention(src, src, src, src_mask)

# dropout, residual connection and layer norm
src = self.self_attn_layer_norm(src + self.dropout(_src)) # [batch size, src len, hid dim]

# positionwise feedforward - 앞서 추출한 features를 refine하는 역할
_src = self.positionwise_feedforward(src)

# dropout, residual and layer norm
src = self.ff_layer_norm(src + self.dropout(_src)) # [batch size, src len, hid dim]

return src
```

ii. Multi-Head Attention module에서 self-attention을 수행하여 한 sequence 내의 서로 다른 token들을 관련시켜 각 token들의 representation을 구함



✓ Self-attention : Query, Key, Value의 출처가 같은 (모두 encoder vector)

✓ Multi-Head : 벡터의 차원을 축소시키고 attention을 병렬적으로 수행

- 다른 관점에서 정보들을 수집 가능
- 각 attention head마다 W^Q, W^K, W^V값이 다름

✓ $Head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

```
# query vector, key vector, value vector 생성
Q = self.fc_q(query) # [batch size, query len, hid dim]
K = self.fc_k(key)   # [batch size, key len, hid dim]
V = self.fc_v(value) # [batch size, value len, hid dim]

# Head 수 만큼 dim을 쪼개어 각 head에 해당하는 query, key, value 값 생성
Q = Q.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3) # [batch size, n heads, query len, head dim]
K = K.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3) # [batch size, n heads, key len, head dim]
V = V.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3) # [batch size, n heads, value len, head dim]
```

- Linear layer를 통해 인풋 벡터로부터 query, key, value 생성
- Head 만큼 dimension을 쪼개어 각 head에 해당되는 query, key, value값 생성

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

- 행렬 연산을 통해 query와 key간의 유사도를 계산. 관련 있는 토큰 간의 유사도는 큰 값을 가지게 됨. 값이 너무 커지는 것을 방지하기 위해 $\sqrt{d_k}$ 로 나누어 normalize함 (d_k : key의 $dimension$ 크기)

```
energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale  # [batch size, n heads, query len, key len]
```

- Softmax를 통해 normalize하여 attention score matrix를 얻음

```
attention = torch.softmax(energy, dim = -1)   # [batch size, n heads, query len, key len]
```

- Attention score matrix와 value를 행렬 연산하여 attention matrix를 얻음

```
x = torch.matmul(self.dropout(attention), V)  # [batch size, n heads, query len, head dim]
```

✓ $MultiHead(Q,K,V) = Concat(Head_1,\cdots,Head_h)W^O$

- 각 head를 concatenate하여 원래 dimension으로 만들고 weight를 곱하여 최종 output을 얻음

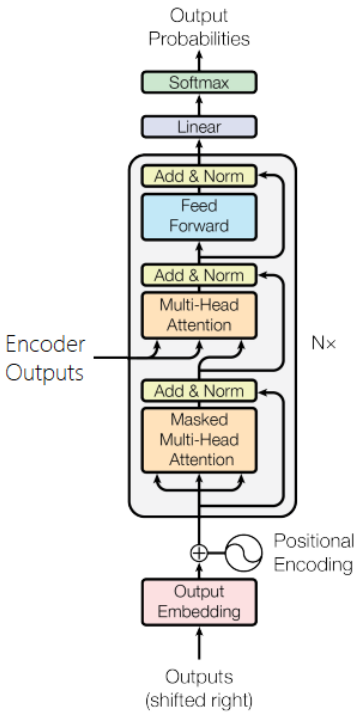
```
# 각 head를 concat하여 원래 dimension으로 만듦
x = x.view(batch_size, -1, self.hid_dim)  # [batch size, query len, hid dim]
# W_O를 곱하여 최종 output을 얻음
x = self.fc_o(x)  # [batch size, query len, hid dim]
```

C. Position-wise Feed Forward

i. 앞서 추출한 feature를 refine하는 역할을 해 줌

```
x = self.dropout(torch.relu(self.fc_1(x)))  # [batch size, seq len, pf dim]
x = self.fc_2(x)    # [batch size, seq len, hid dim]
```

D. Transformer Decoder



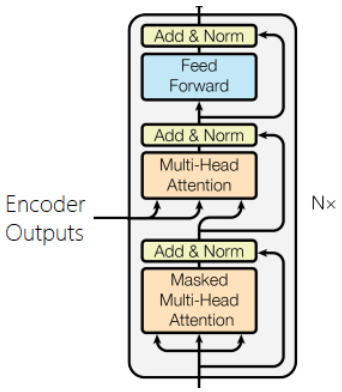
i. Decoder는 encoder의 정보를 받아 target sentence를 predict함

ii. 입력 값으로 이전 decoder layer에서 예측한 출력 값이 들어 감

iii. Encoder 과정과 동일하게 token embedding과 positional encoding과정을 거쳐 decoder layer로 들어 감

```
# token embedding & positional encoding
trg = self.dropout((self.tok_embedding(trg) * self.scale) + self.pos_embedding(pos)) # [batch size, trg len, hid dim]

# decoder layers
for layer in self.layers:
    trg, attention = layer(trg, enc_src, trg_mask, src_mask) # [batch size, trg len, hid dim], [batch size, n heads, trg len, src len]

output = self.fc_out(trg) # [batch size, trg len, output dim]
```

E. Decoder layer



i. Encoder layer와 달리 self-attention 과정과 feed forward과정 사이에 cross-attention (non self-attention) 과정이 추가됨

ii. Masked Multi-Head Self-Attention

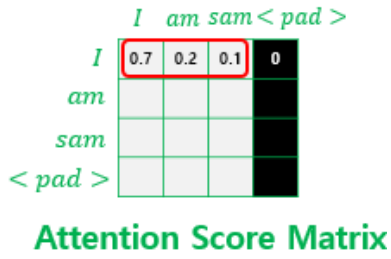
```
# Masked Multi-Head Self Attention : encoder의 self-attention과 달리 일부 원소는 masking
_trg, _ = self.self_attention(trg, trg, trg, trg_mask)

# dropout, residual connection and layer norm
trg = self.self_attn_layer_norm(trg + self.dropout(_trg)) # [batch size, trg len, hid dim]
```

✓ Query, key, value의 출처가 모두 decoder vector로 같으므로 self-attention이라 할 수 있음

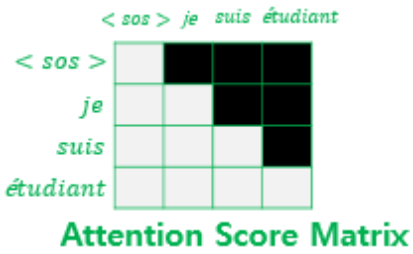
✓ Encoder의 self-attention 과정과 동일하지만 아래 조건에 해당하는 일부 원소 값은 매우 작은 음수 값을 곱해 masking해준다는 점이 다름

```
def make_trg_mask(self, trg):
    # Make attention masks
    # trg : [batch size, trg len]

    # <pad>와의 연산 값은 masking을 통해 가림
    trg_pad_mask = (trg != self.trg_pad_idx).unsqueeze(1).unsqueeze(2) # [batch size, 1, 1, trg len]

    trg_len = trg.shape[1]

    # 예측 단계 뒤에 나올 token과의 연산 값은 masking
    trg_sub_mask = torch.tril(torch.ones((trg_len, trg_len), device = self.device)).bool() # [trg len, trg len]

    # 두 mask를 합침
    trg_mask = trg_pad_mask & trg_sub_mask # [batch size, 1, trg len, trg len]

    return trg_mask
```

- 실질적인 의미를 가진 단어가 아닌 <pad>인 경우



- 현재 시점보다 미래에 있는 단어인 경우



iii. Multi-Head Cross Attention

```
# Multi-Head Cross Attention : Query는 decoder vector / Key-Value는 encoder vector
# Encoder의 정보를 참조하는 과정
_trg, attention = self.encoder_attention(trg, enc_src, enc_src, src_mask)

# dropout, residual connection and layer norm
trg = self.enc_attn_layer_norm(trg + self.dropout(_trg)) # [batch size, trg len, hid dim]
```

✓ Query는 decoder vector, Key-Value는 encoder vector이므로 self-attention이라 할 수 없음

✓ 연산 과정은 multi-head self-attention 과정과 동일

✓ Decoder의 출력을 위해 encoder의 어떤 정보를 참고하면 좋을지 attention을 수행함

F. 최종 Transformer

```
def forward(self, src, trg):
    # src : [batch size, src len]
    # trg : [batch size, trg len]

    # encoder의 attention mask와 decoder의 attention mask 생성
    src_mask = self.make_src_mask(src)  # [batch size, 1, 1, src len]
    trg_mask = self.make_trg_mask(trg)  # [batch size, 1, trg len, trg len]

    # Encoder
    enc_src = self.encoder(src, src_mask) # [batch size, src len, hid dim]
    # Decoder
    output, attention = self.decoder(trg, enc_src, trg_mask, src_mask)  # [batch size, trg len, output dim], [batch size, n heads, trg len, src len]

    return output, attention
```

i. Encoder 과정을 먼저 거친 후 encoder에서 추출한 정보를 모든 decoder layers에 넘겨줌

G. 모델 training

```
optimizer.zero_grad()

output, _ = model(src, trg[:,:-1]) # batch size, trg len - 1, output dim]

output_dim = output.shape[-1]

output = output.contiguous().view(-1, output_dim) # [batch size * trg len - 1, output dim]
trg = trg[:,1:].contiguous().view(-1) # [batch size * trg len - 1]

# loss 계산
loss = criterion(output, trg)
# back propagation
loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

# update parameters
optimizer.step()
```

i. Cross-entropy loss 사용

- ✓ N : batch size, C : class 개수
- ✓ Target field의 총 vocab 수를 class개수로 하여 classification하는 것과 같음
- ✓ Target sentence의 첫번째 토큰인 <sos> 토큰은 제외하고 처리하여 (output token 수 -1)만큼의 크기를 가짐

```
Epoch: 01 | Time: 0m 18s
        Train Loss: 4.232 | Train PPL:  68.831
         Val. Loss: 3.040 |  Val. PPL:  20.904
Epoch: 02 | Time: 0m 18s
        Train Loss: 2.819 | Train PPL:  16.755
         Val. Loss: 2.302 |  Val. PPL:   9.995
Epoch: 03 | Time: 0m 18s
        Train Loss: 2.237 | Train PPL:   9.368
         Val. Loss: 1.976 |  Val. PPL:   7.215
Epoch: 04 | Time: 0m 19s
        Train Loss: 1.885 | Train PPL:   6.584
         Val. Loss: 1.804 |  Val. PPL:   6.076
Epoch: 05 | Time: 0m 18s
        Train Loss: 1.637 | Train PPL:   5.141
         Val. Loss: 1.716 |  Val. PPL:   5.561
Epoch: 06 | Time: 0m 18s
        Train Loss: 1.450 | Train PPL:   4.264
         Val. Loss: 1.651 |  Val. PPL:   5.214
Epoch: 07 | Time: 0m 18s
        Train Loss: 1.298 | Train PPL:   3.661
         Val. Loss: 1.625 |  Val. PPL:   5.081
Epoch: 08 | Time: 0m 18s
        Train Loss: 1.170 | Train PPL:   3.224
         Val. Loss: 1.618 |  Val. PPL:   5.041
Epoch: 09 | Time: 0m 18s
        Train Loss: 1.061 | Train PPL:   2.890
         Val. Loss: 1.631 |  Val. PPL:   5.109
Epoch: 10 | Time: 0m 18s
        Train Loss: 0.965 | Train PPL:   2.625
         Val. Loss: 1.637 |  Val. PPL:   5.141
```

H. Inference

i. 데이터 샘플 중 하나를 사용하여 translation 진행

✓ Source sentence

```
src = ['eine', 'frau', 'mit', 'einer', 'großen', 'geldbörse', 'geht', 'an', 'einem', 'tor', 'vorbei', '.']
```

✓ Target sentence

```
trg = ['a', 'woman', 'with', 'a', 'large', 'purse', 'is', 'walking', 'by', 'a', 'gate', '.']
```

✓ Translation

▪ Code 분석

```python
def translate_sentence(sentence, src_field, trg_field, model, device, max_len = 50):
    # model evaluation mode
    model.eval()
    # source sentence tokenize
    if isinstance(sentence, str):
        nlp = spacy.load('de_core_news_sm')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]
    tokens = [src_field.init_token] + tokens + [src_field.eos_token]
    src_indexes = [src_field.vocab.stoi[token] for token in tokens]
    src_tensor = torch.LongTensor(src_indexes).unsqueeze(0).to(device)
    # enocder의 attention mask 생성
    src_mask = model.make_src_mask(src_tensor)
    # Encoder 실행
    with torch.no_grad():
        enc_src = model.encoder(src_tensor, src_mask)
    # decoder의 첫 입력값으로 <sos> 토큰 사용
    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    for i in range(max_len):
        trg_tensor = torch.LongTensor(trg_indexes).unsqueeze(0).to(device)
        # Decoder의 attention mask 생성
        trg_mask = model.make_trg_mask(trg_tensor)
        # Decoder 실행
        with torch.no_grad():
            output, attention = model.decoder(trg_tensor, enc_src, trg_mask, src_mask)
        # Prediction
        pred_token = output.argmax(2)[:,-1].item()

        trg_indexes.append(pred_token)
        # translation 종료
        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break
    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]

    return trg_tokens[1:], attention
```

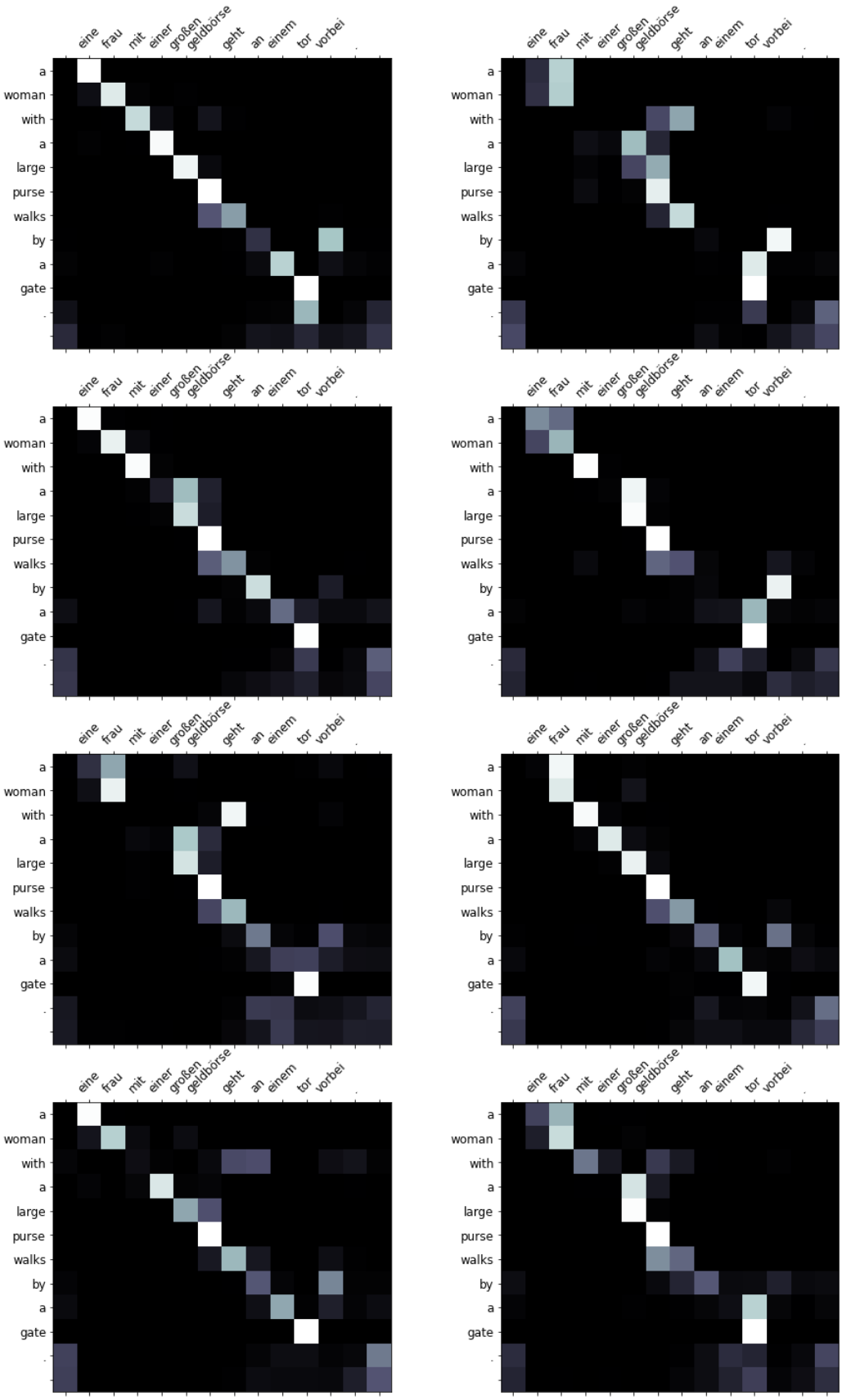
- Prediction

```python
translation, attention = translate_sentence(src, SRC, TRG, model, device)
print(f'predicted trg = {translation}')
```

```
predicted trg = ['a', 'woman', 'with', 'a', 'large', 'purse', 'walks', 'by', 'a', 'gate', '.', '']
```

I. Attention Matrix 시각화

i. 각 head에서 어느 부분에 attention을 두고 있는지 확인


eine frau mit einer großen geldbörse geht an einem tor vorbei .
a woman with a large purse walks by a gate .

J. BLEU score 측정

```
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = translate_sentence(src, src_field, trg_field, model, device, max_len)

        #cut off  token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)
```

```
bleu_score = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score*100:.2f}')
```

```
BLEU score = 35.73
```