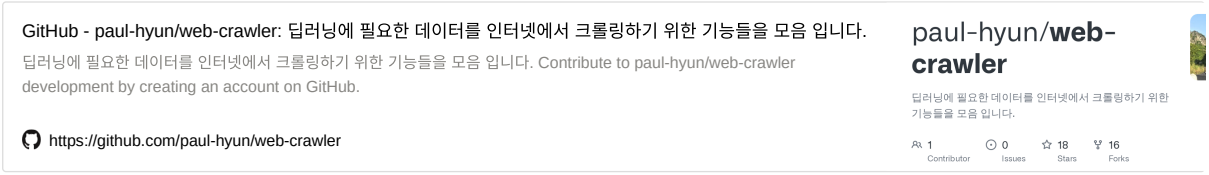
Transformer

1. Google SentencePiece를 활용해 Vocab 만들기

1.1 말뭉치 만들기(한국어 위키)

- 한국어 위키 말뭉치 사용
- pages-articles.xml.bz2 다운로드 한 후, wikiextractor를 이용해 처리된 결과 파일을 텍스트로 변환




```
$ git clone https://github.com/paul-hyun/web-crawler.git
$ cd web-crawler
$ pip install tqdm
$ pip install pandas
$ pip install bs4
$ pip install wget
$ pip install pymongo
$ python kowiki.py
```

1.2 Google SentencePiece 설치하기

```
$ pip install sentencepiece
```

1.3 Vocab 만들기

- vocab_size는 vocab의 개수로 기본 8,000개에 스페셜 토큰 7개를 더해서 8,007개 사용

- vocab_size가 커지면 성능이 좋아지고, 모델 파라미터 수 증가
- 코드 실행 → **kowiki.model, kowiki.vocab 파일 생성**

```
import sentencepiece as spm

corpus = "kowiki.txt"
prefix = "kowiki"
vocab_size = 8000
spm.SentencePieceTrainer.train(
    f"--input={corpus} --model_prefix={prefix} --vocab_size={vocab_size + 7}" +
    " --model_type=bpe" +
    " --max_sentence_length=999999" + # 문장 최대 길이
    " --pad_id=0 --pad_piece=[PAD]" + # pad (0)
    " --unk_id=1 --unk_piece=[UNK]" + # unknown (1)
    " --bos_id=2 --bos_piece=[BOS]" + # begin of sequence (2)
    " --eos_id=3 --eos_piece=[EOS]" + # end of sequence (3)
    " --user_defined_symbols=[SEP],[CLS],[MASK]") # 사용자 정의 토큰
```

2. Naver 쇼핑 감정분석 데이터 전처리하기

2.1 Vocab

- Sentencepiece를 활용해 Vocab 만들기를 통해 만들어 놓은 vocab 로드

```
# vocab loading
vocab_file = "/content/kowiki.model"
vocab = spm.SentencePieceProcessor()
vocab.load(vocab_file)
```

2.2 데이터 전처리

- 다운로드된 데이터를 vocab으로 미리 tokenize해서 json형태로 저장
- 코드 실행 → **ratings_train.json, ratings_test.json 전처리된 파일 생성**

```
import json

""" train data 준비 """
def prepare_train(vocab, infile, outfile):
    df = pd.read_csv(infile, sep="\t", engine="python")
    with open(outfile, "w") as f:
        for index, row in df.iterrows():
            document = row["reviews"]
            if type(document) != str:
                continue
            instance = { "ratings": row["ratings"], "reviews": vocab.encode_as_pieces(document), "label": row["label"] }
            f.write(json.dumps(instance))
            f.write("\n")
```

3. Transformer (Attention Is All You Need) 구현하기

3.1 Embedding

- ‘Input Embedding’과 ‘Position Embedding’ 두가지를 합하여 Transformer의 Embedding 사용

3.1.1 Input Embedding

- input(2,8)에 대한 embedding 값인 input_embs은 (2,8,128) shape를 갖는다.
    a. input에 대한 embedding 값 input_embs 구한다.

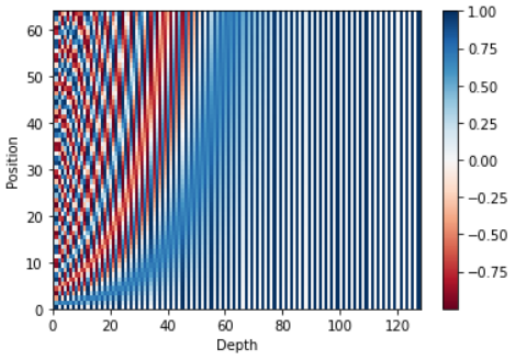
```
n_vocab = len(vocab) # vocab count
d_hidn = 128 # hidden size
nn_emb = nn.Embedding(n_vocab, d_hidn) # embedding 객체

input_embs = nn_emb(inputs) # input embedding
print(input_embs.size())
```

3.1.2 Position Embedding

- **Position encoding 값을 구하기 위한 함수**
  a. 각 position별도 angle 값을 구한다.
  b. 구해진 angle 중 짝수 index의 값에 대한 sin 값을 구한다.
  c. 구해진 angle 중 홀수 index의 값에 대한 cos 값을 구한다.

```
#sinusoid position embedding
def get_sinusoid_encoding_table(n_seq, d_hidn):
    def cal_angle(position, i_hidn):
        return position / np.power(10000, 2 * (i_hidn // 2) / d_hidn)
    def get_posi_angle_vec(position):
        return [cal_angle(position, i_hidn) for i_hidn in range(d_hidn)]

    sinusoid_table = np.array([get_posi_angle_vec(i_seq) for i_seq in range(n_seq)])
    sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2])  # even index sin
    sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2])  # odd index cos

    return sinusoid_table
```

```
n_seq = 64
pos_encoding = get_sinusoid_encoding_table(n_seq, d_hidn)

print (pos_encoding.shape) # 크기 출력
plt.pcolormesh(pos_encoding, cmap='RdBu')
plt.xlabel('Depth')
plt.xlim((0, d_hidn))
plt.ylabel('Position')
plt.colorbar()
plt.show()
```



- **position embedding 구하는 절차**
  a. 위에서 구해진 position encoding 값을 이용해 position embedding을 생성합니다. 학습되는 값이 아니므로 freeze옵션을 True로 설정
  b. 입력 inputs과 동일한 크기를 갖는 positions 값을 구한다.
  c. input값 중 pad(0)값을 찾는다.
  d. positions값중 pad부분은 0으로 변경한다.
  e. positions값에 해당하는 embedding값을 구한다.
- inputs의 pad(0) 위치에 positions의 값이 pad(0)으로 변경됨.

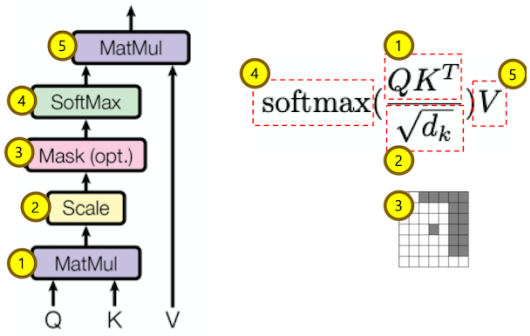
- pos_embs, input_embs의 shape → (3,13,128)

```
pos_encoding = torch.FloatTensor(pos_encoding)
nn_pos = nn.Embedding.from_pretrained(pos_encoding, freeze=True)

positions = torch.arange(inputs.size(1), device=inputs.device, dtype=inputs.dtype).expand(inputs.size(0), inputs.size(1)).contiguc
pos_mask = inputs.eq(0)

positions.masked_fill_(pos_mask, 0)
pos_embs = nn_pos(positions) # position embedding

print(inputs)
print(positions)
print(pos_embs.size())
```

- transformer에 입력할 input은 input_embs과 pos_embs를 더한 값이다.

```
input_sums = input_embs + pos_embs
```

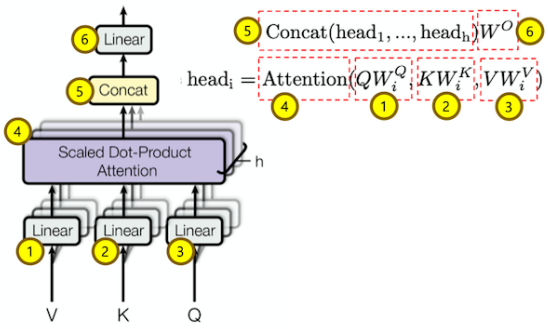
3.2 Scaled Dot Product Attention



- 입력값은 Q(query), K(key), V(value) 그리고 attention mask로 구성
- K, V는 같은 값 (Q,K,V가 모두 동일한 경우는 self attention)
  a. Q * K-transpose를 구하는 코드 → 각 단어상호간에 가중치를 표현하는 테이블 생성
  b. k-dimension에 루트를 취한 값으로 나누는 코드
  c. mask를 하는 코드
  d. softmax를 하는 코드 → 가중치가 확률로 변환된다.
  e. attn_prov * V를 하는 코드

```
#scale dot product attention
class ScaledDotProductAttention(nn.Module):
    def __init__(self, d_head):
        super().__init__()
        self.scale = 1 / (d_head ** 0.5)

    def forward(self, Q, K, V, attn_mask):
        # (bs, n_head, n_q_seq, n_k_seq)
        scores = torch.matmul(Q, K.transpose(-1, -2)).mul_(self.scale)
        scores.masked_fill_(attn_mask, -1e9)
        # (bs, n_head, n_q_seq, n_k_seq)
        attn_prob = nn.Softmax(dim=-1)(scores)
        # (bs, n_head, n_q_seq, d_v)
        context = torch.matmul(attn_prob, V)
        # (bs, n_head, n_q_seq, d_v), (bs, n_head, n_q_seq, n_v_seq)
        return context, attn_prob
```

3.3 Multi-Head Attention



입력값

- Q, K, V, attn_mask는 ScaledDotProductAttention과 동일
- head 개수는 2개 head의 dimension은 64
  a. Q를 여러개의 head로 나눈다. → Q값이 head 단위로 나눠진다.
  b. K를 여러개의 head로 나눈다. → K값이 head 단위로 나눠진다.
  c. V를 여러개의 head로 나눈다. → V값이 head 단위로 나눠진다.
  d. Attention Mask를 Multi Head로 변경한 후, Multi Head에 대한 Attention을 구한다.
  e. Multi Head를 한개로 합치고 Linear 과정을 거친다. → 입력 Q와 동일한 shape를 가진 Multi Head Attention이 구해진다.

```
#multi head attention
class MultiHeadAttention(nn.Module):
    def __init__(self, d_hidn, n_head, d_head):
        super().__init__()
        self.d_hidn = d_hidn
        self.n_head = n_head
        self.d_head = d_head

        self.W_Q = nn.Linear(d_hidn, n_head * d_head)
        self.W_K = nn.Linear(d_hidn, n_head * d_head)
        self.W_V = nn.Linear(d_hidn, n_head * d_head)
        self.scaled_dot_attn = ScaledDotProductAttention(d_head)
        self.linear = nn.Linear(n_head * d_head, d_hidn)

    def forward(self, Q, K, V, attn_mask):
        batch_size = Q.size(0)
        # (bs, n_head, n_q_seq, d_head)
        q_s = self.W_Q(Q).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)
        # (bs, n_head, n_k_seq, d_head)
        k_s = self.W_K(K).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)
        # (bs, n_head, n_v_seq, d_head)
        v_s = self.W_V(V).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)

        # (bs, n_head, n_q_seq, n_k_seq)
        attn_mask = attn_mask.unsqueeze(1).repeat(1, self.n_head, 1, 1)

        # (bs, n_head, n_q_seq, d_head), (bs, n_head, n_q_seq, n_k_seq)
        context, attn_prob = self.scaled_dot_attn(q_s, k_s, v_s, attn_mask)
        # (bs, n_head, n_q_seq, h_head * d_head)
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.n_head * self.d_head)
        # (bs, n_head, n_q_seq, e_embd)
        output = self.linear(context)
        # (bs, n_q_seq, d_hidn), (bs, n_head, n_q_seq, n_k_seq)
        return output, attn_prob
```

3.4 Masked Multi-Head Attention

- Masked Multi-Head Attention은 Multi-Head Attention과 attention mask를 제외한 부분이 모두 동일

입력값

```
# attention decoder mask
def get_attn_decoder_mask(seq):
    subsequent_mask = torch.ones_like(seq).unsqueeze(-1).expand(seq.size(0), seq.size(1), seq.size(1))
    subsequent_mask = subsequent_mask.triu(diagonal=1) # upper triangular part of a matrix(2-D)
```

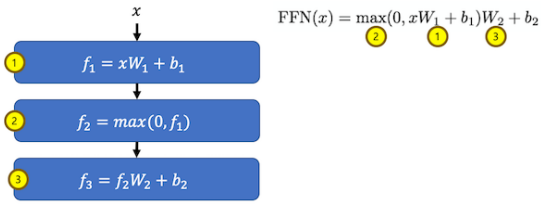
```
    return subsequent_mask


Q = input_sums
K = input_sums
V = input_sums

attn_pad_mask = inputs.eq(0).unsqueeze(1).expand(Q.size(0), Q.size(1), K.size(1))
print(attn_pad_mask[1])
attn_dec_mask = get_attn_decoder_mask(inputs)
print(attn_dec_mask[1])
attn_mask = torch.gt((attn_pad_mask + attn_dec_mask), 0)
print(attn_mask[1])

batch_size = Q.size(0)
n_head = 2
```

Multi-Head Attention

- output의 shape: (2,8,128)
- attn_prob의 shape: (2,2,8,8)

```
attention = MultiHeadAttention(d_hidn, n_head, d_head)
output, attn_prob = attention(Q, K, V, attn_mask)
print(output.size(), attn_prob.size())
```

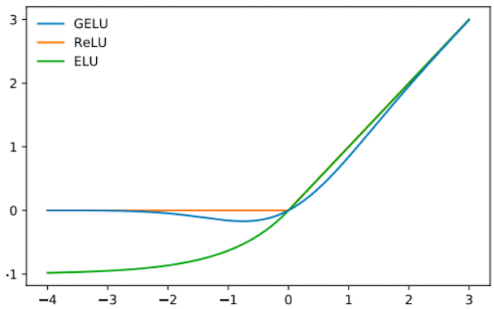
3.5 FeedForward



a. Linear 과정을 거친 후, 입력에 비해 hidden dimension이 4배 커짐.

b. Activation 과정

- ReLU보다 GELU의 성능이 더 좋다. → GELU 사용

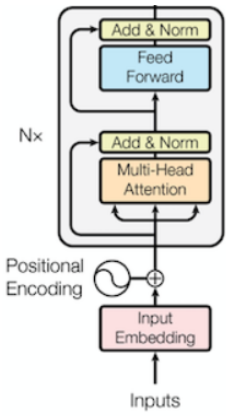


c. Linear 과정을 거쳐 입력과 동일한 shape를 가지게 함. → (2,8,128)

```
#feed forward
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self, d_hidn):
        super().__init__()

        self.conv1 = nn.Conv1d(in_channels=self.config.d_hidn, out_channels=self.config.d_hidn * 4, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=self.config.d_hidn * 4, out_channels=self.config.d_hidn, kernel_size=1)
        self.active = F.gelu

    def forward(self, inputs):
```

```
        # (bs, d_ff, n_seq)
        output = self.active(self.conv1(inputs.transpose(1, 2)))
        # (bs, n_seq, d_hidn)
        output = self.conv2(output).transpose(1, 2)
        # (bs, n_seq, d_hidn)
        return output
```

3.6 Encoder



▼ **Encoder Layer**

→ Encoder에서 루프를 돌며 처리할 수 있도록 Encoder Layer를 정의

a. Multi-Head Attention을 수행한다.

b. a번의 결과와 input을 더한 후 LayerNorm을 실행한다.

c. b번의 결과를 입력으로 Feed Forward를 실행한다.

d. c번의 결과와 b번 결과를 더한 후 LayerNorm을 실행한다.

```
class EncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        self.self_attn = MultiHeadAttention(self.config)
        self.layer_norm1 = nn.LayerNorm(self.config.d_hidn, eps=self.config.layer_norm_epsilon)
        self.pos_ffn = PoswiseFeedForwardNet(self.config)
        self.layer_norm2 = nn.LayerNorm(self.config.d_hidn, eps=self.config.layer_norm_epsilon)

    def forward(self, inputs, attn_mask):
        # (bs, n_enc_seq, d_hidn), (bs, n_head, n_enc_seq, n_enc_seq)
        att_outputs, attn_prob = self.self_attn(inputs, inputs, inputs, attn_mask)
        att_outputs = self.layer_norm1(inputs + att_outputs)
        # (bs, n_enc_seq, d_hidn)
        ffn_outputs = self.pos_ffn(att_outputs)
        ffn_outputs = self.layer_norm2(ffn_outputs + att_outputs)
        # (bs, n_enc_seq, d_hidn), (bs, n_head, n_enc_seq, n_enc_seq)
        return ffn_outputs, attn_prob
```

- **Encoder**

a. 입력에 대한 Position 값을 구한다.

b. Input Embedding과 Position Embedding을 구한 후 더한다.

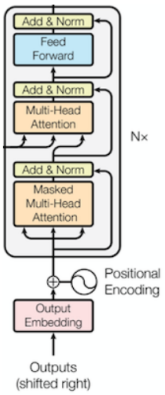
c. 입력에 대한 attention pad mask를 구한다.

d. for 루프를 돌며 각 layer를 실행한다.

```
""" encoder """
class Encoder(nn.Module):
    def __init__(self, config):
```

```
        super().__init__()
        self.config = config

        self.enc_emb = nn.Embedding(self.config.n_enc_vocab, self.config.d_hidn)
        sinusoid_table = torch.FloatTensor(get_sinusoid_encoding_table(self.config.n_enc_seq + 1, self.config.d_hidn))
        self.pos_emb = nn.Embedding.from_pretrained(sinusoid_table, freeze=True)

        self.layers = nn.ModuleList([EncoderLayer(self.config) for _ in range(self.config.n_layer)])

    def forward(self, inputs):
        positions = torch.arange(inputs.size(1), device=inputs.device, dtype=inputs.dtype).expand(inputs.size(0), inputs.size(1)).
        pos_mask = inputs.eq(self.config.i_pad)
        positions.masked_fill_(pos_mask, 0)

        # (bs, n_enc_seq, d_hidn)
        outputs = self.enc_emb(inputs) + self.pos_emb(positions)

        # (bs, n_enc_seq, n_enc_seq)
        attn_mask = get_attn_pad_mask(inputs, inputs, self.config.i_pad)

        attn_probs = []
        for layer in self.layers:
            # (bs, n_enc_seq, d_hidn), (bs, n_head, n_enc_seq, n_enc_seq)
            outputs, attn_prob = layer(outputs, attn_mask)
            attn_probs.append(attn_prob)
        # (bs, n_enc_seq, d_hidn), [(bs, n_head, n_enc_seq, n_enc_seq)]
        return outputs, attn_probs
```

3.7 Decoder



▼ **Decoder Layer**

→ Decoder에서 루프를 돌며 처리할 수 있도록 Decoder Layer를 정의

a. 1번의 결과와 input(residual)을 더한 후 LayerNorm을 실행한다.

b. Encoder-Decoder Multi-Head Attention을 수행한다.

c. c번의 결과와 b번의 결과(residual)을 더한 후 LayerNorm을 실행한다.

d. d번의 결과를 입력으로 Feed Forward를 실행한다.

e. e번의 결과와 d번의 결과(residual)을 더한 후 LayerNorm을 실행한다.

```
""" decoder layer """
class DecoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        self.self_attn = MultiHeadAttention(self.config)
        self.layer_norm1 = nn.LayerNorm(self.config.d_hidn, eps=self.config.layer_norm_epsilon)
        self.dec_enc_attn = MultiHeadAttention(self.config)
        self.layer_norm2 = nn.LayerNorm(self.config.d_hidn, eps=self.config.layer_norm_epsilon)
        self.pos_ffn = PoswiseFeedForwardNet(self.config)
        self.layer_norm3 = nn.LayerNorm(self.config.d_hidn, eps=self.config.layer_norm_epsilon)

    def forward(self, dec_inputs, enc_outputs, self_attn_mask, dec_enc_attn_mask):
        # (bs, n_dec_seq, d_hidn), (bs, n_head, n_dec_seq, n_dec_seq)
```

```
        self_att_outputs, self_attn_prob = self.self_attn(dec_inputs, dec_inputs, dec_inputs, self_attn_mask)
        self_att_outputs = self.layer_norm1(dec_inputs + self_att_outputs)
        # (bs, n_dec_seq, d_hidn), (bs, n_head, n_dec_seq, n_enc_seq)
        dec_enc_att_outputs, dec_enc_attn_prob = self.dec_enc_attn(self_att_outputs, enc_outputs, enc_outputs, dec_enc_attn_mask)
        dec_enc_att_outputs = self.layer_norm2(self_att_outputs + dec_enc_att_outputs)
        # (bs, n_dec_seq, d_hidn)
        ffn_outputs = self.pos_ffn(dec_enc_att_outputs)
        ffn_outputs = self.layer_norm3(dec_enc_att_outputs + ffn_outputs)
        # (bs, n_dec_seq, d_hidn), (bs, n_head, n_dec_seq, n_dec_seq), (bs, n_head, n_dec_seq, n_enc_seq)
        return ffn_outputs, self_attn_prob, dec_enc_attn_prob
```

- Decoder
  a. 입력에 대한 Position 값을 구한다.
  b. Input Embedding과 Position Embedding을 구한 후 더한다.
  c. 입력에 대한 attention pad mask를 구한다.
  d. 입력에 대한 decoder attention mask를 구한다.
  e. attention pad mask와 decoder attention mask 중 1곳이라도 mask되어 있는 부분인 mask 되도록 attention mask를 구한다.
  f. Q(decoder input), K(encoder output)에 대한 attention mask를 구한다.
  g. for 루프를 돌며 각 layer를 실행한다.

```
""" decoder """
class Decoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        self.dec_emb = nn.Embedding(self.config.n_dec_vocab, self.config.d_hidn)
        sinusoid_table = torch.FloatTensor(get_sinusoid_encoding_table(self.config.n_dec_seq + 1, self.config.d_hidn))
        self.pos_emb = nn.Embedding.from_pretrained(sinusoid_table, freeze=True)

        self.layers = nn.ModuleList([DecoderLayer(self.config) for _ in range(self.config.n_layer)])

    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        positions = torch.arange(dec_inputs.size(1), device=dec_inputs.device, dtype=dec_inputs.dtype).expand(dec_inputs.size(0),
        pos_mask = dec_inputs.eq(self.config.i_pad)
        positions.masked_fill_(pos_mask, 0)

        # (bs, n_dec_seq, d_hidn)
        dec_outputs = self.dec_emb(dec_inputs) + self.pos_emb(positions)

        # (bs, n_dec_seq, n_dec_seq)
        dec_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs, self.config.i_pad)
        # (bs, n_dec_seq, n_dec_seq)
        dec_attn_decoder_mask = get_attn_decoder_mask(dec_inputs)
        # (bs, n_dec_seq, n_dec_seq)
        dec_self_attn_mask = torch.gt((dec_attn_pad_mask + dec_attn_decoder_mask), 0)
        # (bs, n_dec_seq, n_enc_seq)
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs, self.config.i_pad)

        self_attn_probs, dec_enc_attn_probs = [], []
        for layer in self.layers:
            # (bs, n_dec_seq, d_hidn), (bs, n_dec_seq, n_dec_seq), (bs, n_dec_seq, n_enc_seq)
            dec_outputs, self_attn_prob, dec_enc_attn_prob = layer(dec_outputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask
            self_attn_probs.append(self_attn_prob)
            dec_enc_attn_probs.append(dec_enc_attn_prob)
        # (bs, n_dec_seq, d_hidn), [(bs, n_dec_seq, n_dec_seq)], [(bs, n_dec_seq, n_enc_seq)]S
        return dec_outputs, self_attn_probs, dec_enc_attn_probs
```

3.8 Transformer

a. Encoder Input을 입력으로 Encoder를 실행한다.
b. Encoder Ouput과 Decoder Input을 입력으로 Decoder를 실행한다.

```
""" transformer """
class Transformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
```

```
        self.encoder = Encoder(self.config)
        self.decoder = Decoder(self.config)

    def forward(self, enc_inputs, dec_inputs):
        # (bs, n_enc_seq, d_hidn), [(bs, n_head, n_enc_seq, n_enc_seq)]
        enc_outputs, enc_self_attn_probs = self.encoder(enc_inputs)
        # (bs, n_seq, d_hidn), [(bs, n_head, n_dec_seq, n_dec_seq)], [(bs, n_head, n_dec_seq, n_enc_seq)]
        dec_outputs, dec_self_attn_probs, dec_enc_attn_probs = self.decoder(dec_inputs, enc_inputs, enc_outputs)
        # (bs, n_dec_seq, n_dec_vocab), [(bs, n_head, n_enc_seq, n_enc_seq)], [(bs, n_head, n_dec_seq, n_dec_seq)], [(bs, n_head, n_de
        return dec_outputs, enc_self_attn_probs, dec_self_attn_probs, dec_enc_attn_probs
```

4. Evaluate & Train

- 학습된 ShoppingClassification 모델의 성능을 평가하기 위해 정확도(accuracy) 사용

4.1 ShoppingClassification 모델 학습

a. Encoder input과 Decoder input을 입력으로 ShoppingClassification을 실행

b. a번의 결과 중 첫 번째 값 → 예측 logits

c. logits 값과 labels의 값을 이용해 Loss를 계산한다.

d. loss, optimizer를 이용해 학습한다.

```
""" 모델 epoch 학습 """
def train_epoch(config, epoch, model, criterion, optimizer, train_loader):
    losses = []
    model.train()

    with tqdm(total=len(train_loader), desc=f"Train {epoch}") as pbar:
        for i, value in enumerate(train_loader):
            labels, enc_inputs, dec_inputs = map(lambda v: v.to(config.device), value)

            optimizer.zero_grad()
            outputs = model(enc_inputs, dec_inputs)
            logits = outputs[0]

            loss = criterion(logits, labels)
            loss_val = loss.item()
            losses.append(loss_val)

            loss.backward()
            optimizer.step()

            pbar.update(1)
            pbar.set_postfix_str(f"Loss: {loss_val:.3f} ({np.mean(losses):.3f})")
    return np.mean(losses)
```

- 선언한 내용('GPU 사용 여부, 출력 값의 개수, learning_rate, 학습 epoch')을 바탕으로 학습 진행
- 10 epoch 학습

```
config.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
config.n_output = 2
print(config)

learning_rate = 5e-5
n_epoch = 10

model = ShoppingClassification(config)
model.to(config.device)

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

losses, scores = [], []
for epoch in range(n_epoch):
    loss = train_epoch(config, epoch, model, criterion, optimizer, train_loader)
    score = eval_epoch(config, model, test_loader)

    losses.append(loss)
    scores.append(score)
```

5. Result

- **정확도(score) → 90.5%**

epoch	loss	score
1	0.364576	88.03%
2	0.290578	89.1%
3	0.268474	89.5%
4	0.251539	89.6%
5	0.233904	90.0%
6	0.219072	90.1%
7	0.203699	90.4%
8	0.190238	**90.5%**
9	0.174301	90.29%
10	0.160867	90.24%