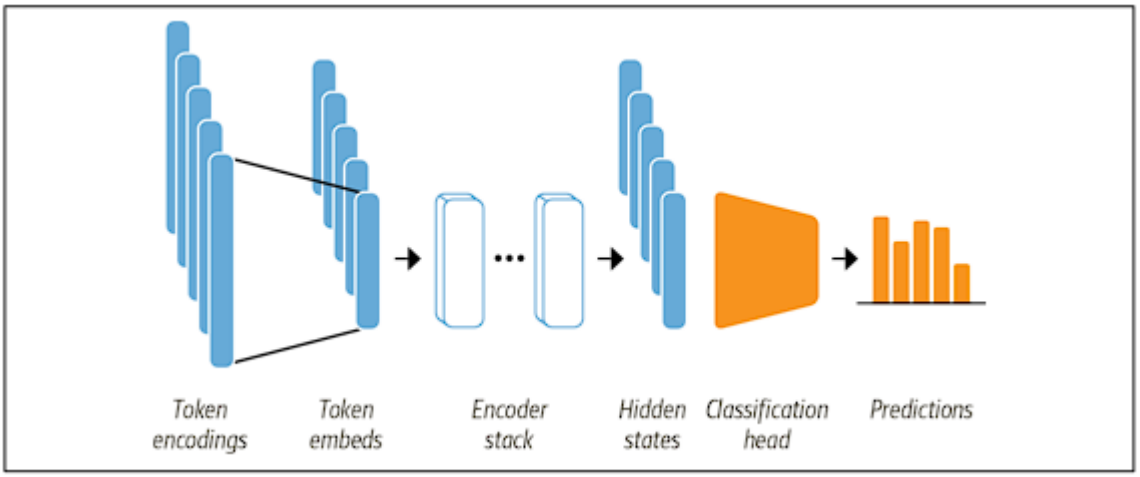
2주차 - 텍스트 분류 2.3절~3.5절

2.3 텍스트 분류 모델 훈련하기

인코더 기반 모델의 구조



- 토큰 인코딩
  - 텍스트를 토큰화하여 label encoding을 진행
  - 토크나이저의 어휘사전의 크기 = 토큰 인코딩 차원 결정
- 토큰 임베딩
  - 인코더 블록 층 통과시켜서 토큰 인코딩의 임베딩을 획득
- 은닉 상태
  - 토큰 임베딩을 인코더 블록 층에 통과시켜 각 입력 토큰에 대한 은닉 상태 만듦.
  - 사전 훈련 목표를 달성하기 위해 마스킹된 입력 토큰을 예측하는 층으로 전달됨.
- 분류 헤드
  - 언어 모델링층을 분류 층으로 바꿈

```
class MaskedLanguageModel(nn.Module):
    """
    predicting origin token from masked input sequence
    n-class classification problem, n-class = vocab_size
    """
```

```
    def __init__(self, hidden, vocab_size):
        """
        :param hidden: output size of BERT model
        :param vocab_size: total vocab size
        """
        super().__init__()
        self.linear = nn.Linear(hidden, vocab_size)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self, x):
        return self.softmax(self.linear(x))

...

for i, data in data_iter:
    # 0. batch_data will be sent into the device(GPU or cpu)
    data = {key: value.to(self.device) for key, value in data

    # 1. forward the next_sentence_prediction and masked_lm m
    next_sent_output, mask_lm_output = self.model.forward(dat
```

- `mask_lm_output`
  - Masked Language Model(MLM)의 출력
  - 모델이 마스킹된 토큰을 예측하는 작업
  - 입력 시퀀스의 각 토큰에 대한 은닉 상태를 생성
  - `mask_lm_output`는 각 토큰에 대한 예측 확률을 포함하는 텐서
- `MaskedLanguageModel`:
  - 입력 문장의 각 토큰에 대한 임베딩을 출력 해줌
  - `[batch_size, n_tokens, hidden_dim]`
- last hidden state:

```
model = AutoModel.from_pretrained(model_name).to(device)
outputs = model(tokenizer(text, return_tensors='pt'))
```

```
lhs = outputs.last_hidden_state[:, 0]
```

- BERT의 2가지 output:
  - `last_hidden_state` `pooler_output`
- **`last_hidden_state`**
  - Contains the hidden representations for each token in each sequence of the batch
  - `[batch_size, n_tokens, hidden_dim]`
- **`pooler_output`**
  - a "representation" of each sequence in the batch
  - `[batch_size, hidden_size]`
  - take the hidden representation of the [CLS] token of each sequence in the batch (which is a vector of size `hidden_size`), and then run that through the `BertPooler` nn.Module

```
class BertPooler(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, conf
        self.activation = nn.Tanh()

    def forward(self, hidden_states):
        # We "pool" the model by simply taking the hidd
        # to the first token.
        first_token_tensor = hidden_states[:, 0]
        pooled_output = self.dense(first_token_tensor)
        pooled_output = self.activation(pooled_output)
        return pooled_output
```

    - The weights of **this linear layer are already pretrained on the next sentence prediction task** (note that BERT is pretrained on 2 tasks: masked language modeling and next sentence prediction).
  - This consists of a linear layer followed by a Tanh activation function

2.3.2 트랜스포머 미세 튜닝하기

사전 훈련된 모델 로드하기

```
from transformers import AutoModelForSequenceClassification

num_labels = 6
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt, num_labels=num_labels)
         .to(device))
```

성공 지표 정의하기

```
from sklearn.metrics import accuracy_score, f1_score

def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    f1 = f1_score(labels, preds, average="weighted")
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "f1": f1}
```

- 훈련하는 동안 성능을 모니터링
- Trainer에 사용할 `compute_metrics()`
- `dict` 자료형을 반환해야 함.
- The `Trainer` accepts a `compute_metrics` keyword argument that passes a function to compute metrics.
    - One can specify the evaluation interval with `evaluation_strategy` in the `TrainerArguments.`

```
# Setup evaluation
metric = evaluate.load("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references
```

- 분류 모델에 대한 평가 코드

```
# Setup evaluation
nltk.download("punkt", quiet=True)
metric = evaluate.load("rouge")

def compute_metrics(eval_preds):
    preds, labels = eval_preds

    # decode preds and labels
    labels = np.where(labels != -100, labels, tokenizer.pad_t
    decoded_preds = tokenizer.batch_decode(preds, skip_specia
    decoded_labels = tokenizer.batch_decode(labels, skip_spec

    # rougeLSum expects newline after each sentence
    decoded_preds = ["\n".join(nltk.sent_tokenize(pred.strip(
    decoded_labels = ["\n".join(nltk.sent_tokenize(label.stri

    result = metric.compute(predictions=decoded_preds, refere
    return result
```

- Seq2Seq task - 텍스트 요약 또는 생성적인 작업에 대한 모델의 성능을 평가
- ROUGE 평가 지표는 텍스트 요약 및 생성적인 작업에서 많이 사용
  - 모델의 요약 결과와 실제 요약 사이의 유사성을 측정
- 레이블(labels) 배열에서 값이 -100인 부분:
  - 모델 내부적으로 손실 함수(loss function) 계산 시 무시되도록 설계
  - 모델이 특정 위치에 대해 레이블을 예측하지 않도록 학습
  - -100 값이 들어있는 레이블을 모델의 패딩 토큰으로 처리하여 올바른 평가를 수행

모델 훈련하기

- `TrainingArguments`

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
```

```
    num_train_epochs=1,                # total number of tra
    per_device_train_batch_size=1,     # batch size per devi
    per_device_eval_batch_size=10,     # batch size for eval
    warmup_steps=1000,                 # number of warmup st
    weight_decay=0.01,                 # strength of weight
    logging_dir='./logs',              # directory for stori
    logging_steps=200,                 # How often to print
    do_train=True,                     # Perform training
    do_eval=True,                      # Perform evaluation
    evaluation_strategy="epoch",       # evalute after eachh
    gradient_accumulation_steps=64,    # total number of ste
    fp16=True,                         # Use mixed precision
    fp16_opt_level="02",               # mixed precision mod
    run_name="ProBert-BFD-MS",         # experiment name
    seed=3                             # Seed for experiment
)
```

- `fp16`:
  - 16-bit floating point precision을 사용하여 모델을 학습할지 여부를 결정
  - GPU 메모리 사용량을 줄이고 학습 속도를 향상
- `gradient_accumulation_steps`:
  - 그래디언트 업데이트를 여러 미니배치에 걸쳐 누적할 스텝 수를 나타냄
  - 더 큰 배치 크기를 처리할 수 있음
  - `per_device_train_batch_size` * `gradient_accumulation_steps`가 실제로 사용되는 배치 크기
  - `gradient_accumulation_steps`가 64로 설정되어 있으므로, 실제로는 각 장치당 배치 크기가 64인 미니배치가 1개의 업데이트를 수행
  - 필요한 상황: 만약 64라는 batch_size로 학습하는 것이 최적의 파라미터이고 batch_size가 최대 8밖에 못 쓸 경우, `gradient_accumulation_steps`=8로 두면, `gradient_accumulation_steps` 만큼 배치가 지나고 나서 누적된 연산 결과를 통해 연산을 수행하게 되서, 8 * 8 = 64 배치의 효과가 발생함.
    - 훈련 시간이 줄어드는 것은 아님.
- `fp16_opt_level`:

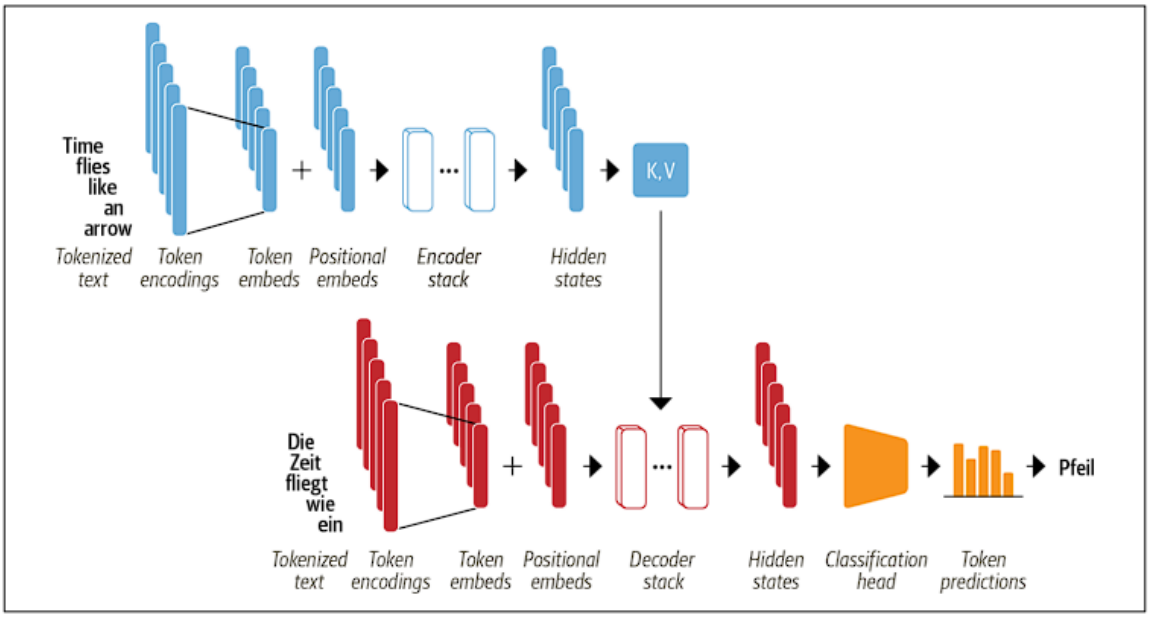
- mixed precision은 일부 연산을 32-bit로 처리하고 다른 연산을 16-bit로 처리하여 계산 속도를 향상시키는 방법
- 이 파라미터는 16-bit floating point precision을 사용하는 경우 사용할 mixed precision optimization 레벨을 나타냄
- "O2"로 설정되어 있으므로 중간 레벨의 최적화가 적용

- `evaluation_strategy`:
  - 모델을 언제 평가할지를 결정

3장 트랜스포머 파헤치기

3.1 트랜스포머 아키텍처

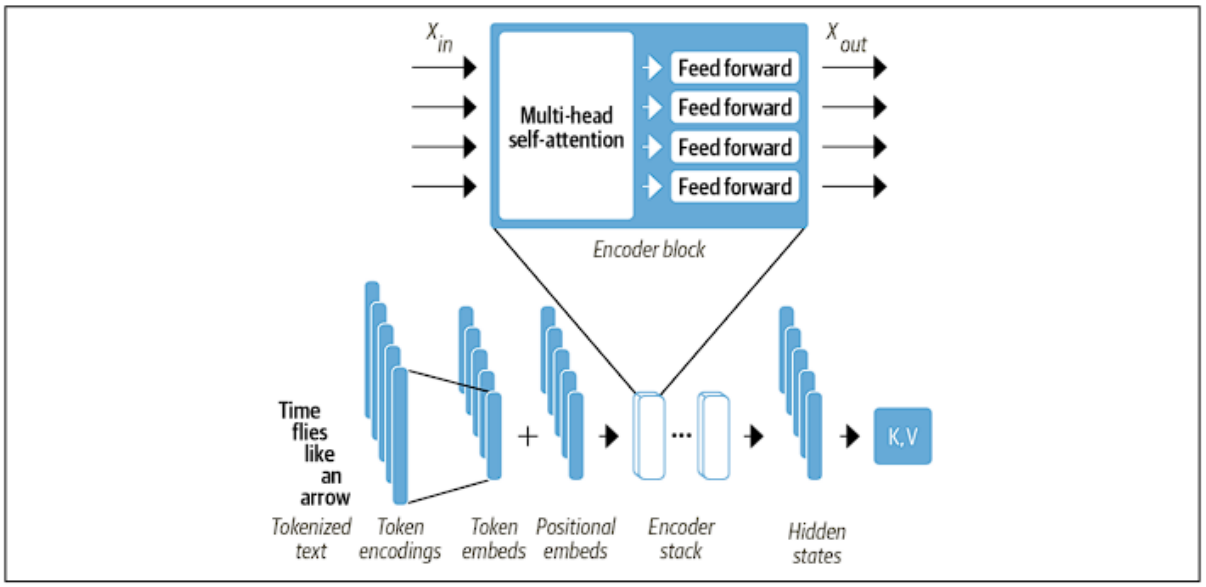
- 트랜스포머는 인코더-디코더 구조를 기반으로 함
- 인코더
  - 입력 토큰의 시퀀스를 은닉 상태라는 임베딩 벡터로 변환
- 디코더
  - 인코더의 은닉 상태를 사용해서 출력 토큰의 시퀀스를 한 번에 하나씩 반복적으로 생성



  - 디코더 층마다 인코더의 출력이 주입됨.
  - 디코더는 시퀀스에서 가장 가능성 있는 다음 토큰을 예측함.

- 출력된 토큰은 디코더로 다시 주입되서 다음 토큰을 생성함.
- 이 과정을 EOS(end-of-sequence) 토큰이 나올 때까지 반복함.

3.2 인코더



- 인코더는 여러 개의 인코더 층이 서로 쌓여서 구성됨 (encoder stack)
- 인코더 layer 구성
  - 멀티 헤드 셀프 어텐션 층
  - 완전 연결 피드 포워드 층

3.2.1 셀프 어텐션

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

- $x_1, x_2, \ldots, x_n \rightarrow x'_1, x'_2, \ldots, x'_n$
- 입력 시퀀스의 각 토큰들 간의 관련성을 계산하여 토큰별로 임베딩의 크기를 조절하여, 문맥을 더 많이 내포할수록 더 많은 가중치를 할당함.

- contextualized embedding

스케일드 점곱 어텐션

1. 토큰화된 텍스트에 대해, q, k, v 각각에 대해 임베딩 벡터 획득
2. q와 k 간의 dot product를 통해 어텐션 score 계산(가중치)
    - 쿼리와 키가 비슷하면 값이 크고
    - 겹치는 부분이 적을수록 값이 작음
    - n by n 어텐션 score 행렬 형성
3. 어텐션 score 행렬에 scaling factor를 곱하고 분산을 정규화하고 스프트맥스 함수를 적용해 모든 열의 합이 1이 되게 함. 어텐션 가중치 획득
4. 토큰 임베딩 업데이트: 어텐션 가중치와 v를 곱해서 토큰 임베딩 업데이트

```
def scaled_dot_product_attention(query, key, value):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim
    weights = F.softmax(scores, dim=-1)
    return torch.bmm(weights, value)
```

멀티 헤드 어텐션

- 어텐션 헤드:
    - 토큰 임베딩에 q, k, v 벡터를 활용해서 토큰 임베딩을 업데이트하는 부분

    ```
    class AttentionHead(nn.Module):
        def __init__(self, embed_dim, head_dim):
            super().__init__()
            self.q = nn.Linear(embed_dim, head_dim)
            self.k = nn.Linear(embed_dim, head_dim)
            self.v = nn.Linear(embed_dim, head_dim)

        def forward(self, hidden_state):
            attn_outputs = scaled_dot_product_attention(
                self.q(hidden_state), self.k(hidden_state), se
            return attn_outputs
    ```

- 멀티 헤드 어텐션:

- 여러 개의 어텐션 헤드를 둠으로써, 다양한 측면의 유사도를 활용할 수 있게 된다.
- 컴퓨터 비전에서의 filter과 같은 효과

```
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads
        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for _ in r
        )
        self.output_linear = nn.Linear(embed_dim, embed_di

    def forward(self, hidden_state):
        x = torch.cat([h(hidden_state) for h in self.heads
        x = self.output_linear(x)
        return x
```

- emb_dim = 768, num_heads = 8
- `h(hidden_state).shape`
  - `[batch_size, n_tokens, 96]`
- `torch.cat([h(hidden_state) for h in self.heads], dim=-1).shape`
  - `[batch_size, n_tokens, 96 * 8 = 768]`

3.2.2 피드 포워드 층

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.
        self.linear_2 = nn.Linear(config.intermediate_size, c
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
```

```
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

- config.intermediate_size = config.hidden_size * 4
- hidden_size를 늘려서 비선형성을 증가시킴
- 계산 비용 낮추기 위해 원래 크기로 줄임

3.2.3 층 정규화 추가하기

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # 층 정규화를 적용하고 입력을 쿼리, 키, 값으로 복사합니다.
        hidden_state = self.layer_norm_1(x)
        # 어텐션에 스킵 연결을 적용합니다.
        x = x + self.attention(hidden_state)
        # 스킵 연결과 피드 포워드 층을 적용합니다.
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

3.2.4 위치 임베딩

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_siz
                                             config.hidden_si
        self.position_embeddings = nn.Embedding(config.max_po
```

```
                                                    config.hidden
        self.layer_norm = nn.LayerNorm(config.hidden_size, ep
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # 입력 시퀀스에 대해 위치 ID를 만듭니다.
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, dtype=torch.l
        # 토큰 임베딩과 위치 임베딩을 만듭니다.
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(positi
        # 토큰 임베딩과 위치 임베딩을 합칩니다.
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

- 데이터셋이 적을 때
  - 포지션
- 데이터셋이 많을 때
  - 임베딩

```
class TransformerEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = Embeddings(config)
        self.layers = nn.ModuleList([TransformerEncoderLayer(
                                     for _ in range(config.nu

    def forward(self, x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

```
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, confi

    def forward(self, x):
        x = self.encoder(x)[:, 0, :] # [CLS] 토큰의 은닉 상태를 선
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

3.3 디코더

```
seq_len = inputs.input_ids.size(-1)
mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0)
mask[0]
```

```
scores.masked_fill(mask == 0, -float("inf"))
```

- 음의 무한대 → 소프트맥스 함수 적용 시 0으로, 어텐션 시 고려하지 않음.

```
tensor([[[26.6014,    -inf,    -inf,    -inf,    -inf],
         [-1.4428, 29.7577,    -inf,    -inf,    -inf],
         [-0.0424, -1.2473, 28.9098,    -inf,    -inf],
         [ 0.6426, -0.1071,  0.7338, 27.5270,    -inf],
         [ 0.0342, -1.5766,  2.2961,  0.7159, 28.5473]]],
       grad_fn=<MaskedFillBackward0>)
```

```
def scaled_dot_product_attention(query, key, value, mask=None
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float("-inf"))
```

```
    weights = F.softmax(scores, dim=-1)
    return weights.bmm(value)
```