NLP 6주차 스터디

작성일시	@2024년 2월 22일 오전 8:49
복습	☐

8.1 의도 탐지 예제

의도 탐지 예제를 통해 트랜스포머 모델의 메모리 사용량을 줄이는 기술의 장단점 살펴보기

▫ 예제

고객이 상담원과 대화할 필요 없도록 회사의 콜 센터 전용 텍스트 기반 **어시스턴트**를 만든다 가정

[고객이 보낸 메세지]

> 📣 Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in Paris and I need a 15 passenger van

- 의도 분류기는 이를 자동으로 `Car Rental`로 분류하고 행동을 선택해 응답해야함
- 이 분류기가 안정적으로 수행되려면 예상 범위 밖의 쿼리도 처리해야함

그림 8-2 사람(오른쪽)과 텍스트 기반 어시스턴트(왼쪽)가 개인 금융 정보에 대해 나눈 대화

2번 - 고객이 범위 안에 없는 스포츠 질문을 하자 어시스턴트는 범위 안에 있는 의도로 잘못 분류해

급여일을 알려주는 중

3번 - 카테고리에 없는 쿼리를 감지하는 훈련을 받은 텍스트 어시스턴트는 고객에게 응답이 가능한

주제를 알려줌

사용할 데이터셋 및 기준 모델

- CLINC150 데이터셋에서 미세 튜닝해 약 94% 정확도를 달성한 BERT 베이스 모델을 기준 모델로 사용
- 이 데이터셋에는 150개의 의도와 10개 분야로 분류된 22,500개 쿼리가 포함
- 범위를 벗어난 쿼리도 1,200개 > 의도 클래스 **oos** (out of scope)에 속함

▼ 🤗 허브에서 모델 다운 후 텍스트 분류 파이프라인 준비

```
from transformers import pipeline

bert_ckpt = "transformersbook/bert-base-uncased-finetuned-
pipe = pipeline("text-classification", model=bert_ckpt)
```

▼ 쿼리 전달하기

```
query = """Hey, I'd like to rent a vehicle from Nov 1st to
Paris and I need a 15 passenger van"""
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.5490034222602844}]
```

모델로부터 예측한 의도와 신뢰도 점수를 얻음

8.2 벤치마크 클래스 만들기

기준 모델의 성능을 평가할 벤치마크 클래스 만들기

트랜스포머를 제품 환경에 배포하려면 여러 가지 제약 조건을 해결해야함

- **모델 성능**

모델이 얼마나 잘 작동하는가

모델 성능은 특히 오류가 발생했을 때 손실 비용이 큰 상황이나, 수백만 개의 샘플에서 추론을

실행해야 하므로 모델 지표가 조금 향상되면 전체적으로 큰 이득을 얻을 수 있는 상황에서 중요함.

- **레이턴시**

모델이 얼마나 빠르게 예측을 만드는가

레이턴시는 보통 대량의 트래픽을 처리하는 실시간 환경에서 고려함

- **메모리**

메모리는 모바일과 에지 장치(IoT 디바이스, 센서 등)에서 특별히 중요한 역할을 수행함

이 환경에서는 모델이 클라우드 서버에 접속하지 않고 예측을 만들어야함

> 쓸 수 있는 메모리가 작아서겟지

▼ 제약을 다양한 압축 기법으로 최적화하는 방법을 알아보기 위해 벤치마크 클래스 만들기

```
# 파이프라인과 test set이 주어지면 성능을 측정
class PerformanceBenchmark:
    def __init__(self, pipeline, dataset, optim_type="BERT
        self.pipeline = pipeline
        self.dataset = dataset
        self.optim_type = optim_type

    def compute_accuracy(self):
        # 나중에 정의합니다
        pass

    def compute_size(self):
        # 나중에 정의합니다
        pass

    def time_pipeline(self):
        # 나중에 정의합니다
        pass

    def run_benchmark(self):
        metrics = {}
        metrics[self.optim_type] = self.compute_size()
        metrics[self.optim_type].update(self.time_pipeline
        metrics[self.optim_type].update(self.compute_accur
        return metrics
```

- `optim_type` : 여러 가지 최적화 기법의 성능을 추적하기 위해 정의한 매개변수
- `run_benchmark()` 메서드를 사용해 딕셔너리에 `optim_type`을 키로 모든 지표를 저장

▼ 테스트셋에서 모델 정확도를 계산하도록 클래스 완성하기

```
# 테스트할 데이터셋 다운
from datasets import load_dataset
```

```
clinc = load_dataset("clinc_oos", "plus")
```

- `plus` 설정은 범위 밖의 훈련 샘플이 담긴 subset을 의미 >> 추가 데이터

▼ test set의 샘플 하나 살펴보기

```
sample = clinc["test"][42]
sample
```

```
{'text': 'transfer $100 from my checking to saving account', 'intent': 133}
```

- CLINC150의 각 샘플은 **text** 열에 있는 쿼리와 이에 상응하는 의도(**intent**)로 구성됨

▼ `.features[]` 속성 사용

```
intents = clinc["test"].features["intent"]
intents.int2str(sample["intent"])
```

```
'transfer'
```

- 의도는 ID로 제공되지만 `features` 속성을 사용하면 문자열로 쉽게 매핑됨

▼ PerformanceBenchmark의 `compute_accuracy()` 메서드 구현

```
import evaluate
```

```
accuracy_score = evaluate.load("accuracy")
```

이 데이터셋은 의도 클래스 간에 **균형**이 잡혀 있으므로 성능 지표로 **정확도** 사용

▼ 데이터셋의 모든 예측과 레이블을 리스트로 취합한 후 정확도를 계산해 반환

```
def compute_accuracy(self):
    """PerformanceBenchmark.compute_accuracy() 메서드를 오버라
    preds, labels = [], []
    for example in self.dataset:
        pred = self.pipeline(example["text"])[0]["label"]
        label = example["intent"]
        preds.append(intents.str2int(pred))
        labels.append(label)
    accuracy = accuracy_score.compute(predictions=preds, r
    print(f"테스트 세트 정확도 - {accuracy['accuracy']:.3f}")
    return accuracy

PerformanceBenchmark.compute_accuracy = compute_accuracy
```

- 정확도 지표는 **정수**로 표현된 예측과 정답 레이블을 기대하기 때문에

  파이프라인을 사용해 text 필드에서 예측을 추출하고 intents 객체의 str2int() 메서드를

  사용해 각 예측을 해당 ID로 매핑하였음

- 그 후 이 메서드를 PerformanceBenchmark 클래스에 추가

▼ 모델 저장

```
torch.save(pipe.model.state_dict(), "model.pt")
```

- 파이토치의 `torch.save()` 함수를 사용해 모델을 저장

- 파이토치에서는 모델을 저장할 때 `state.dict()` 메서드 사용 추천
  이 메서드는 모델의 층과 학습 가능한 파라미터(가중치와 편향)를 매핑하는 파이썬 딕셔너리 반환

▼ 기준 모델의 `state_dict()` 메서드가 반환한 내용 확인

```
list(pipe.model.state_dict().items())[42]
```

```
('bert.encoder.layer.2.attention.self.value.bias',
 tensor([-2.7834e-02,  4.9434e-02,  8.3551e-02,  4.1092e-02,  6.0157e-01,
          1.1774e-01, -5.2112e-02, -6.5143e-02, -2.9358e-02, -4.2250e-02,
          7.9177e-02,  8.0409e-02,  2.9921e-03,  1.7816e-01, -5.0480e-02,
         -1.5634e-01, -2.1707e-02,  1.4381e-02,  2.5132e-02, -2.4110e-02,
         -1.9183e-01, -7.8657e-02,  5.0709e-02,  3.3632e-02, -3.1946e-02,
          1.1616e-01,  9.2720e-02, -1.1787e-01,  2.3233e-01, -1.2678e-02,
         -1.3138e-01, -4.0024e-02,  7.4823e-02, -5.4148e-02, -1.5184e-01,
         -7.4407e-02,  1.1559e-01,  8.2729e-02, -1.3787e-01,  8.3528e-02,
          1.2154e-01,  1.6880e-02, -5.6629e-02, -3.9295e-02,  5.3725e-02,
          6.8602e-02, -1.1294e-01,  4.4001e-02, -2.5884e-01,  1.6767e-01,
          1.8316e-01,  5.6272e-02, -3.6874e-02, -2.7938e-02, -9.3204e-02,
         -7.5239e-03,  4.1141e-02, -1.1542e-02, -9.9749e-02, -3.0910e-02,
          4.1398e-02, -4.4389e-02, -2.6279e-02,  7.2100e-02,  7.5179e-03,
         -7.4382e-03,  2.9311e-02, -1.3391e-02,  6.9966e-03, -9.3249e-03,
          9.4272e-03, -1.1783e-02,  1.3849e-02,  1.8157e-03, -1.1522e-02,
          1.3364e-02, -2.6307e-02,  2.3725e-03, -4.8451e-03,  6.2261e-03,
          1.2653e-02,  1.7601e-02, -1.7971e-02, -2.9247e-03, -3.3447e-03,
          1.4263e-02, -3.5629e-03, -9.2794e-03, -2.1326e-02,  1.9390e-02,
```

각각의 키/값 쌍이 BERT의 층과 텐서에 해당.

▼ 성능 지표 중 크기 부분 클래스 채워주기

```
import torch
from pathlib import Path
```

```
def compute_size(self):
    """PerformanceBenchmark.compute_size() 메서드를 오버라이드
    state_dict = self.pipeline.model.state_dict()
    tmp_path = Path("model.pt")
    torch.save(state_dict, tmp_path)
    # 메가바이트 단위로 크기를 계산합니다
    size_mb = Path(tmp_path).stat().st_size / (1024 * 1024
    # 임시 파일을 삭제합니다
    tmp_path.unlink()
    print(f"모델 크기 (MB) - {size_mb:.2f}")
    return {"size_mb": size_mb}

PerformanceBenchmark.compute_size = compute_size
```

- `Path.stat()` : 저장된 파일의 정보 얻기
- `Path("model.pt").stat().st_size` 에는 모델 크기가 메가바이트 단위로 저장됨

이 정보를 모두 사용해 `compute_size()` 함수를 만들고 PerformanceBenchmark 클래스에 추가

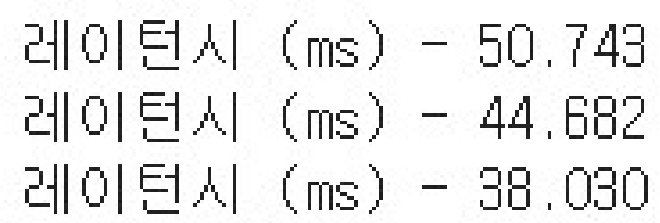
▼ 쿼리마다 평균적인 레이턴시 측정하기 위한 `time_pipeline()` 구현

```
# 파이프라인에 테스트 쿼리를 전달하고
from time import perf_counter

for _ in range(3):
        #  perf_counter()를 사용해 코드 실행의 시작과 끝 시간의 차
    start_time = perf_counter()
    _ = pipe(query)
    latency = perf_counter() - start_time
    print(f"레이턴시 (ms) - {1000 * latency:.3f}")
```

- 여기서 **레이턴시**는 파이프라인에 텍스트 쿼리를 주입해서 모델로부터 예측된 의도가 반환되기까지 걸린 시간을 의미

- 파이프라인은 내부적으로 텍스트를 토큰화하지만 이 작업은 예측을 생성하는 것보다

  천 배 가량 더 빠름. >> 토큰화 작업이 전체 레이턴시에 미치는 영향은 무시!



>> 평균 레이턴시 차이가 큼.

코드를 실행할 때마다 실행 시간 결과가 **달라져서** 파이프라인을 여러 번 실행해 레이턴시를 수집하여 값이 얼마나 퍼져 있는지 파악해보기!

```
import numpy as np

def time_pipeline(self, query="What is the pin number for
    """PerformanceBenchmark.time_pipeline() 메서드를 오버라이드
    latencies = []

    # CPU 워밍업 왜?? >> 시스템 초기화 및 최적화에 굿
    for _ in range(10):
        _ = self.pipeline(query)
    # 실행 측정
    for _ in range(100):
        start_time = perf_counter()
        _ = self.pipeline(query)

                # 실행시간 측정 후 시작 시간 빼서 레이턴시 측정
        latency = perf_counter() - start_time
        latencies.append(latency)
```

```
    # 평균과 표준편차 계산
    time_avg_ms = 1000 * np.mean(latencies)
    time_std_ms = 1000 * np.std(latencies)

        # 밀리초 단위로 변환
    print(f"평균 레이턴시 (ms) - {time_avg_ms:.2f} +\- {time_
    return {"time_avg_ms": time_avg_ms, "time_std_ms": tim

# 클래스에 추가
PerformanceBenchmark.time_pipeline = time_pipeline
```

보통 레이턴시는 쿼리 길이에 따라 달라지므로 제품 환경에서 마주하게 될 쿼리를 사용해

모델을 벤치마킹하는 것이 좋음

▼ 클래스 완성. 사용해보기 ><

```
# BERT 기준 모델로 시작
pb = PerformanceBenchmark(pipe, clinc["test"])
perf_metrics = pb.run_benchmark()
```

기준 모델의 경우 벤치마크를 수행할 파이프라인과 데이터셋만 전달하면 된다

각 모델의 성능을 추적하기 위해 `perf_metrics` 딕셔너리에 결과를 저장

```
모델 크기 (MB) - 418.15
평균 레이턴시 (ms) - 28.53 +₩- 2.69
테스트 세트 정확도 - 0.867
```

기준 점수 획득 완

8.3 지식 정제로 모델 크기 줄이기

지식 정제

- 느리고 크지만 성능은 더 좋은 **티처**의 동작을 모방하도록 작은 **스튜던트** 모델을 훈련
- 언어 모델의 파라미터 개수가 꾸준히 증가하는 경향을 고려할 때,
  대규모 모델을 **압축**해 실용적으로 애플리케이션 구축을 할 수 있는 인기있는 방법

훈련하는 동안 지식은 어떻게 정제되고 티처에서 스튜던트로 전달될까?

8.3.1 미세 튜닝에서의 지식 정제

미세 튜닝 같은 지도 학습에서는 티처의 **소프트 확률**로 정답 레이블을 보강해서
스튜던트가 학습할 때 **부가 정보**를 제공하는 것이 주요 아이디어!

예를 들어, BERT 기반 분류기가 **여러 개**의 의도에 높은 확률을 할당한다면,
이 의도는 특성 공간 안에서 서로 가까이 위치한다는 신호일 가능성이 있음.
이런 확률을 **모방**하도록 스튜던트를 훈련해서 티처가 학습한 검은 지식을 정제함.

- 검은 지식 : 레이블만으로는 얻지 못하는 지식

▫ 수학적으로 살펴보기

티처

$$\frac{\exp\left(z_i(x)\right)}{\sum_j \exp\left(z_i(x)\right)}$$

1. 입력 시퀀스 x를 티처에 전달해 로짓 벡터 $z(x) = [z_1(x), ..., z_N(x)]$ 를 생성
2. 이 로짓벡터를 소프트 맥스 함수에 적용하면 **확률**로 변환됨

하지만 대부분 티처가 **한** 클래스에 높은 확률을 할당해서 나머지 클래스 확률이 **0**에 가까워짐

이 경우 티처는 정답 레이블 외에 **추가 정보**를 많이 제공하지 않음 ㅜㅜ

$$p_i(x) = \frac{\exp(z_i(x)/T)}{\sum_j \exp(z_i(x)/T)}$$

따라서 소프트 맥스 함수를 적용하기 전에 **온도 파라미터 T**로 로짓의 **스케일**을 조정해 확률을 소프트하게 만들어야함.

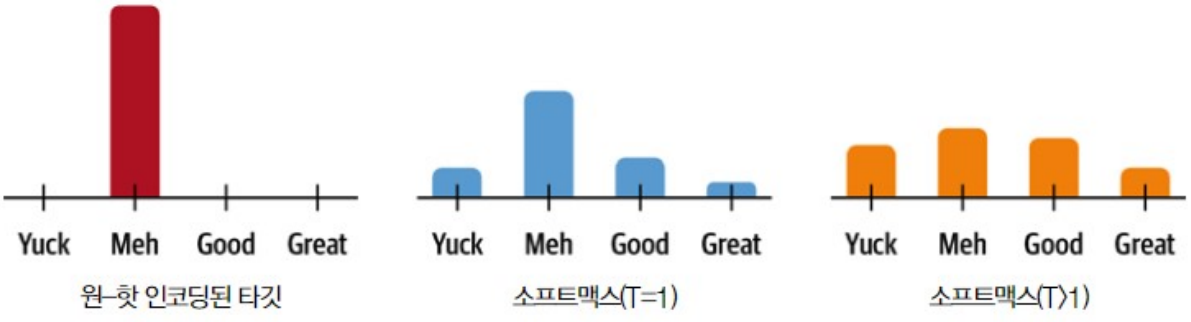


그림 8-3 원-핫 인코딩 레이블(왼쪽)과 소프트맥스 확률(중간), 온도 파라미터를 적용한 클래스 확률(오른쪽)의 비교

- 주황이 같이 **높은 T값**은 클래스에 대해 완만한 확률 분포를 만듦

  티처가 각 훈련 샘플로부터 학습한 결정 경계에 대한 정보가 더 많이 드러남 ?

  >> 아마 확률 분포가 완만하면 클래스 간 경계가 흐물하니까 모델이 더 복잡한 패턴을 학습해야돼서 그런게 아닐까 ~ ㅅ ~

스튜던트

$$D_{KL}(p,\ q) = \sum_i p_i(x)\, log \frac{p_i(x)}{q_i(x)}$$

스튜던트도 자신의 소프트 확률 $q_i(x)$을 만들 수 있으므로 쿨백-라이블러(KL) 발산을 사용해

티처랑 스튜던트 확률 분포의 차이를 측정해서 최소화하는 식~

▫ **Distillation Loss**

$$L_{KD} = T^2 D_{KL}$$

KL 발산을 사용해 티처의 확률 분포를 스튜던트로 근사할 때 손실되는 양을 계산하여

지식 정제 손실을 정의함

여기서 T^2은 소프트 레이블이 생성한 그래디언트의 크기가 $1/T^2$로 스케일 조정된다는 사실을 고려한 정규화 인자 ??

▫ **Student Loss**

$$L_{\text{student}} = \alpha L_{\text{CE}} + (1 - \alpha) L_{\text{KD}}$$

분류 작업에서 스튜던트 손실은 정답 레이블의 일반적인 크로스 엔트로피 손실 L_{CE}로

정제 손실을 가중 평균 한 것

- 여기서 a는 각 손실의 상대적인 강도를 제어하는 하이퍼파라미터

▫ 지식 정제 프로세스

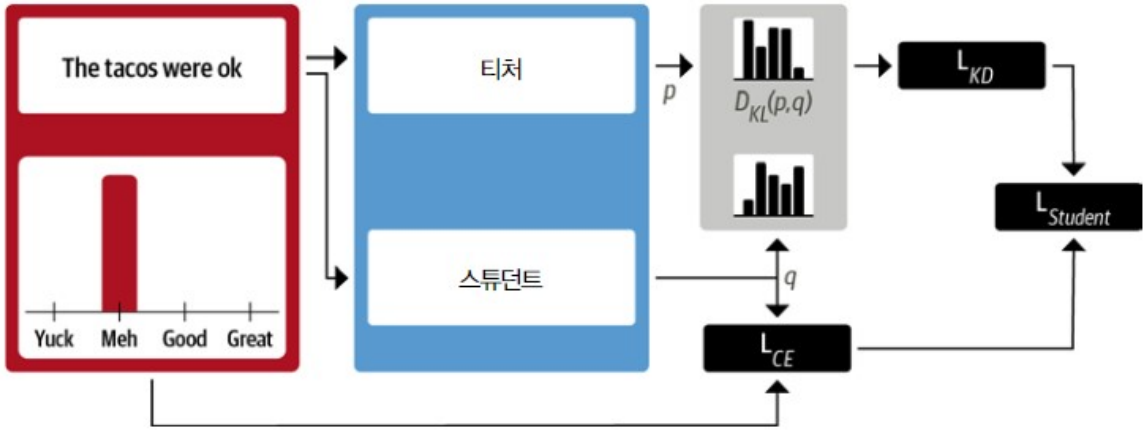
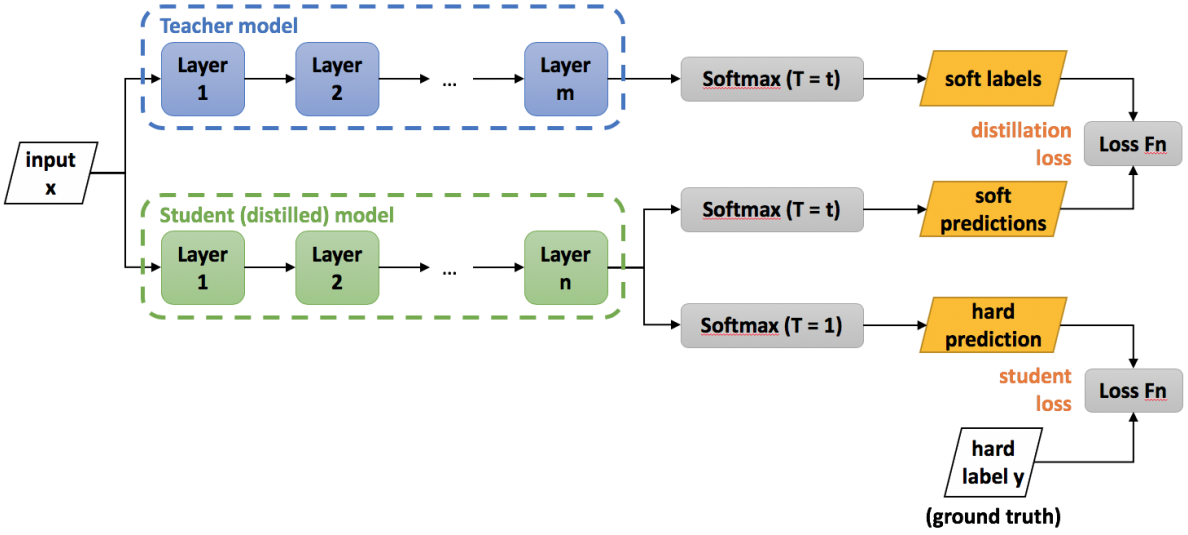


그림 8-4 지식 정제 프로세스



- 입력 문장을 바탕으로 teacher 네트워크가 반환하는 확률이 타깃이 되며, student 네트워크가

  반환하는 확률은 예측이 됨. 이러한 각각의 네트워크 출력은 소프트 타깃, 소프트 예측이 됨.

소프트 타깃과 소프트 예측 사이의 Cross-entropy loss 가 정제 손실(Distillation loss)이 됨.

- Soft prediction은 총합이 1이 되도록 softmax 그대로 사용하는 것이고
- Hard prediction은 softmax 이후 one-hot vector로 제일 큰 값만 1을 가지고 나머지는 0을

  가지도록 하는 것을 말함. Teacher Network를 더 잘 모방하도록 정제 손실의 경우 Soft label 사용

- 추론 시에는 표준 소프트맥스 확률을 얻기 위해 온도를 1로 지정함

8.3.2 사전 훈련에서의 지식 정제

사전 훈련하는 동안 후속 작업에서 미세 튜닝이 가능한 스튜던트를 만들기 위해
지식 정제를 사용할 수도 있음

- 이 경우 **티처**는 마스크드 언어 모델링의 지식을 **스튜던트**에 전달하는 BERT 같은 애들

- 예를 들면 DistilBERT 논문에서 마스크드 언어 모델링 손실 L_{mlm}은 **지식 정제 항**과 **코사인 임베딩 손실** $L_{cos} = 1 - cos(h_s, h_t)$로 보강됨

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}$$

▼ 무슨 소리냐면

1. DistilBERT의 핵심 아이디어는 작은 크기의 스튜던트 모델을 큰 티처 모델로부터
   지식을 전달받아 훈련시키는 것. 따라서 지식 정제 항이 손실 함수에 추가됨.

2. 코사인 임베딩 손실에서 *hs*는 스튜던트 모델의 은닉 상태 벡터이고,
   *ht*는 티처 모델의 은닉 상태 벡터. 코사인 임베딩 손실은 두 모델 간의
   은닉 상태 벡터가 유사한 방향을 가지도록 강제하여, 티처 모델로부터
   더 많은 지식을 스튜던트 모델로 전달할 수 있도록 도움.

결론은 DistilBERT에서는 MLM 손실에 **지식 정제 항**과 **코사인 임베딩 손실**을 추가하여
스튜던트 모델이 더 나은 성능을 얻을 수 있도록 한다는 말임 ~ ㅅ ~

8.3.3 지식 정제 트레이너 만들기

지식 정제를 구현하기 위해 Trainer 클래스에 몇 가지를 추가해야함

1. 새로운 하이퍼파라미터 a, T
    - a - 정제 손실의 상대적인 가중치를 제어
    - T - 레이블의 확률 분포를 얼마나 완만하게 만들지 조절함
2. 미세 튜닝한 티처 모델. 여기서는 BERT 베이스
3. 크로스 엔트로피와 지식 정제 손실을 연결한 새로운 손실 함수

▼ 새로운 하이퍼파라미터 추가

```
from transformers import TrainingArguments

class DistillationTrainingArguments(TrainingArguments):
    def __init__(self, *args, alpha=0.5, temperature=2.0,
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = temperature
```

`TrainingArguments` 클래스를 상속해 새로운 속성을 추가하면 끝

▼ 새로운 손실 함수

```
import torch.nn as nn
import torch.nn.functional as F
from transformers import Trainer

class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model
```

```
    def compute_loss(self, model, inputs, return_outputs=F
        device = torch.device("cuda" if torch.cuda.is_avai
        inputs = inputs.to(device)
        outputs_stu = model(**inputs)
        # 스튜던트의 크로스 엔트로피 손실과 로짓을 추출합니다
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # 티처의 로짓을 추출합니다
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits
        # 확률을 부드럽게하고 정제 손실을 계산합니다
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 * loss_fct(
            F.log_softmax(logits_stu / self.args.temperatu
            F.softmax(logits_tea / self.args.temperature,
        # 가중 평균된 스튜던트 손실을 반환합니다
        loss = self.args.alpha * loss_ce + (1. - self.args
        return (loss, outputs_stu) if return_outputs else
```

- `Trainer` 클래스를 상속하고 지식 정제 손실 항 L_{KD}을 추가하기 위해 `compute_loss()`

  메서드를 오버라이딩하면 구현 가능

- `DistillationTrainer`클래스의 객체를 만들 때 이 작업에서 이미 미세 튜닝된 티처를

  `teacher_model` 매개변수에 전달함

- 그 다음 `compute_loss()` 메서드에서 스튜던트와 티처의 로짓을 추출하고,

  온도로 스케일을 조정하고, 그 다음 소프트맥스로 정규화한 후

  파이토치 `nn.KLDivLoss()`함수에 전달해서 KL 발산을 계산

- `nn.KLDivLoss()` 에서 입력은 로그 확률, 레이블은 일반 확률로 기대한다는 점이 특이

- 이 때문에 `F.log_softmax()`를 사용해 스튜던트 로짓을 정규화하고 티처 로짓은 표준 소프트맥스 함수를 사용해 확률로 변환함

- `nn.KLDivLoss()`의 `reduction= “batchmean”` 매개변수는 배치 차원에서 손실을 평균함

트레이너 만들기 완

8.3.4 좋은 스튜던트 선택하기

- 보통은 레이턴시와 메모리 사용량을 줄이기 위해 **작은 모델**을 골라야 함
- 논문에 의하면 티처와 스튜던트가 동일한 종류의 모델일 때 지식 정제가 잘 동작함
  - 종류가 다르면 출력 임베딩 공간이 달라서 스튜던트가 티처를 모방하는데 방해 되기 때문

+) 여기서는 티처가 BERT이므로 DistilBERT가 자연스럽게 스튜던트 후보가 됨!

▼ 먼저 쿼리 토큰화 및 인코딩

```
from transformers import AutoTokenizer

# DistilBERT의 토크나이저 초기화
student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_

# 전처리를 수행할 tokenize_text() 함수 만들기
def tokenize_text(batch):
    return student_tokenizer(batch["text"], truncation=Tru

# text 열은 더 이상 필요하지 않기 때문에 삭제
```

```
clinc_enc = clinc.map(tokenize_text, batched=True, remove_
# 트레이너가 자동으로 감지하도록 intent열은 labels로 바꿈
clinc_enc = clinc_enc.rename_column("intent", "labels")
```

▼ 하이퍼파라미터와 `DistillationTrainer` 클래스를 위한 함수 정의

```
from huggingface_hub import notebook_login

notebook_login()
```

모델을 허깅페이스 허브에 저장하기 위해 로그인

```
# 훈련하는 동안 추적할 성능 지표 정의
def compute_metrics(pred):
    predictions, labels = pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy_score.compute(predictions=predictions,
```

- 성능 벤치마크와 동일하게 **정확도**를 주요 지표로 사용

- 이 함수에서 시퀀스 모델링 헤드가 출력한 예측은 로짓의 형태이기 때문에 `np.argmax()` 함수를 사용해 확률이 가장 높은 클래스를 찾고 정답 레이블과 비교

▼ a=1로 지정해 티처로부터 어떤 신호도 받지 않았을 때의 DistilBERT 성능 확인

```
batch_size = 48

finetuned_ckpt = "distilbert-base-uncased-finetuned-clinc"
student_training_args = DistillationTrainingArguments(
    output_dir=finetuned_ckpt, evaluation_strategy = "epoc
    num_train_epochs=5, learning_rate=2e-5,
```

```
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, alpha=1, weight
    push_to_hub=True)
```

- 이런 방식의 미세 튜닝하는 방식을 **task-agnostic**(종속되지 않는) 정제라고 함
- 에포크 횟수, 가중치 감쇠, 학습률 같은 하이퍼 파라미터의 기본값을 바꿨음

▼ 스튜던트 모델 만들기

```
# 스튜던트 모델에 의도와 레이블 ID의 매핑을 제공
id2label = pipe.model.config.id2label
label2id = pipe.model.config.label2id
```

```
# 사용자 정의 모델 설정 만들기
from transformers import AutoConfig

num_labels = intents.num_classes
student_config = (AutoConfig
                  .from_pretrained(student_ckpt, num_label
                                   id2label=id2label, labe
```

레이블 매핑에 관한 정보를 사용해 스튜던트를 위한 설정을 만들고

모델이 기대할 클래스 개수도 지정해줌

```
import torch
from transformers import AutoModelForSequenceClassificatio

device = torch.device("cuda" if torch.cuda.is_available()

def student_init():
    return (AutoModelForSequenceClassification
            .from_pretrained(student_ckpt, config=student_
```

위의 설정을 AutoModel~ 클래스의 `from_pretained()` 함수에 전달해서 초기화하는 함수 제작

Distillation 트레이너를 위한 요소 모두 준비 완

▼ 티처 로드 및 모델 훈련

```
# 티처 로드
teacher_ckpt = "transformersbook/bert-base-uncased-finetun
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_ckpt, num_labels=
                 .to(device))
```

```
# 스튜던트 모델 훈련
distilbert_trainer = DistillationTrainer(model_init=studen
    teacher_model=teacher_model, args=student_training_arg
    train_dataset=clinc_enc['train'], eval_dataset=clinc_e
    compute_metrics=compute_metrics, tokenizer=student_tok

distilbert_trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy
1	4.296700	3.281034	0.718065
2	2.614600	1.865238	0.840323
3	1.537800	1.147739	0.897742
4	1.004300	0.849075	0.913548
5	0.790200	0.768114	0.918387

약 92% 정확도 달성. 꽤괜.

▼ 모델 허브에 저장 및 성능 지표 계산

```
distilbert_trainer.push_to_hub("Training completed!")
```

```
finetuned_ckpt = "ssunho1/distilbert-base-uncased-finetune
pipe = pipeline("text-classification", model=finetuned_ckp
```

```
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=
perf_metrics.update(pb.run_benchmark())
```

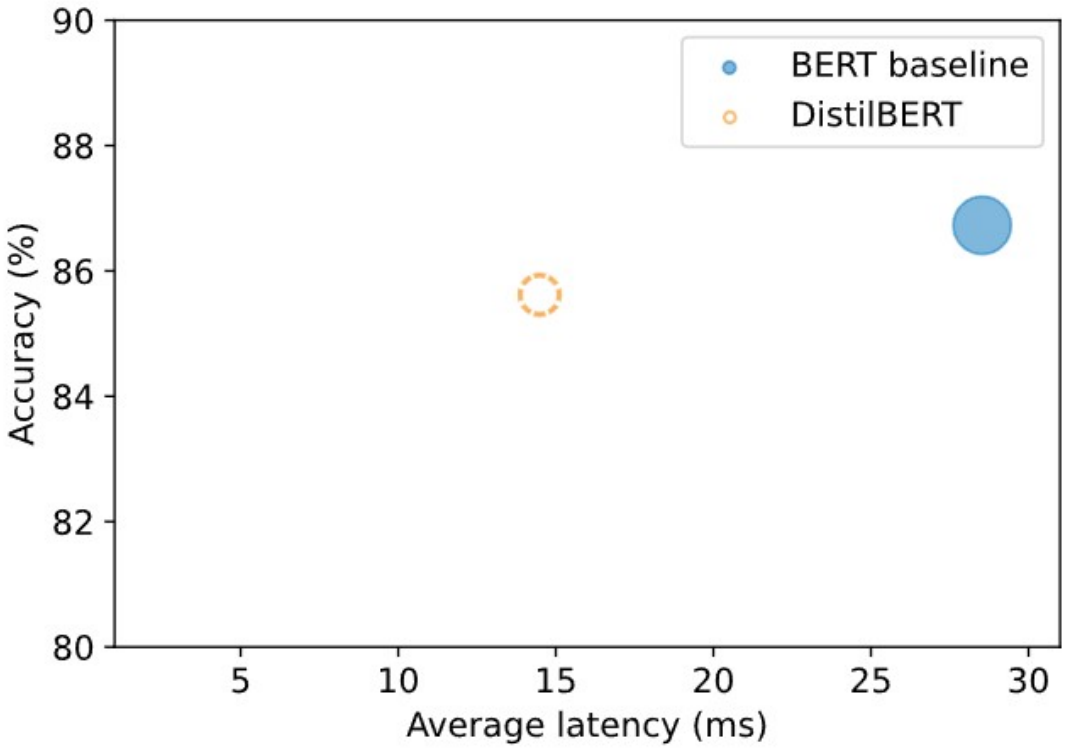
파이프 라인을 `PerformanceBenchmark` 클래스에 전달해 모델에 대한 성능 지표 계산

```
모델 크기 (MB) - 255.88
평균 레이턴시 (ms) - 14.44 +뭐- 0.90
테스트 세트 정확도 - 0.856
```

▼ 산점도를 이용해 기준 모델과 결과 비교

```
import pandas as pd

def plot_metrics(perf_metrics, current_optim_type):
    df = pd.DataFrame.from_dict(perf_metrics, orient='inde

    for idx in df.index:
        df_opt = df.loc[idx]
        # 현재 최적화 방법을 점선으로 그립니다
        if idx == current_optim_type:
            plt.scatter(df_opt["time_avg_ms"], df_opt["acc
                        alpha=0.5, s=df_opt["size_mb"], la
                        marker='$\u25CC$')
```

```
        else:
            plt.scatter(df_opt["time_avg_ms"], df_opt["acc
                        s=df_opt["size_mb"], label=idx, al

    legend = plt.legend(bbox_to_anchor=(1,1))
    for handle in legend.legend_handles:
        handle.set_sizes([20])

    plt.ylim(80,90)
    # 가장 느린 모델을 사용해 x 축 범위를 정합니다
    xlim = int(perf_metrics["BERT baseline"]["time_avg_ms"
    plt.xlim(1, xlim)
    plt.ylabel("Accuracy (%)")
    plt.xlabel("Average latency (ms)")
    plt.show()

plot_metrics(perf_metrics, optim_type)
```



- 각 포인트의 반지름은 모델의 크기에 해당
- 정확도는 1% 줄었지만 작은 모델을 사용해 평균 레이턴시가 크게 준 것을 확인 가능

8.3.5 옵투나로 좋은 하이퍼파라미터 찾기

옵투나는 검색 문제를 여러 시도를 통해 최적화할 목적 함수로 표현해주는 최적화 프레임워크

▫ 예시

로젠브록의 바나나 함수를 최소화한다고 가정하자

(함수의 이름은 아래 사진의 휘어진 등고선에서 따왔음)

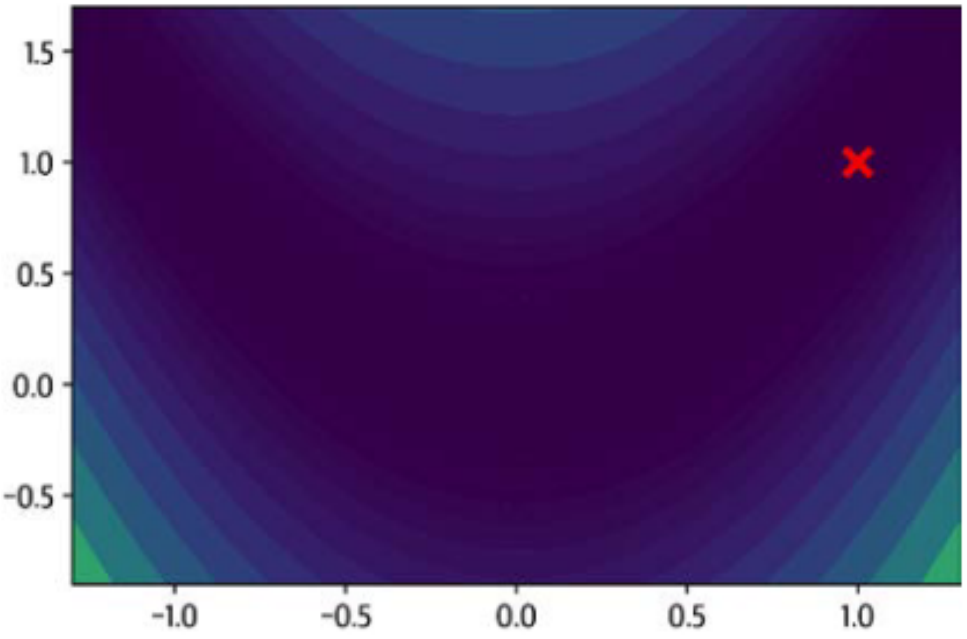


그림 8-5 두 변수를 가진 로젠브록 함수 그래프

$$f(x,\ y) = (1 - x)^2 + 100\,(y - x^2)^2$$

이 함수는 $(x, y) = (1, 1)$에서 전역 최솟값을 가지는데 골짜기 영역을 찾는 것은

어렵지 않은 최적화 문제이나 전역 최솟값으로 **수렴**하기는 어려움

- 옵투나에서는 $f(x, y)$의 값을 반환하는 `objective()` 함수를 정의해 **최솟값**을 찾음

```
def objective(trial):
    x = trial.suggest_float("x", -2, 2)
    y = trial.suggest_float("y", -2, 2)
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
```

`trial.suggest_float()` : 샘플링할 파라미터 범위를 지정

정수형을 위한 suggest_int나 범주형을 위한 suggest_categorical도 제공

▼ 옵투나는 여러 시도를 하나의 스터디로 수집

```
import optuna

study = optuna.create_study()
study.optimize(objective, n_trials=1000)
```

- `objective()` 함수를 `study.optimize()` 메서드에 전달

```
study.best_params
```

스터디가 끝나면 최상의 파라미터 겟또

```
{'x': 1.0884186262922515, 'y': 1.1894112108107937}
```

전역 최솟값이랑 상당히 유사!

🤗 트랜스포머스에서 옵투나 사용하기

▼ 먼저 최적화하려는 하이퍼파라미터 공간 정의

```
def hp_space(trial):
    return {"num_train_epochs": trial.suggest_int("num_tra
        "alpha": trial.suggest_float("alpha", 0, 1),
        "temperature": trial.suggest_int("temperature", 2,
```

a, T, 훈련 에포크 횟수 포함시킴

▼ Trainer로 하이퍼파라미터 검색 실행

```
best_run = distilbert_trainer.hyperparameter_search(
    n_trials=20, direction="maximize", hp_space=hp_space)
```

- 트레이너의 `hyperparameter_search()` 메서드에 시도 횟수와 최적화 방향을 지정하고

  하이퍼 파라미터 검색 공간을 전달하면

```
print(best_run)
```

```
BestRun(run_id='10', objective=0.9293548387096774,
hyperparameters={'num_train_epochs': 10, 'alpha': 0.01626185622974391,
'temperature': 2}, run_summary=None)
```

- `hyperparameter_search()` 메서드가 최대화된 목적 함수의 값과 해당 시도에 사용한 하이퍼파라미터를 담은 BestRun 객체를 반환

▼ 위 값으로 훈련 매개변수를 수정하고 최종 훈련 ㄱㄱ

```
for k,v in best_run.hyperparameters.items():
    setattr(student_training_args, k, v)

# 정제된 모델을 저장할 새로운 저장소를 정의합니다
distilled_ckpt = "distilbert-base-uncased-distilled-clinc"
student_training_args.output_dir = distilled_ckpt

# 최적의 매개변수로 새로운 Trainer를 만듭니다
distil_trainer = DistillationTrainer(model_init=student_in
    teacher_model=teacher_model, args=student_training_arg
    train_dataset=clinc_enc['train'], eval_dataset=clinc_e
    compute_metrics=compute_metrics, tokenizer=student_tok

distil_trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy
1	0.857600	0.451161	0.679032
2	0.340700	0.165502	0.844194
3	0.161100	0.089016	0.905806
4	0.104600	0.066544	0.920968
5	0.083100	0.057513	0.925484
6	0.072700	0.052300	0.931290
7	0.066400	0.049449	0.928710
8	0.062500	0.047550	0.931290
9	0.060300	0.046273	0.930968
10	0.058900	0.046025	0.931935

스튜던트의 매개변수 개수는 티처의 절반이지만 티처의 정확도에 버금가는 것을 확인 가능

▼ 모델 허브에 업로드

```
distil_trainer.push_to_hub("Training complete")
```

정확도 높은 스튜던트 모델 제작 완

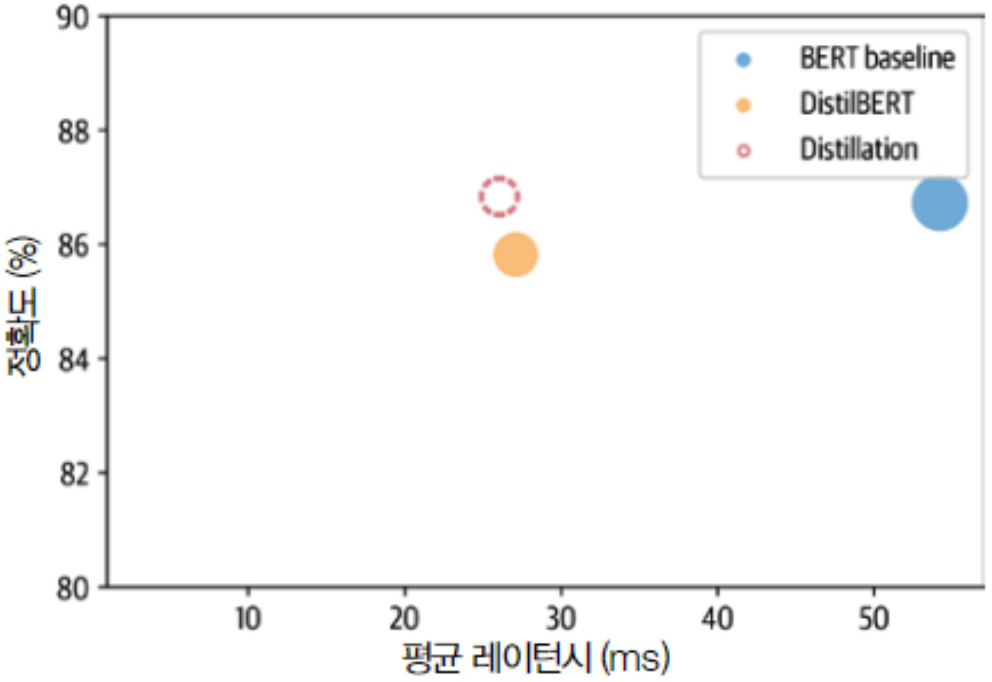
8.3.6 정제 모델 벤치마크 수행하기

▼ 파이프라인 만들고 벤치마크 다시 수행해 테스트 셋에서 성능 확인하기

```
# 성능 지표 계산
distilled_ckpt = "ssunho1/distilbert-base-uncased-distille
pipe = pipeline("text-classification", model=distilled_ckp
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=
perf_metrics.update(pb.run_benchmark())
```

```
모델 크기 (MB) - 255.88
평균 레이턴시 (ms) - 14.48 +₩- 0.86
테스트 세트 정확도 - 0.857
```

```
# 결과 비교를 위한 시각화
plot_metrics(perf_metrics, optim_type)
```



- 모델 크기와 레이턴시는 기본적으로 DistilBERT랑 유사
- 정확도 향상 + 티처보다 좋은 성능!

  >> 이는 티처가 스튜던트처럼 체계적으로 미세 튜닝되지 않았을 가능성이 있음
- 추후에 **양자화** 기술을 사용하면 모델을 더 압축 가능 >> 다음 시간에