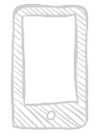


Text Classification

트랜스포머

Team



CONTENTS

Ⅰ. 프로젝트 개요

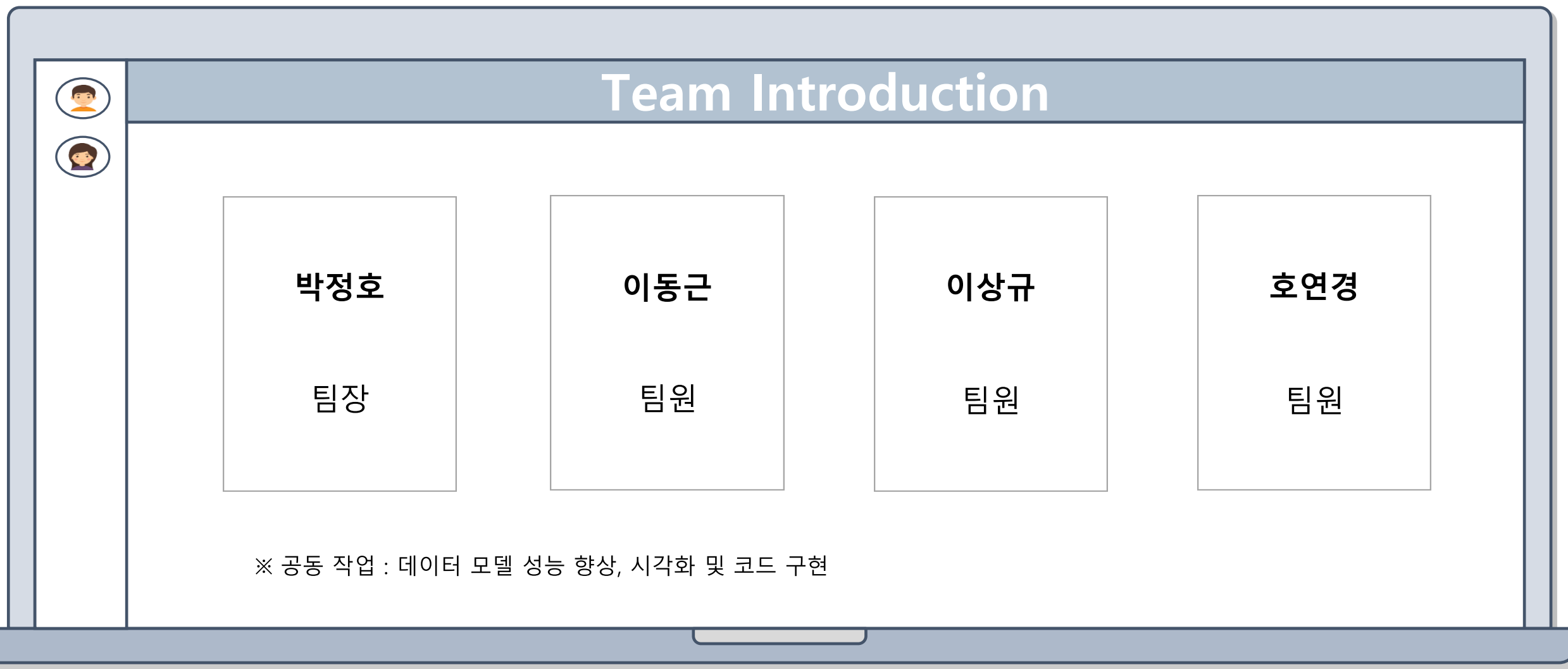
Ⅱ. 팀 소개

Ⅲ. 데이터 분석 및 결과

Ⅳ. 프로젝트 수행 느낀점



Ⅴ. Q&A

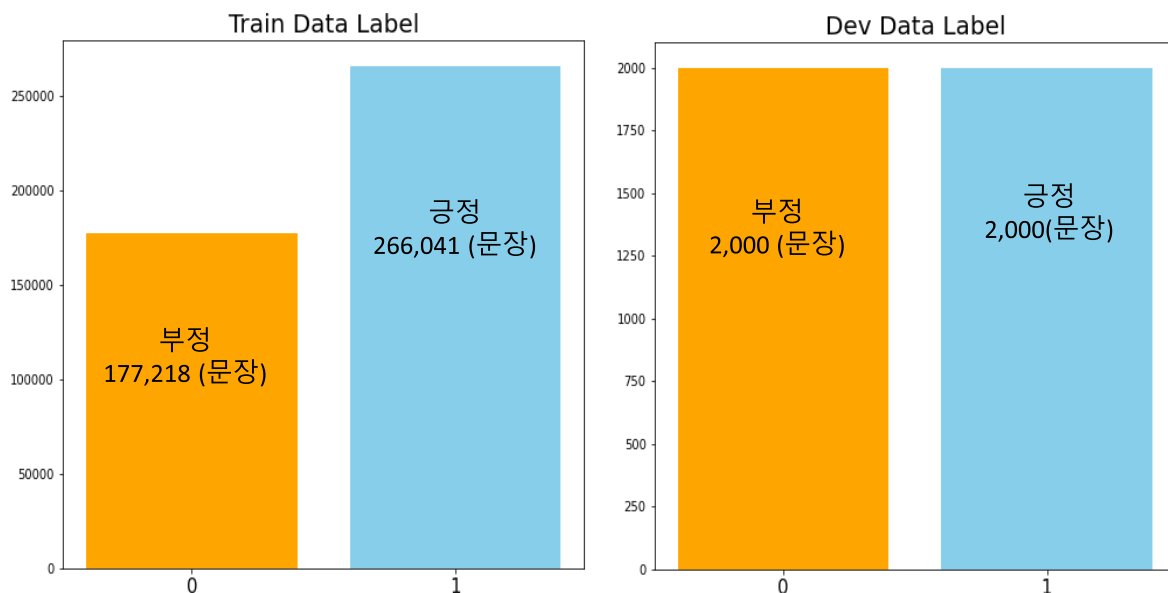
II. 팀 소개



I. 프로젝트 개요

01. 데이터 소개 및 탐색

1. 주제: 영어 문장에 대한 긍정/부정 리뷰 분류
2. 목표
 - ✓ Learning Rate, Batch_size, epoch 조정하여 학습 훈련
 - ✓ 모델 성능을 높이기 위한 파라미터 조정 및 스케줄러 적용
 - ✓ 모델을 도출하여 Accuracy 높이기
3. 활용 라이브러리 및 프레임워크:  
4. 데이터 설명
 - ✓ 학습 데이터: 443,259(문장) / 평가 데이터: 4,000(문장)



긍정 데이터 워드클라우드 결과



- ✓ 긍정 데이터 핵심 단어 <great, good, food, service> 등

부정 데이터 워드클라우드 결과



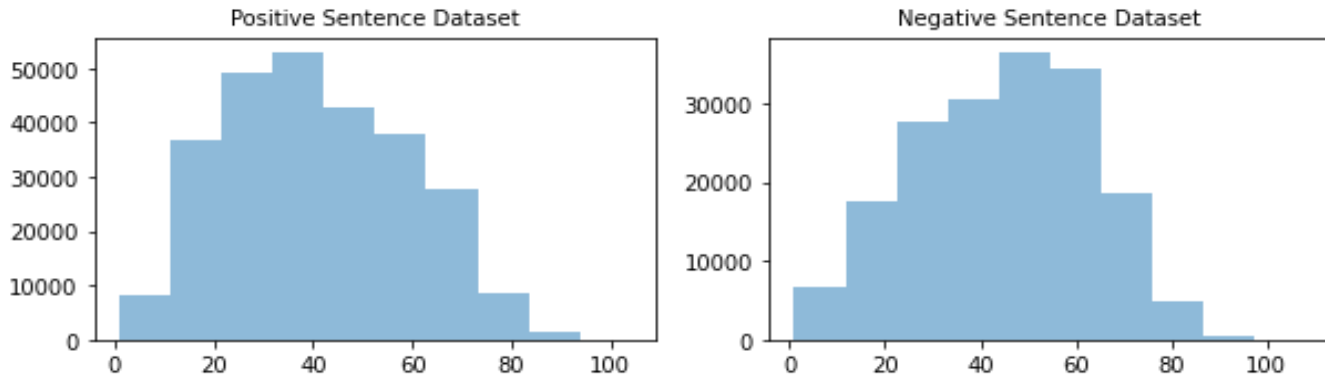
- ✓ 부정 데이터 핵심 단어 <num, service, never, bad> 등

II. 프로젝트 개요

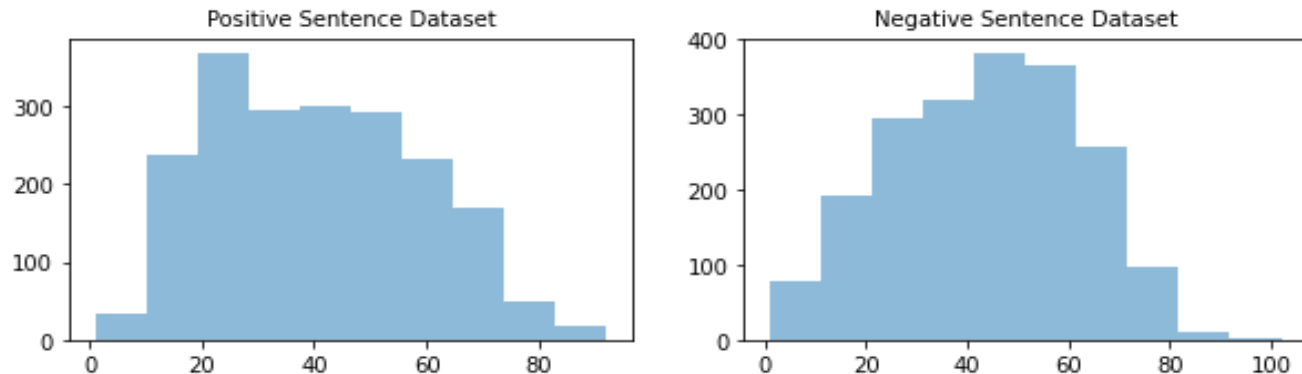
01. 데이터 소개 및 탐색

데이터 단어 길이 확인

Train Dataset Sentence Length



Dev Dataset Sentence Length



학습데이터

평가데이터

평균문장 길이	42.25 단어	42.13 단어
최대문장 길이	108 단어 (긍정문: 104 단어) (부정문: 108 단어)	102 단어 (긍정문: 92 단어) (부정문: 102 단어)
문장의 분포	대체로 30 ~ 50 단어 길이의 문장이 많이 분포함	대체로 30 ~ 50 단어 길이의 문장이 많이 분포함

Ⅲ. 데이터 분석 및 결과

01. 전처리 결과 비교

전처리 및 Tokenization

baseline 전처리 코드

```
def collate_fn_sentiment(samples):
    input_ids, labels = zip(*samples)
    max_len = max(len(input_id) for input_id in input_ids)
    sorted_indices = np.argsort([len(input_id) for input_id in input_ids])[::-1]

    attention_mask = torch.tensor(
        [[1] * len(input_ids[index]) + [0] * (max_len - len(input_ids[index])) for index in sorted_indices])
    input_ids = pad_sequence([torch.tensor(input_ids[index]) for index in sorted_indices],
                             batch_first=True)
    token_type_ids = torch.tensor([[0] * len(input_ids[index]) for index in sorted_indices])
    position_ids = torch.tensor([list(range(len(input_ids[index]))) for index in sorted_indices])
    labels = torch.tensor(np.stack(labels, axis=0)[sorted_indices])

    return input_ids, attention_mask, token_type_ids, position_ids, labels
```

결과값	(array([101, 6581, 2833, 1012, 102]), array([1]))
특징	collate_fn함수를 활용함으로써 코드를 보다 자유롭게 customise할 수 있는 자유도가 높아짐

최종 전처리 코드

```
def preprocess(text_data, label_data, batch_size=8):
    batch_input = tokenizer(text_data, truncation=True, padding=True)

    batch_input = {key : torch.tensor(value) for key, value in batch_input.items()}

    label = torch.tensor(label_data)

    dataset = TensorDataset(batch_input['input_ids'],
                             batch_input['token_type_ids'],
                             batch_input['attention_mask'], label)
    dataset_sampler = RandomSampler(dataset)
    dataset = DataLoader(dataset, sampler = dataset_sampler, batch_size= batch_size)

    return dataset
```

결과값	{'attention_mask': tensor([[1, 1, 1, ..., 0, 0, 0], 'input_ids': tensor([[101, 6581, 2833, ..., 0, 0, 0], 'token_type_ids': tensor([[0, 0, 0, ..., 0, 0, 0],
특징	BertTokenizer 대비 1.5배 속도가 빠른 BertTokenizer Fast를 사용 전처리 과정에서 tokenizer의 padding과 truncation등의 API를 활용하여 collate_fn을 사용하지 않아 간편함

Ⅲ. 데이터 분석 및 결과

02. 모델 정확도 비교

```

train_epoch = 4
lowest_valid_loss = 9999,

param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'gamma', 'beta']
optimizer_grouped_parameters = [
    {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.01},
    {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.0}
]

optimizer = AdamW(optimizer_grouped_parameters, lr=1e-5, eps=1e-8)

total_steps = len(train_loader) * train_epoch

scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=total_steps)

for epoch in range(train_epoch):
    with tqdm(train_loader, unit="batch") as tepoch:
        train_predictions = []
        train_target_labels = []
        for iteration, batches in enumerate(tepoch):
            tepoch.set_description(f"Epoch {epoch}")
            batch = tuple(d.to(device) for d in batches)

            input = {'input_ids' : batch[0],
                     'token_type_ids' : batch[1],
                     'attention_mask' : batch[2],
                     'labels' : batch[3]}

            optimizer.zero_grad()

            output = model(**input)

            logits = output.logits
            loss = output.loss
            loss.backward()

            optimizer.step()
            scheduler.step()

            batch_predictions = [0 if example[0] > example[1] else 1 for example in logits]
            batch_labels = [int(example) for example in input['labels']]

            train_predictions += batch_predictions
            train_target_labels += batch_labels

```

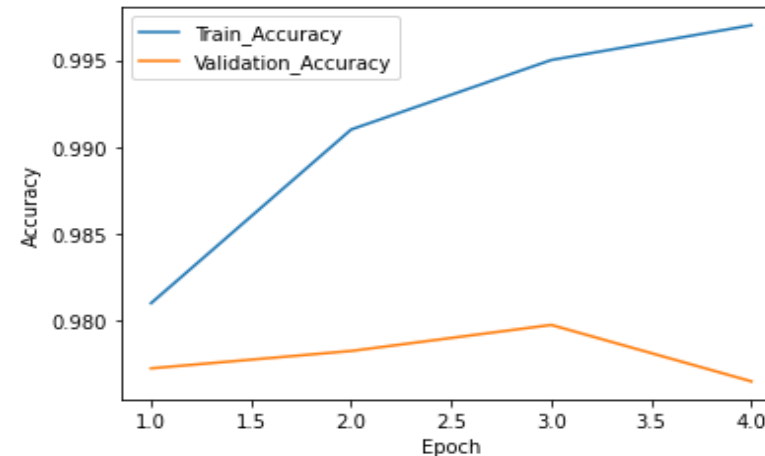
모델 성능을 높이기 위해 적용한 방법!

Scheduler

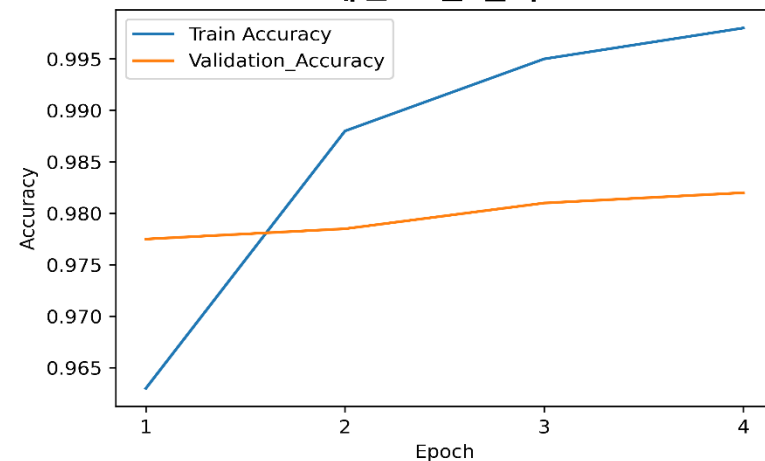
- 학습을 통해 효율적인 러닝레이트 조정
- 학습률을 미세조정 하여 높은 정확도에 도달할 수 있도록 가중치 설정



Baseline 모델 결과



개선 모델 결과



Ⅲ. 데이터 분석 및 결과

03. 모델 성능 개선

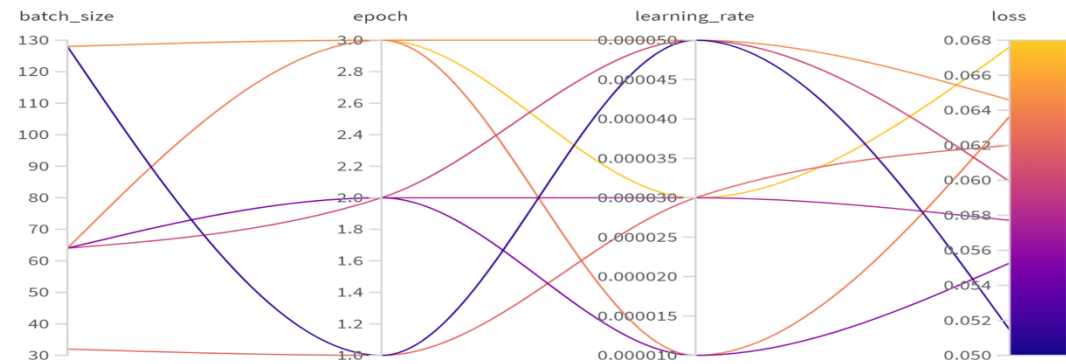
Baseline Code 모델 학습 결과

Hyper Parameters	Model #1	Model #2	Model #3	Model #4	Model #5
Learning Rate	1e-5	3e-5	3e-5	1e-5	2e-5
Batch Size	64	64	64	64	64
Epoch	1	3	4	3	3
Accuracy	98%	98.8%	98.3%	98.4%	98.7%

1. Model #2에서는 Learning Rate을 3e-5, Epoch을 3으로 증가
→ Model #1 대비 0.8% 성능 향상됨
2. 성능향상을 위해 같은 조건에서 Epoch수만 4로 설정
→ 오히려 Model #2 대비 0.5%의 성능 저하가 발생
3. Model #4는 Learning Rate을 1e-5로, Epoch수를 3으로 설정
→ Model #3 대비 0.1% 성능 향상됨
4. Model #5는 Learning Rate을 2e-5로 설정
→ Model #4 대비 0.3% 성능 향상됨

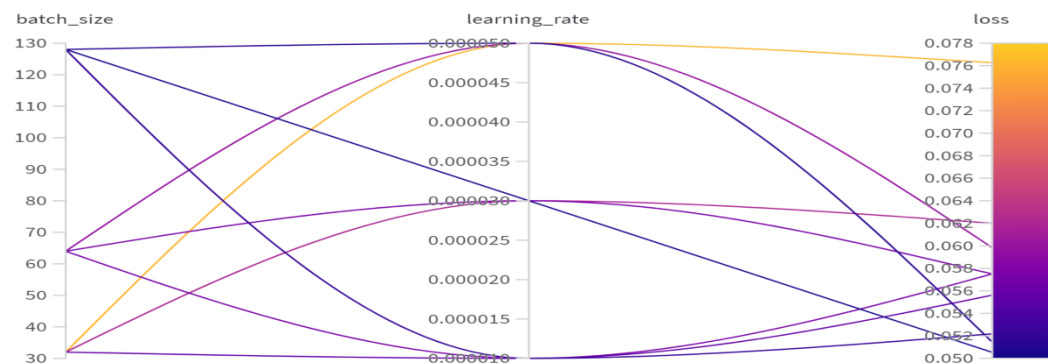
※ 최종 모델

모델 파라미터 bias를 고정시키고 스케줄러를 적용해 효율적으로 러닝레이트를 조정한 결과 캐글 점수 0.989점까지 상승시킴



method : **random**

- ✓ batch_size, epoch, learning_rate 3종류를 각각 3개의 값으로 돌림
- ✓ $3 \times 3 \times 3 = 27$ 이지만, method가 random이기 때문에 몇몇 경우만 실행됨
- ✓ 총 실행시간 8시간 소요, 성능 향상에 미흡



method : **grid**

- ✓ batch_size와, learning_rate 2종류만 3×3 돌림
- ✓ batch_size=32, learning_rate=5e-5일때 오차가 유독 큰 모습을 보임

IV. 프로젝트 수행 느낀점

◆ 로깅의 중요성

실험 결과의 기록을 체계적으로 하지 못하여 모델 분석에 어려움이 있었음
이후에는 각 진행 과정 및 결과를 상세히 로깅할 필요성을 느낌

◆ 체계적인 역할 분담의 필요성

모델을 돌리는데 시간이 많이 소모되어, 추후에는 제한된 자원내에서 보다 효율적인 시간 관리 및 실험을
하기 위해 보다 체계적인 역할분담이 필요함

◆ 프로젝트 관리

프로젝트 진행 시에 주어진 일정에 따라 좀 더 명확한 단계를 설정해야 할 필요성을 느낌
각 단계를 마칠 때마다 꾸준히 취합을 거치면 더 좋을 듯함

