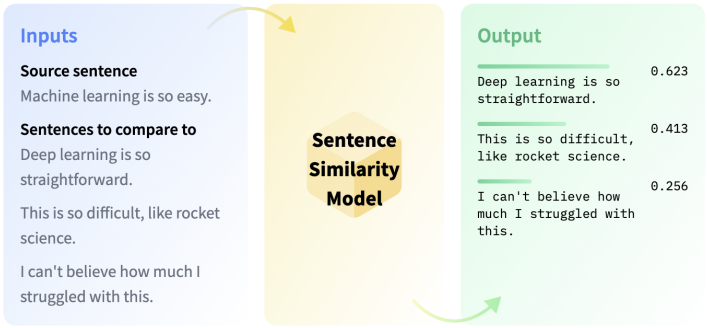
NLP-01조 Wrap up 리포트(업로드용)

▼ Task 소개 및 그라운드 룰

task 소개

- Sematic Text Similarity(문맥적 유사도 판단)



- 평가 방법
  - 문장 A, B를 input으로 넣으면 각 문장의 상관 정도에 따라 0 ~ 5의 값을 출력
  - 전체 출력 결과와 전체 Label의 피어슨 상관 계수로 모델의 성능을 측정

$$\text{피어슨 상관계수} = \frac{\text{공분산}}{\text{표준편차}_{*}\text{표준편차}}$$

- 데이터셋 (이미지 : train dataset head 10)

id	source	sentence_1	sentence_2	label	binary-label
boostcamp-sts-v1-train-000	nsmc-sampled	스릴도있고 반전도 있고 여느 한국영화 쓰레기들하고는 차원이 다르네요~	반전도 있고,사랑도 있고재미도있네요.	2.2	0.0
boostcamp-sts-v1-train-001	slack-rtt	앗 제가 접근권한이 없다고 뜹니다;;	오, 액세스 권한이 없다고 합니다.	4.2	1.0
boostcamp-sts-v1-train-002	petition-sampled	주택청약조건 변경해주세요.	주택청약 무주택기준 변경해주세요.	2.4	0.0
boostcamp-sts-v1-train-003	slack-sampled	입사후 처음 대면으로 만나 반가웠습니다.	화상으로만 보다가 리얼로 만나니 정말 반가웠습니다.	3.0	1.0
boostcamp-sts-v1-train-004	slack-sampled	뿌듯뿌듯 하네요!!	꼬옥 실제로 한번 뵈어요 뿌뿌뿌~!~!	0.0	0.0
boostcamp-sts-v1-train-005	nsmc-rtt	오마이가뜨지저스크라이스트퐛	오 마이 갓 지저스 스크론 이스트 퐨	2.6	1.0
boostcamp-sts-v1-train-006	slack-rtt	전 암만 찍어도 까만 하늘.. ㅠㅠ	암만 찍어도 하늘은 까맣다.. ㅠㅠ	3.6	1.0
boostcamp-sts-v1-train-007	nsmc-sampled	이렇게 귀여운 쥐들은 처음이네요.ㅎㅎㅎ	이렇게 지겨운 공포영화는 처음..	0.6	0.0
boostcamp-sts-v1-train-008	petition-sampled	미세먼지 해결이 가장 시급한 문제입니다!	가장 시급한 것이 신생아실 관리입니다!!!	0.4	0.0
boostcamp-sts-v1-train-009	petition-sampled	크림하우스 환불조치해주세요.	크림하우스 환불조치할 수 있도록해주세여	4.2	1.0
boostcamp-sts-v1-train-010	slack-rtt	그 책부터 언능 꺼내봐야 겠어요!	책에서 꺼내야겠어요!	2.4	0.0

- 총 데이터 개수 : 10,974 문장 쌍 (train/dev/test, 85/5/10)
  - Train 데이터 개수 : 9,324
  - Test 데이터 개수 : 1,100 (테스트는 label, binary-label column 미포함)
  - Dev 데이터 개수 : 550

- 데이터 구성
  - petition (국민청원 게시판 제목 데이터)
  - NSMC (네이버 영화 감성 분석 코퍼스)
  - slack (업스테이지(Ustage) 슬랙 데이터)

- Label 점수: 0 ~ 5사이의 실수
  - 5점 : 두 문장의 핵심 내용이 동일하며, 부가적인 내용들도 동일함
  - 4점 : 두 문장의 핵심 내용이 동등하며, 부가적인 내용에서는 미미한 차이가 있음
  - 3점 : 두 문장의 핵심 내용은 대략적으로 동등하지만, 부가적인 내용에 무시하기 어려운 차이가 있음
  - 2점 : 두 문장의 핵심 내용은 동등하지 않지만, 몇 가지 부가적인 내용을 공유함
  - 1점 : 두 문장의 핵심 내용은 동등하지 않지만, 비슷한 주제를 다루고 있음
  - 0점 : 두 문장의 핵심 내용이 동등하지 않고, 부가적인 내용에서도 공통점이 없음

팀 구성 및 역할

- 이상문(PM) : 프로젝트 관리 및 전체적인 방향 설정, 모델 결과 모니터링 및 앙상블 담당
- 김해원(Research) : Task에 적용가능한 기법 리서치 담당, 다양한 Optimizer, Loss function 실험
- 신혜진(Data) : EDA 및 Data agumetation 담당, 모델 출력 결과 분석
- 임성근(Code reviewer) : 코드 리뷰 및 베이스라인 코드 커스터마이징, AutoML 환경(Sweep) 구축 담당
- 양봉석(Code reviewer) : 팀 리포지토리 코드 리뷰 및 다양한 모델 실험 담당

수행 절차

- 팀 리포지토리는 git flow 규칙에 따라 관리 (branch 생성해서 본인 담당 코드 구현 및 풀 리퀘 요청, 코드 리뷰어가 머지 담당, 머지 후 브랜치 삭제)
- 커밋 메시지 알아보기 쉽게 룰에 따라 작성 (https://blog.ull.im/engineering/2019/03/10/logs-on-git.html)
- zoom 필수 접속 시간 외에 gather, slack을 이용하여 자유롭게 소통
- 팀 wandb project 생성 및 실시간으로 서로의 실험 결과를 모니터링
- notion의 각자의 실험결과를 정리하여 공유하기
- Timeline

▼ 코드 변경 및 AutoML 환경 구축 내용

- Dataloader에서 model_name, batch_size, shuffle, train_path, dev_path, test_path, predict_path의 파라미터를 넘겨받는 부분을 config파일 하나를 받아오는 것으로 변환.

- tokenizer에서 max_length를 직접 숫자로 입력해줘야 정상적으로 작동한다는 것을 확인하여 수정

```
outputs = self.tokenizer(text, add_special_tokens=True, max_length=128, padding='max_length', truncation=True)
```

- 전처리를 위한 preprocessing.py 추가, 전처리 시행시 걸린 시간을 측정하여 출력.

```
def preprocessing(self, sentence):

    # remove emojis
    sentence = core.replace_emoji(sentence, replace='')

    # spacing # tensorflow vs lightning 버전 충돌 문제로 잠시 보류.
    # spaced = self.spacer.space([sentence])[0]

    # spell_check
    try:
        spell_checked = spell_checker.check(sentence).as_dict()['checked']
    except:
        # 에러 발생시 맞춤법 교정은 생략하고 다음 단계로 넘김.
        spell_checked = sentence

    # emoticon_normalize가 'ㅋㅋㅋ' 연쇄 앞뒤의 음절을 지우지 않도록 해당 연쇄 앞뒤에 공백 추가
    p = re.compile('[ㄱ-ㅎ]{2,}')
    pattern_list = p.findall(spell_checked)
    if pattern_list:
        for pattern in pattern_list:
            spell_checked = spell_checked.replace(pattern, ' ' + pattern + ' ')

    # '맞앜ㅋㅋ' -> '맞아 ㅋㅋㅋ' 와 같이 정규화합니다.
    normalized = emoticon_normalize(spell_checked, num_repeats=2)

    return normalized.strip()
```

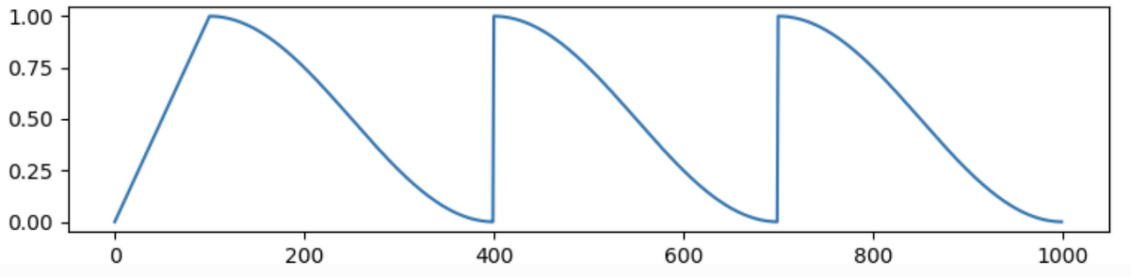
- hanspell과 emoji를 이용하여 맞춤법 교정, 정규화를 시행하였다.
- 똑같은 결과를 구현하기 위해 모든 seed를 고정

- setup 함수에서는 k_fold를 적용할때는 train + dev를 합쳐서 k만큼 잘라서 다시 만든 train과 dev 데이터셋으로 분리

- model에서는 model_name과 lr를 config파일 에서 받아오는것으로 수정
- transformer내부의 dropout과 learning rate warmup 구현

- 기존의 L1 loss대신 SmoothL1을 사용할수 있도록 구현

- training_step에 Rdrop 구현
- transformer의 get_cosine_with_hard_restarts_schedule_with_warmup함수 사용



- main함수에서는 기존의 args를 다 수정하여 config파일을 골라서 받아오는 형식으로 수정

```
parser = argparse.ArgumentParser()
parser.add_argument('--model_name', default='klue/roberta-small', type=str)
parser.add_argument('--batch_size', default=16, type=int)
parser.add_argument('--max_epoch', default=1, type=int)
parser.add_argument('--shuffle', default=True)
parser.add_argument('--learning_rate', default=1e-5, type=float)
parser.add_argument('--train_path', default='train.csv')
parser.add_argument('--dev_path', default='dev.csv')
parser.add_argument('--test_path', default='dev.csv')
parser.add_argument('--predict_path', default='test.csv')
args = parser.parse_args(args=[])
```

  - 원래 코드

```
parser = argparse.ArgumentParser()
parser.add_argument('--config', type=str, default='')
args, _ = parser.parse_known_args()
cfg = OmegaConf.load(f'./config/{args.config}.yaml')
```

  - 수정 후

```
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning.callbacks import LearningRateMonitor
from pytorch_lightning.callbacks import ModelCheckpoint
```

- pytorch lightning의 모듈을 이용하여 log기록, LR모니터링, 최고성능의 모델만 저장하는 checkpoint 구현

```
# wandb logger
wandb_logger = WandbLogger(name=f'{cfg.model.saved_name}_{str(k)}th_fold', project=cfg.repo.project_name)
wandb.watch(model)
```

```
lr_monitor = LearningRateMonitor(logging_interval='step')
```

```
checkpoint_callback = ModelCheckpoint(dirpath="models/",
                                      filename=f'{cfg.model.saved_name}_{str(k)}th_fold',
                                      save_top_k=1,
                                      monitor="val_pearson",
                                      mode='max')
```

- K-fold적용 유무에 따라 구현
  - cfg.train.k_fold일 경우 k번의 모델을 train하여 나온 결과 값을 모두 더해 k로 나눠 mean값을 취한다.

- utils.py와 load.py
  - config에서 받아와 optimizer_selector를 통해 원하는 optimizer로 학습 가능하게 선택
  - Adam, AdamW, RMSprop, NAdam, RAdam

- prediction_analysis.py
  - inference.py의 결과값으로 나온 csv를 분석하여 0점부터 1점단위로 group화
  - 0: 0~1, 1: 1~2, 2: 2~3, 3: 3~4, 4: 4~5
  - 그룹화 하여 pearson상관계수 계산하여 각 label별로 점수를 확인 → 해당 모델이 어느 점수대 예측이 용이한지 확인할 수 있음

- install.sh
  - 버전을 통일하기위해 작성된 requirements.txt의 설치 진행, 맞춤법 검사기인 hanspell은 직접 설치가 불가능한 이슈가 있어 github에서 다운로드 후 설치하도록 하였음.

- inference.py

```
if not cfg.inference.weighted_ensemble: # soft voting
    length = len(output)

    # make void tensor to store each model's predictions
    tmp_sum = torch.zeros((length,),dtype=torch.float32)

    for each in cfg.inference.ensemble:

        trainer = pl.Trainer(gpus=cfg.train.gpus, max_epochs=cfg.train.max_epoch, log_every_n_steps=cfg.train.logging_step)

        # Inference part
        if each.endswith('.ckpt'):
            model = Model.load_from_checkpoint(checkpoint_path=f'models/{each}')
        else:
            model = torch.load('models/' + each)
        each_pred = trainer.predict(model=model, datamodule=dataloader)
        each_pred = torch.cat(each_pred)
        tmp_sum += each_pred

    # divide total_sum by the number of models
    tmp_sum = tmp_sum / len(cfg.inference.ensemble)
    predictions = list(round(float(i), 1) for i in tmp_sum)

    output['target'] = predictions
    output.to_csv('output.csv', index=False)

else: #Weighted voting ensemble
    trainer = pl.Trainer(gpus=cfg.train.gpus, max_epochs=cfg.train.max_epoch, log_every_n_steps=cfg.train.logging_step)
    weights = cfg.inference.weighted_ensemble
    vote_predictions = weighted_voting(cfg.inference.ensemble, weights, trainer, dataloader)
    output['target'] = vote_predictions
    output.to_csv('output.csv', index=False)
```

Grouping by label

- 0~1: 0, 1~2: 1, 2~3: 2, 3~4: 3, 4~5: 4

- 여러 모델을 ensemble, soft와 weighted voting 두가지 방법으로 구현
- soft는 각 class별로 모델이 예측한 값을 합산해서 가장 높은 출력
- weighted는 각각의 모델별로 가중치(pearson계수가 높은 모델에 더 가중치를 주었음)

- base_config.yaml
  - path, data, model_name, train파라미터, repo 등 세팅 총망라

```
path:
    train_path: ../data/preprocessed/train_swap_preprocessed.csv
    dev_path: ../data/preprocessed/dev_preprocessed.csv
    test_path: ../data/preprocessed/dev_preprocessed.csv
    predict_path: ../data/test_preprocessed.csv
```

```
data:
    shuffle: True
    augmentation: # adea, bt 등등
    use_prepro : False
    max_length : 128

model:
    model_name: monologg/koelectra-base-v3-discriminator # (Required)
    saved_name: ko-electra # (Required)

train:
    seed: 2022
    gpus: 1
    batch_size: 32
    max_epoch: 5
    learning_rate: 1e-5
    logging_step: 1
    drop_out: 0.2
    precision: 32 # [32(default), 16]
    k_fold : 0 # default : 0 => not using
    warmup_ratio : 0.01 # default : 0
    loss_function : torch.nn.SmoothL1Loss
    smart_loss: True # True or Fasle
    optimizer : Adam # [Adam, AdamW, RMSprop, NAdam, RAdam]
    R_drop : True
    R_drop_alpha : 1 # default : 1

inference:
    ensemble : # List or False #
    weighted_ensemble : # List or False
repo:
  entity: nlp_level1_team1_2
  project_name: SG_test # (Required value) Need to specify your project name
```

- sweep를 활용한 automl 구축 내용

```
sweep_config = {
    'method': 'random', # random: 임의의 값의 parameter 세트를 선택
                        #'grid' 하이퍼파라미터 모두다
                        #'bayes' 좀더 확인
    'project': 'SG_test',
    'entity': 'nlp_level1_team1',
    'name' : 'monologg/koelectra-base-v3-discriminator',
    'metric': {
            'name':'val_pearson',
            'goal':'maximize'
    },
    'parameters': {
        'lr':{
            'distribution': 'uniform',  # parameter를 설정하는 기준을 선택합니다. uniform은 연속적으로 균등한 값들을 선택합니다.
           #'values': [1e-5, 1e-4, 2e-4, 2e-5]
            'min':1e-5,                 # 최소값을 설정합니다.
            'max':1e-4                  # 최대값을 설정합니다.
        },
        'optimizer':{
            'values': ["adam", 'sgd', "adamW"]
        },
        'batch_size': {
            'values': [16, 32]
        },
        'epochs': {
            'values': [10, 20]
        }
    },
    'early_terminate': {
        'type': 'hyperband', #earlystop
        'min_iter': 5
    }
}
```

- method는 random, grid, bayes 중 랜덤으로 선택하는 random 사용
- lr, optimizer, batch_size, epochs를 랜덤선택하여 진행하도록 설정
- hyperband타입을 적용하여 최소횟수를 돌고 earlystop하게함.

```
sweep_id = wandb.sweep(
    sweep=sweep_config,      # config 딕셔너리를 추가합니다.
)
wandb.agent(
    sweep_id=sweep_id,       # sweep의 정보를 입력하고
    function=sweep_train,    # train이라는 모델을 학습하는 코드를
    count=2                  # 총 5회 실행해봅니다.
)
```

- sweep을 이용하여 count만큼 자동으로 parameter를 조정하여 실행된다.

▼ 데이터

EDA

EDA 실시 내역

전처리

EDA 결과, train set의 띄어쓰기가 잘 되어 있지 않은 문장들에 대해 토크나이저가 Unknown token을 반환하는 현상을 발견하였다. 가장 심각했던 문제는 아래의 예시와 같이 모든 문장을 Unknown token으로 처리하는 경우가 있었다는 것이었다.

원문장	토크나이징 결과	전처리 결과
소소한재미ㅜㅋㅋ여탯껏삶을다시돌아보게하는영화ㅜ	[CLS] [UNK] [SEP]	소소한 재미ㅜㅋㅋ탯껏 삶을 다시 돌아보게 하는 영화ㅜ
#수경재배입문챕터 #파룻파룻	[CLS] # [UNK] # 파룻파룻 [SEP]	# 수경대 배입문 챕터 # 파룻파룻

문장의 대부분이 Unknown token으로 처리가 된다면, 문장의 의미를 파악해야 하는 STS task를 잘 학습하기 어려울 것이다. 한편, '맞알ㅋㅋ'와 같이 종성이 없는 어미에 자음이 개입된 형식에 대해서 해당 어절 전체가 Unknown token으로 처리되는 경우

도 있었다. 이러한 문제점을 해결하기 위해 train set, dev set, test set에 대해서 다음과 같은 전처리 과정을 수행하였다. 첫째, py-hanspell library를 이용해 맞춤법 및 띄어쓰기 교정을 수행하였다. 둘째, soynlp library를 통해 '맞앜ㅋㅋ'와 같은 형식을 '맞아 ㅋㅋㅋ'와 같은 형식으로 normalizing을 해주었다.

quickspacer, pyKopsacer 등 띄어쓰기 전용 library도 활용하고자 하였으나, 해당 library들이 기존 작업 환경의 dependency와 충돌하여 결국 사용하지 못했다.

아래의 표를 보면, 전처리를 적용한 데이터로 학습한 모델이 전처리를 적용하지 않은 데이터로 학습한 모델보다 pearson correlation이 0.1점 이상 상승한 것을 볼 수 있다.

Hyperparameter	데이터 전처리 여부	validation_loss	validation_perason_correlation
batch_size=16 / lr = 3e-6 / max_epoch=15 / loss=L1 / optimizer=AdamW	X	0.4560	0.9232
batch_size=16 / lr = 3e-6 / max_epoch=15 / loss=L1 / optimizer=AdamW	O	0.4123	0.9341

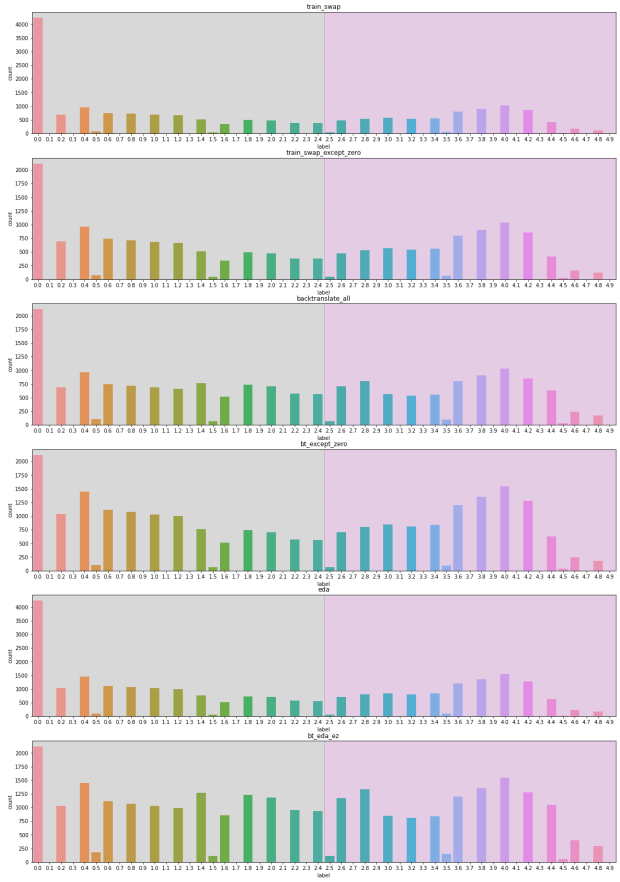
데이터 증강 : Swap, Backtranslation, EDA

- Swap : 부족한 label을 채워주면 점수가 상승할 것이라고 생각해 label==0을 제외하고 sentence_1과 sentence_2를 바꾸는 swap_except_zero데이터셋, 모든 label에 대해 바꾸는 swap데이터셋 사용
- Backtranslation : sentence1과 sentence2중 하나만 또는 둘 다 바꿔서 증강
  - Label수가 작은 것들만 골라서 증강 (bt_sep, bt_all)
  - Label이 0인 것 빼고 증강 (bt_except_zero)
  - 모두 증강 (bt_with_zero)
- EDA : Synonym Replacement, Random Insertion, Random Swap, Random Deletion을 모두 수행한 후 길이가 가장 긴 것을 선택
  - eda, eda_except_zero, bt_eda로 조합해서 만들어보았다.
- 실험 결과

data	pearson_corr
train_swap	0.9036
train_sez	0.9171
bt_all	0.911
bt_ez	0.8958
eda	0.9101
bt_eda_ez	0.909

- model: tunib/ko-en-base
- batch_size: 32
- epoch: 30
- learning_rate: 1e-5

label분포와 성능 간의 관계를 확인하기 위해 그래프로 나타내 보았다 →



- 실험 결과 tunib/electra-ko-en-base에서는 train_swap_except_zero가 가장 성능이 좋았고, klue/roberta-large에서는 train_swap이 성능이 좋았다.
- Backtranslation과 Easy data augmentation 사용해 증강한 데이터가 많을수록 모델은 성능이 오히려 떨어졌는데 문장의 의미가 달라져서 유사도 점수에 영향을 미칠 수 있기 때문에 점수를 자세하게 맞춰야 하는 regression task에 맞지 않았던 것 같다.
- Label 불균형이 있을 때 데이터를 증강해서 맞춰주는 것이 항상 성능에 좋은 것은 아니라는 것과 점수대별로 잘 맞추는 모델과 못 맞추는 모델을 앙상블하는 것이 효과적이라는 것을 알게 되었다.

▼ 모델

모델 선정 과정

	YNAT	KLUE-STS		KLUE-NLI	KLUE-NER		KLUE-RE		KLUE-DP		KLUE-MRC		WoS	
Model	F1	R^P	F1	ACC	$F1^E$	$F1^C$	$F1^{mic}$	AUC	UAS	LAS	EM	ROUGE	JGA	$F1^S$
mBERT$_{BASE}$	81.55	84.66	76.00	73.20	76.50	89.23	57.88	53.82	90.30	86.66	44.66	55.92	35.46	88.63
XLM-R$_{BASE}$	83.52	89.16	82.01	77.33	80.37	92.12	57.46	54.98	89.20	87.69	27.48	53.93	39.82	89.61
XLM-R$_{LARGE}$	**86.06**	92.97	85.86	85.93	82.27	**93.22**	58.39	61.15	92.71	**88.70**	35.99	66.77	41.20	89.80
KR-BERT$_{BASE}$	84.58	88.61	81.07	77.17	74.58	90.13	62.74	60.94	89.92	87.48	48.28	58.54	45.33	90.70
KoELECTRA$_{BASE}$	84.59	92.46	84.84	85.63	**86.11**	92.56	62.85	58.94	92.90	87.77	59.82	66.05	41.58	89.60
KLUE-BERT$_{BASE}$	85.73	90.85	82.84	81.63	83.97	91.39	66.44	66.17	89.96	88.05	62.32	68.51	46.64	91.61
KLUE-RoBERTa$_{SMALL}$	84.98	91.54	85.16	79.33	83.65	91.14	60.89	58.96	90.04	88.14	57.32	62.70	46.62	91.44
KLUE-RoBERTa$_{BASE}$	85.07	92.50	85.40	84.83	84.60	91.44	67.65	68.55	93.04	88.32	68.67	73.98	47.49	91.64
KLUE-RoBERTa$_{LARGE}$	85.69	**93.35**	**86.63**	**89.17**	85.00	91.86	**71.13**	**72.98**	**93.48**	88.36	**75.58**	**80.59**	**50.22**	**92.23**

KLUE-RoBERTa, KoELECTRA의 Fine-tuning 결과(Park et al., 2021)

모델 선정 기준은 다음과 같다. 첫째, 한국어 데이터로 학습한 PLM을 선정하였다. 우리가 수행할 task를 잘 수행하기 위해서는 한국어 문장의 문장을 잘 파악해야 하고, 구어체 문장을 잘 이해할 수 있어야 한다고 생각하였다. 따라서 다양한 한국어 데이터로

학습한 PLM이 다국어 모델보다 유리할 것이라고 판단하였다. 둘째, STS task에서 좋은 성적을 달성한 모델들을 탐색하였다. Park et al. (2021)의 실험 결과에 따르면, KLUE-RoBERTa_small, KLUE-RoBERTa_base, KLUE-RoBERTa_large 모델이 KLUE-STS task에서 준수한 성능을 보였으며, KoELECTRA_base도 좋은 성능을 달성하였다. 이 기준에 따라 **KLUE-RoBERTa-small, KLUE-RoBERTa-base, KLUE-RoBERTa-large, KoELECTRA-base**를 선정하였다. 또한, KoELECTRA-base 와 같은 ELECTRA 계열 모델 **tunib/electra-ko-base**도 자체 실험(https://github.com/tunib-ai/tunib-electra) 결과에서 KorSTS task에 대해 KoELECTRA-base과 대등한 성능을 보였다고 보고한 점을 참고하여, 실험 대상 모델에 포함하였다.

RoBERTa, ELECTRA 모두 문맥을 잘 학습하기 위한 pre-training 방법을 고안한 모델이다. RoBERTa(Yihan et al., 2019)는 Dynamic Masked Language Modeling을 통해 문맥을 잘 학습할 수 있다. ELECTRA(Clark et al., 2020)는 MLM이 사용하는 [MASK] 토큰이 실제 downstream task의 데이터에는 존재하지 않는다는 문제를 해결하기 위해 Replaced Token Detection으로 문맥 학습 방법을 개선한 모델이다. 두 모델의 pre-training 방법은 문장과 문장 사이의 의미적인 관계를 잘 모델링할 수 있도록 해준다. 따라서, 우리 조는 RoBERTa와 ELECTRA가 두 문장을 입력으로 받아서 두 문장 간의 유사도를 측정해야 하는 STS task를 잘 풀 수 있을 것이라고 판단하였다.

선정한 5가지 모델들이 STS task를 잘 수행하는지 관찰하기 위해, 조원 1명당 1개의 모델을 맡아서 다양한 실험을 진행하였다. 여러 실험을 거듭한 결과, KLUE-RoBERTA-large 모델이 dev set에 대한 pearson correlation을 최고 92~93 점까지 달성하여 전반적으로 가장 우수한 성능을 보였다. 이에 **KLUE-RoBERTa-large**를 최종 실험 모델로 선정하였다.

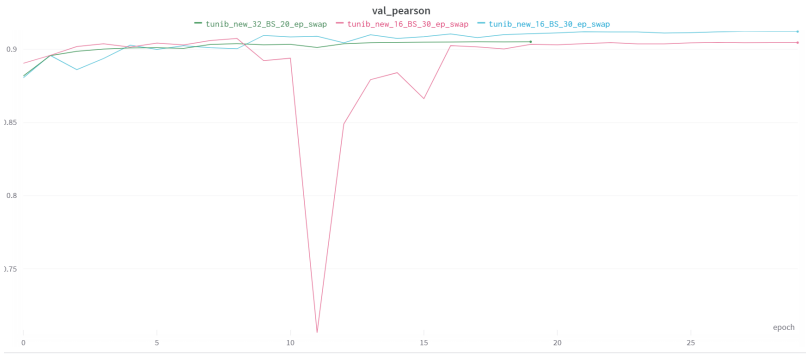
- KLUE-RoBERTa-large

KLUE-RoBERTa-large의 자세한 pre-training 정보는 다음과 같다(Park et al., 2021).

Parameter	Masking	Training steps	Batch size	Learning Rate	Device
337M	Dynamic, Whole Word Masking	500k	2048	1e-4	8× V100 GPUs

모델링 시도

- RoBERTa output의 스페셜 토큰 [SEP] 활용
  - [CLS]뿐만 아니라 [SEP] 토큰도 문맥의 정보를 잘 압축할 수 있을 것이라는 가정 하에, [CLS] 토큰과 [SEP] 토큰의 임베딩을 더하거나 연결(concat)하여 학습을 수행해 보았다. 그 결과 두 토큰의 임베딩을 연결한 것이 서로 더한 것보다 성능이 좋게 나왔다.
  - 구현 방법
    - input_ids에서 SEP에 해당하는 인덱스를 기억하고 hidden_state에서 그 인덱스 자리에 있는 state를 뽑아 CLS의 hidden state와 concat또는 add했다.
    - dense layer, dropout layer, output layer를 만들고 forward함수에서 차례로 통과시켜서 최종 label을 구했다.



  - 실험결과 concat(하늘색)이 add(핑크색)보다 성능이 좋았다.
  - baseline model(초록색)에 비해서도 성능이 조금 더 상승한 것을 확인했다.

▼ Loss function, Optimizer

Loss Function

현재 베이스라인 코드 기준 Loss function을 L1Loss를 사용하고 있다. 실험을 위해 이외에 다른 Loss Function을 조사하였고 MSELoss, HuberLoss, SmoothL1Loss를 후보로 정해 각 성능을 평가하고, L1Loss와 비교하였다. 후보 loss function을 설정한 이유는 다음과 같다. MSELoss는 Regression task에 자주 사용되기 때문에 해당 태스크에서 적절하다고 판단하였고, HuberLoss, SmoothL1Loss는 L1Loss와 MSELoss의 장점을 모두 결합한 것을 베이스로 하기 때문에 성능 향상을 기대할 수 있다.

후보 Loss Function 설명

- **L1Loss** : 실제값과 예측값의 각 요소 사이의 평균절대오차(MAE)를 측정하는 기준을 만든다.
- **MSELoss :** 실제값과 예측값의 각 요소 사이의 평균 제곱 오차(squared L2 norm)를 측정하는 기준을 만든다.
- **HuberLoss** : absolute element-wise error가 delta보다 작으면 제곱항을 사용하고, 그렇지 않으면 delta-scaled L1 값을 구하는 것이다. HuberLoss는 L1Loss와 MSELoss의 장점을 모두 결합한 것으로, delta-scaled L1 영역은 MSELoss보다 이상치에 덜 민감하게 만드는 반면, L2 영역은 0에 가까운 L1Loss에서 부드럽다.
- **SmoothL1Loss** : absolute element-wise error가 beta 값 보다 작으면 제곱항을 사용하고, 그렇지 않으면 L1 값을 구하는 것이다. MSELoss보다 이상치에 덜 민감하고 exploding gradients를 막기도 한다.

Name (4 visualized)	val_pearson_corr	val_loss	train_loss
roberta-small_3_BS_32_LR_1e-05_loss_SmoothL1Loss()	0.8765	0.2376	0.1858
roberta-small_3_BS_32_LR_1e-05_loss_HuberLoss()	0.8765	0.2376	0.1858
roberta-small_3_BS_32_LR_1e-05_loss_L1Loss()	0.856	0.5847	0.4896
roberta-small_3_BS_32_LR_1e-05_loss_MSELoss()	0.8482	0.649	0.5297

1) **실험 조건**

- model : Klue/roberta-small
- epoch : 3
- batch size : 32
- learning rate : 1e-5

2) 실험 결과

실험 결과 MSELoss는 L1Loss에 비해 성능이 떨어졌다. MSELoss가 L1Loss보다 성능이 떨어지는 이유는 MSELoss는 다른 값들에 비해 훨씬 큰 값이나 작은 값인 이상치가 존재하는 경우 영향을 받아 값이 커지게 되기 때문에 위의 결과처럼 성능이 나온 것으로 보인다.

실험 전 예상한 것과 같이 HuberLoss나 SmoothL1Loss가 L1Loss보다 성능이 좋았다. 또한, 두 함수는 동일한 성능을 보였다. 그 이유는 다음과 같다.

- 만약 delta를 1로 설정하면 HuberLoss는 SmoothL1Loss와 같다. HuberLoss는 SmoothL1Loss와 delta 값 (SmoothL1Loss에서 beta 값)에 따라서 달라진다.
- SmoothL1Loss는 HuberLoss와 밀접하게 관련이 있으며, Huber(x, y) / beta와 동일하다. (SmoothL1Loss의 beta는 HuberLoss의 delta로 알려져 있다) 하지만 다음과 같은 차이가 발생한다.
  - beta → 0 일 때, SmoothL1Loss는 L1Loss로 수렴하고, HuberLoss는 일정하게 상수 0으로 수렴한다. 만약 beta=0 이라면 SmoothL1Loss는 L1Loss와 같다.
  - beta → ∞ 일 때, SmoothL1Loss는 상수 0으로 수렴하는 반면, HuberLoss는 MSELoss로 수렴한다.

- SmoothL1Loss의 경우 beta 값이 변화함에 따라 L1 segment는 일정한 기울기를 가진다. HuberLoss의 경우 L1 segment의 기울기는 beta이다.

현재 실험에서는 SmoothL1Loss와 HuberLoss 함수 둘 다 delta와 beta값을 default값인 1.0으로 두고 실험했기 때문에 동일한 성능이 나온 것으로 보인다. 모델 학습에서 loss function을 HuberLoss, SmoothL1Loss, L1Loss를 옵션으로 선택할 수 있도록 구현하여 다양한 실험을 할 수 있도록 하였다.

Optimizer

현재 베이스라인 코드 기준 옵티마이저를 AdamW를 사용하고 있다. 실험을 위해 이외에 다른 옵티마이저를 조사하였고 아래와 같이 후보를 정해 각 성능을 평가하고, AdamW와 비교하였다. 후보 선정 이유는 다음과 같다. 가장 보편적으로 사용되고 있는 옵티마이저인 Adam이 일반적으로 좋은 학습 성능을 보이기 때문에 함께 실험하였고, Adam이 RMSProp와 Momentum을 조합한 방식이라고 볼 수 있기 때문에 비교를 위해 RMSprop도 추가적으로 실험하였다. NAdam과 RAdam의 경우 AdamW와 같이 각자 다른 관점에서 Adam을 수정하고 개선한 것이기 때문에 성능 비교가 필요하다고 판단하였다.

Name (5 visualized)	val_pearson ▾	val_loss	train_loss
roberta-large_5_32_1e-5_Adam	0.9242	0.1741	0.03912
roberta-large_5_32_1e-5_AdamW	0.9182	0.1981	0.02946
roberta-large_5_32_1e-5_NAdam	0.918	0.1781	0.0317
roberta-large_5_32_1e-5_RAdam	0.9082	0.1939	0.05135
roberta-large_5_32_1e-5_RMSprop	0.908	0.2638	0.05088

1) **실험 조건**

- model : klue/roberta-large
- epoch : 5
- batch size : 32
- learning rate : 1e-5
- loss_function : torch.nn.SmoothL1Loss
- drop_out: 0.1

2) **실험 결과**

실험 결과 Adam이 가장 좋은 성능을 보였다. 그러므로 Adam을 우선으로 두고 앞으로의 모델 성능 실험을 진행하도록 권장하고, 다른 옵티마이저들도 옵션으로 선택할 수 있도록 구현하여 다양한 실험을 할 수 있도록 하였다. 또한 Adaptive learning rate을 사용하는 optimizer는 ‘Bad local optima convergence problem’ 현상이 발생할 수 있기 때문에 warmup 과 함께 사용할 수 있도록 구현하였다.

Learning Rate Scheduler

learning rate는 옵티마이저가 손실 함수의 최솟값에 도달하는 단계의 크기를 제어한다. learning rate가 성능에 영향을 주는 요소인 만큼 어떻게 설정하는지 매우 중요하다. 그러므로 learning rate scheduler를 통하여 learning rate를 조정하여 더 좋은 성능을 기대할 수 있을 것이다.

또한, AdamW가 소개된 논문 'Decoupled Weight Decay Regularization'(2019)에서 Adam이 learning rate scheduler를 통해 성능이 크게 향상될 수 있다고 주장하였기 때문에 해당 실험이 필요하다고 판단하였다. 논문의 실험에서 적용된 step-drop,

cosine annealing, warm restart with cosine annealing 스케줄러 형태인 파이토치의 StepLR, CosineAnnealingLR, CosineAnnealingWarmRestarts를 후보로 두고 성능 비교를 진행하였다.

- **StepLR** : 가장 흔히 사용되는 learning rate scheduler 중 하나로, 일정한 Step 마다 learning rate에 gamma를 곱해주는 방식이다.
- **CosineAnnealingLR** : learning rate가 cosine 함수의 주기를 따라 감소하고, 증가하는 과정을 반복한다.
- **CosineAnnealingWarmRestarts** : cosine annealing 방식에서 Warm restart를 통해 학습 중간중간에 learning rate를 증가시켜 큰 폭의 weight update를 만들어 가파른 local minimum에서 빠져나올 기회를 제공한다.

Name (8 visualized)	val_pearson	val_loss	train_loss
roberta-small_10_32_1e-5_AdamW	0.8841	0.2348	0.07233
roberta-small_10_32_1e-5_AdamW_CosineAnnealingLR	0.8712	0.2553	0.09755
roberta-small_10_32_1e-5_AdamW_CosineAnnealingWarmRestart	0.8782	0.2601	0.05968
roberta-small_10_32_1e-5_AdamW_StepLR	0.8814	0.2344	0.05963

Name (4 visualized)	val_pearson	val_loss	train_loss
roberta-small_20_32_1e-5_AdamW	0.8946	0.2075	0.02654
roberta-small_20_32_1e-5_AdamW_CosineAnnealingLR	0.8881	0.2316	0.04324
roberta-small_20_32_1e-5_AdamW_CosineAnnealingWarmRestart	0.8919	0.2119	0.03604
roberta-small_20_32_1e-5_AdamW_StepLR	0.8818	0.2317	0.04829

1) **실험 조건**

- model : klue/roberta-small
- epoch : [10, 20]
- batch size : 32
- learning rate : 1e-5
- loss_function : torch.nn.SmoothL1Loss
- drop_out: 0.1
- optimizer : AdamW

2) **실험 결과**

실험 결과 스케줄러를 통해서 learning rate를 조절한 것에 비해 고정된 learning rate의 경우가 더 성능이 좋았다. 해당 결과에 대한 이유는 다음과 같이 예상할 수 있다.

- 적절한 파라미터 설정이 어렵다.
  - 각 스케줄러에는 어떤 주기로 learning rate를 낮출 것인지를 사용자가 직접 정해야 하기 때문에 적절한 주기를 선택하기에 어려움이 있다. 또한, learning rate를 어디까지 낮출 것인지 혹은 어떤 비율로 낮출 것인지에 대해서도 생각해야 한다.
- 'Bad local optima convergence problem'는 Warmup이 필요하다.
  - 'Bad local optima convergence problem'은 Adaptive learning rate을 사용하는 optimizer에서는 모두 발생하는 현상이다. 이는 학습 초기에 샘플이 매우 부족하여 adaptive learning rate의 분산이 매우 커지고 이에 따라 최적이 아닌

local optima에 너무 일찍 도달하여 학습이 거의 일어나지 않는 현상이다.

- 학습 초기의 convergence problem을 해결하기 위해 다양한 시도가 있었고, 지금까지 각광받는 방법은 바로 Warmup heuristic이다.
- learning rate warmup은 말그대로 천천히 learning rate를 올리는 작업을 뜻한다. 하지만 실험에서 사용한 스케줄러는 모두 초기 학습률을 선택한 다음 스케줄러에 따라 점차적으로 감소시키는 형태이다.

3) **추가 실험**

위와 같은 실험 결과에 따라 Warmup 이 적용된 형태의 스케줄러가 적절하다고 판단하였고, transformers 라이브러리에서 제공하는 get_linear_schedule_with_warmup와 get_cosine_with_hard_restarts_schedule_with_warmup을 사용하여 성능을 비교하였다.

- **get_linear_schedule_with_warmup** : warmup 기간 동안 옵티마이저에서 설정된 초기 learning rate로 선형적으로 증가하다가 이후 옵티마이저에서 설정된 초기 learning rate에서 0으로 선형적으로 감소하는 학습률로 스케줄을 생성한다.
- **get_cosine_with_hard_restarts_schedule_with_warmup** : 0에서 옵티마이저에 설정된 초기 learning rate까지 선형적으로 증가하는 워밍업 기간 후에 옵티마이저에서 설정된 초기 learning rate 와 0 사이의 코사인 함숫값에 따라 감소하는 학습률로 스케줄을 생성한다.

learning rate 그래프

Name (3 visualized)	val_pearson	val_loss	train_loss
roberta-small_10_32_1e-5_AdamW_cosine_warmup	0.8871	0.2326	0.05074
roberta-small_10_32_1e-5_AdamW_linear_warmup	0.887	0.2327	0.05444
roberta-small_10_32_1e-5_AdamW	0.8841	0.2348	0.07233

3-1) **실험 조건**

- model : klue/roberta-small
- epoch : 10
- batch size : 32
- learning rate : 1e-5
- warmup_ratio : 0.1
- loss_function : torch.nn.SmoothL1Loss
- drop_out: 0.1

- optimizer : AdamW

3-2) **실험 결과**

실험 결과에서 warmup을 적용한 스케줄러를 사용했을 때가 그렇지 않을 때보다 성능이 좋았고, 그 중에서도 cosine 형태의 스케줄러가 linear보다 조금 더 나은 성능을 보였다. 해당 스케줄러의 파라미터의 경우 warmup_ratio만 조절하면 되기 때문에 모델 학습에서 get_cosine_with_hard_restarts_schedule_with_warmup을 사용하기로 결정하였다.

SMART Loss

SMART Loss(Jiang et al., 2019)는 GLUE STS task에서 SOTA를 달성한 Fine-tuning 방법론으로, 이를 RoBERTa에 적용했을 때, RoBERTa보다 파라미터 수가 약 30배 더 많은 T5도 능가하는 성능을 보인 방법론이다. Jiang et al. (2019)는 현재 상용화된 PLM들은 파라미터 수가 많다 보니 fune-tuning 단계에서 적은 dataset에 overfitting될 가능성이 있다고 지적하며, 기존 loss function에 규제 항을 추가하는 방식(Smoothness-Inducing Adversarial Regularization)으로 PLM의 일반화 성능을 높일 수 있다고 제안하였다.

규제 항의 식은 아래와 같다.

$$\mathcal{R}_s(\theta) = \frac{1}{n}\sum_{i=1}^{n} \max_{\|\widetilde{x}_i - x_i\|_p \le \epsilon} \ell_s(f(\widetilde{x}_i;\theta), f(x_i;\theta)),$$

규제 항의 입력으로는 PLM이 원본 문장의 임베딩 x_i을 입력으로 받아 출력하는 logit $f(x_i, \theta)$ 와, PLM이 원본 문장의 임베딩에 노이즈가 더해진 $x_i^{\sim}$ 을 입력으로 받아 출력하는 logit $f(x_i^{\sim}, \theta)$이 들어가고, 규제 모델은 이 두 출력값의 차이인 $l_s(f(x_i, \theta), f(x_i^{\sim}, \theta))$ 를최대화하는 방식으로 학습한다. 이렇게 출력되는 규제 loss는 원래의 loss와 더해져서, 모델이 training set에 overfitting되지 않도록 만들어준다. 본 프로젝트의 task는 regression 문제이므로, Jian et al. (2019)를 따라 l_s 를 $l_s(p, q) = (p - q)^2$ 로 정의해 활용하였다. 실험 결과는 다음과 같다.

PLM	hyperparameter	Loss	Optimizer	validation loss	validation pearson correlation	test pearson correlation (private score 적용)
KLUE-RoBERTa-large	batch_size=32 / lr=3e-6 / max_epoch=15 /	L1 + SMART Loss	AdamW	0.4906	**0.9277**	-
KLUE-RoBERTa-large	batch_size=32 / lr=3e-6 / max_epoch=15 /	L1	AdamW	0.4123	**0.9341**	**0.9285**

SMART Loss를 적용한 결과 validation 성능은 오히려 감소하였다. 대회 제출횟수의 제한으로 인해 SMART 모델로 학습한 단일 모델을 test set에 적용하여 일반화 성능을 검증하지는 못하였으나, 다른 모델과 ensembel 시 팀 최종 제출의 private score(0.9337)를 높이는 데 기여할 수 있었다.

▼ K-fold, 앙상블

K-fold (5-fold로 훈련)

- train_set과 dev_set을 합쳐서 5-fold로 훈련을 하였고 val_pearson score는 0.96대가 나올 정도로 높은 점수가 나옴 (dev_set을 포함하여 훈련을 함과 동시에 dev_set으로 validation을 하니 당연한 결과)
- 실제로 제출을 하였을 때는 리더보드상에서 점수가 높지 않았음 ⇒ 0.9 초반대의 score과 결과로 나옴
- R-drop, 앙상블등의 기타 기법으로 일반화 성능을 높여주는 방식을 채택하기로 함 (K-fold 미사용)

Ensemble

- 성능이 잘 나온 모델들을 앙상블의 후보군으로 선정함

모델	option	test_pearson_score
klue/roberta-large	batch_size :16 lr : 3e-6 loss_fct : L1 Optimizer : AdamW warmup : 0.0	93.41
klue/roberta-large	batch_size :16 lr : 3e-6 loss_fct : SmartLoss Optimizer : Adam warmup : 0.1	92.96
klue/robera-large	batch_size :16 lr : 1e-6 loss_fct : SmoothL1 Optimizer : Adam warmup : 0.001	0.9333
klue/robera-large	batch_size :16 lr : 1e-6 loss_fct : L1 Optimizer : Adam warmup : 0.001	0.9354
klue/robera-large	batch_size :16 lr : 1.5e-05 loss_fct : L1 Optimizer : AdamW warmup : 0.15 dropout : 0.11	0.932
klue/roberta-large	batch_size :16 lr : 1.4e-05 loss_fct : L1 Optimizer : AdamW warmup : 0.18 dropout : 0.11 R-drop : True	0.9327

- 위 모델들에서 앙상블할 대상을 선정하는 시행을 랜덤으로 대량으로 반복 ⇒ 그 중 val_pearson 이 가장 높은 모델을 제출

▼ 그 외 사항

- **R-drop**
- drop_out 기법의 임의성 때문에 훈련 성능과 추론 성능간에 괴리가 발생하는 문제가 발생하였으며 이를 극복하기 위해 고안된 기법
- 하나의 input으로 모델을 2번 통과시켜 나온 output 간의 KL-div 값을 loss 값에 더해 줌으로 써 drop_out 기법의 임의성을 축소시킴 (아래는 R-drop의 loss_fct 계산 알고리즘)

Algorithm 1 R-Drop Training Algorithm

Input: Training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$.
Output: model parameter w.

1: Initialize model with parameters w.
2: **while** not converged **do**
3: randomly sample data pair $(x_i, y_i) \sim \mathcal{D}$,
4: repeat input data twice as $[x_i; x_i]$ and obtain the output distribution $[\mathcal{P}_1^w(y_i|x_i), \mathcal{P}_2^w(y_i|x_i)]$,
5: calculate the negative log-likelihood loss $\mathcal{L}_{NLL}^i$ by Equation (3),
6: calculate the KL-divergence loss $\mathcal{L}_{KL}^i$ by Equation (2),
7: update the model parameters by minimizing loss $\mathcal{L}^i$ of Equation (4).
8: **end while**

- Regression task에서의 R-drop 적용 수식

$$\mathcal{L}_{mse_r} = ||y_1' - y_2'||_2, \tag{7}$$

and we add $\mathcal{L}_{mse_r}$ with conventional MSE loss: $\mathcal{L}_{mse} = ||y - y_1'||_2 + ||y - y_2'||_2$. The final optimization objective is:

$$\mathcal{L} = \mathcal{L}_{mse} + \alpha \mathcal{L}_{mse_r}. \tag{8}$$

- 실제 구현 (loss_func는 L1 및 SmoothL1 사용, R_drop_alpha : 1)

```
# R-drop for regression task
loss_r = self.loss_func(logits1, logits2)
loss = self.loss_func(logits1, y.float()) + self.loss_func(logits2, y.float())
loss = loss + cfg.train.R_drop_alpha*loss_r
```

- **Mixed_precision_Training**
  - 참고 자료 : https://hoya012.github.io/blog/Mixed-Precision-Training/
  - 빠른 확인이 필요한 실험 내용을 16bit precision을 사용하여 훈련하여 메모리와 훈련 속도를 높임
  - tokenizer max_length와 precision의 조합으로 빠르게 다양한 내용들을 실험해봄

▼ References

Clark, K., Luong, M. T., Le, Q. V., & Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*

Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Zhao, T. (2019). Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437*

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*

.

Park, S., Moon, J., Kim, S., Cho, W. I., Han, J., Park, J., ... & Cho, K. (2021). Klue: Korean language understanding evaluation. *arXiv preprint arXiv:2105.09680*