KOREATECH 한국기술교육대학교

9 주차 과제

학번	2020136087
이름	윤아현
수강 과목	자연어처리
담당 교수	임상훈 교수님
제출일	2024.05.16

문제 및 해결 방법 (구현 코드)

1. IMDB PLM 학습 (45 점)

* HW 2 에서 진행한 IMDB 데이터셋을 학습 및 분류하는 PLM 기반 분류기를 작성하시오.
* 실습에 사용하지 않은 PLM 모델을 huggingface hub 에서 불러와 사용하시오.
* 실습에 사용한 데이터는 한국어 데이터이지만 IMDB 의 경우 영어 데이터임

* 최소 3 개의 PLM 을 동일 hyperparameter 를 통해 학습 및 평가 해 보고 성능을 비교 분석하시오
* 각자가 구현 및 제출한 HW2 의 모델과 비교하여 어떠한지
* 각 PLM 별로 성능이 어떠하고 왜 그런지

GRADING
* PLM 1 개당 15 점 (총 45 점)

Code

```
# plm_name = "distilbert-base-uncased"
# plm_name = "albert-base-v2"
plm_name = "xlnet-base-cased"
```

```
# 알아서 Classification 수행해줌
# sentiment_classifier 모델을 직접 안짜도 알아서 다 해줌
from transformers import AutoTokenizer, AutoModel
from transformers import AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained(plm_name)
# label, 대상을 알려준다.
model = AutoModelForSequenceClassification.from_pretrained(plm_name, num_labels=2)
```

```
from torch.utils.data import Dataset, DataLoader

# define dataset class
class SentimentDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.tokenizer= tokenizer

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        label = int(self.data[index][2])
        text = self.data[index][1]
        # tokenizer가 자동으로 수행함
        inputs = self.tokenizer(text, return_tensors="pt", padding="max_length",
                                truncation=True, max_length=128)
        inputs = {key: inputs[key].squeeze() for key in inputs}

        # inputs['label'] = torch.tensor(label)

        return inputs|{'label':torch.tensor(label)} # tokenizer에 label만 추가해주면 됨
```

```python
import lightning as pl

class SentimentClassifierPL(pl.LightningModule):
    def __init__(self, sentiment_classifier):
        super(SentimentClassifierPL, self).__init__()
        self.model = sentiment_classifier

        self.validation_step_outputs = []
        self.test_step_outputs = []
        self.save_hyperparameters()

    def forward(self, inputs):
        return self.model(inputs)

    def training_step(self, batch, batch_idx):
        labels = batch.pop('label')
        outputs = model(**batch, labels=labels)
        loss = outputs.loss # 자동으로 loss 계산해주기 때문에 바로 가져와서 사용
        logits = outputs.logits # 자동으로 logits 계산해주기 때문에 바로 가져와서 사용
        self.log("train_loss", loss)
        return loss

    def validation_step(self, batch, batch_idx):
        labels = batch.pop('label')
        outputs = model(**batch, labels=labels)
        loss = outputs.loss
        logits = outputs.logits
        self.log("val_loss", loss)
        self.validation_step_outputs.append((loss, logits, labels))
        return loss, outputs, labels

    def on_validation_epoch_end(self):
        outputs = self.validation_step_outputs
        avg_loss = torch.stack([x[0] for x in outputs]).mean()
        self.log("avg_val_loss", avg_loss)

        all_outputs = torch.cat([x[1] for x in outputs])
        all_labels = torch.cat([x[2] for x in outputs])
        all_preds = all_outputs.argmax(dim=1)
        accuracy = (all_preds == all_labels).float().mean()
        self.log("val_accuracy", accuracy)
        self.validation_step_outputs.clear()

    def test_step(self, batch, batch_idx):
        labels = batch.pop('label')
        outputs = model(**batch, labels=labels)
        loss = outputs.loss
        logits = outputs.logits
        self.log("test_loss", loss)
        self.test_step_outputs.append((loss, logits, labels))
        return loss, outputs, labels

    def on_test_epoch_end(self):
        outputs = self.test_step_outputs
        avg_loss = torch.stack([x[0] for x in outputs]).mean()
        self.log("avg_test_loss", avg_loss)

        all_outputs = torch.cat([x[1] for x in outputs])
        all_labels = torch.cat([x[2] for x in outputs])
        all_preds = all_outputs.argmax(dim=1)
        accuracy = (all_preds == all_labels).float().mean()
        self.log("test_accuracy", accuracy)
        self.test_step_outputs.clear()

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.model.parameters(), lr=5e-6)
        return optimizer
```

```python
wandb.login()

def check_performance(model,tokenizer, train_data, val_data, test_data, wandb_log_name):
    wandb_logger = WandbLogger(project="NLP", name=wandb_log_name, group="Lec07")

    pl_model = SentimentClassifierPL(model)

    train_dataset = SentimentDataset(train_data,tokenizer)
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_dataset = SentimentDataset(val_data,tokenizer)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
    test_dataset = SentimentDataset(test_data,tokenizer)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

    trainer = pl.Trainer(
        max_epochs=1,
        accelerator="gpu",
        logger=wandb_logger,
        callbacks=[ModelSummary(max_depth=2)],
        precision=16
    )

    trainer.fit(
        model=pl_model,
        train_dataloaders=train_loader,
        val_dataloaders=val_loader
    )

    trainer.test(dataloaders=test_loader)

    wandb.finish()
```

```python
pl_model = SentimentClassifierPL(model)
```

```python
check_performance(pl_model, tokenizer, train_data, val_data, test_data, "distilbert")
```

```python
check_performance(pl_model, tokenizer, train_data, val_data, test_data, "albert")
```

```python
check_performance(pl_model, tokenizer, train_data, val_data, test_data, "xlnet")
```

Outputs

1 번. DistilBERT

Google BERT 의 Model 에서 학습 시간과 모데 크기를 줄인 BERT 의 경량화 버전

성능: 64.67%

Test metric	DataLoader 0
avg_test_loss	0.6585080027580261
test_accuracy	0.6466667056083679
test_loss	0.6591495871543884

2 번. ALBERT

Cross-Layer 파라미터 공유와 같은 기법을 통해 모델 크기를 줄인 BERT 모델

성능: 52.33%

Test metric	DataLoader 0
avg_test_loss	0.688616931438446
test_accuracy	0.5233333706855774
test_loss	0.680382490158081

3 번. XLNet

Transformer 을 기반으로 하여 문맥적 이해가 중요한 task 에 적합한 모델

성능: 57.67%

Test metric	DataLoader 0
avg_test_loss	0.672248363494873
test_accuracy	0.5766666531562805
test_loss	0.6722908616065979

※기존의 HW2 모델과 다른 점

기존 설계 모델의 성능: 51.33%

당연하게도, 직접 설계한 모델보다는 3 개의 사전 훈련 모델을 가져와 사용하는 것이 더 높은 성능을 보였다. 기존의 설계 모델은 데이터셋에만 있는 단어와 단순한 선형망을 통해 훈련을 수행하기 때문에 문맥에 대한 정확한 이해가 어렵다.

사전 훈련 모델은 대규모 언어들로 학습을 시켰기 때문에 기존 모델보다는 훨씬 더 많은 단어에 대해 학습을 수행하였으며, Transformer 과 같은 모델을 사용하였기에 문맥 정보 또한 더 잘 파악할 수 있다는 장점이 있다.

※3 개의 사전 학습 모델 비교

XLNet 이 자연어 처리(복잡한 문맥 파악)에 더 적합하여 성능이 가장 높을 것이라고 가정하였다. 하지만, 예상과 달리 DistilBERT 모델이 더 좋은 성능을 낸 것을 확인할 수 있다.

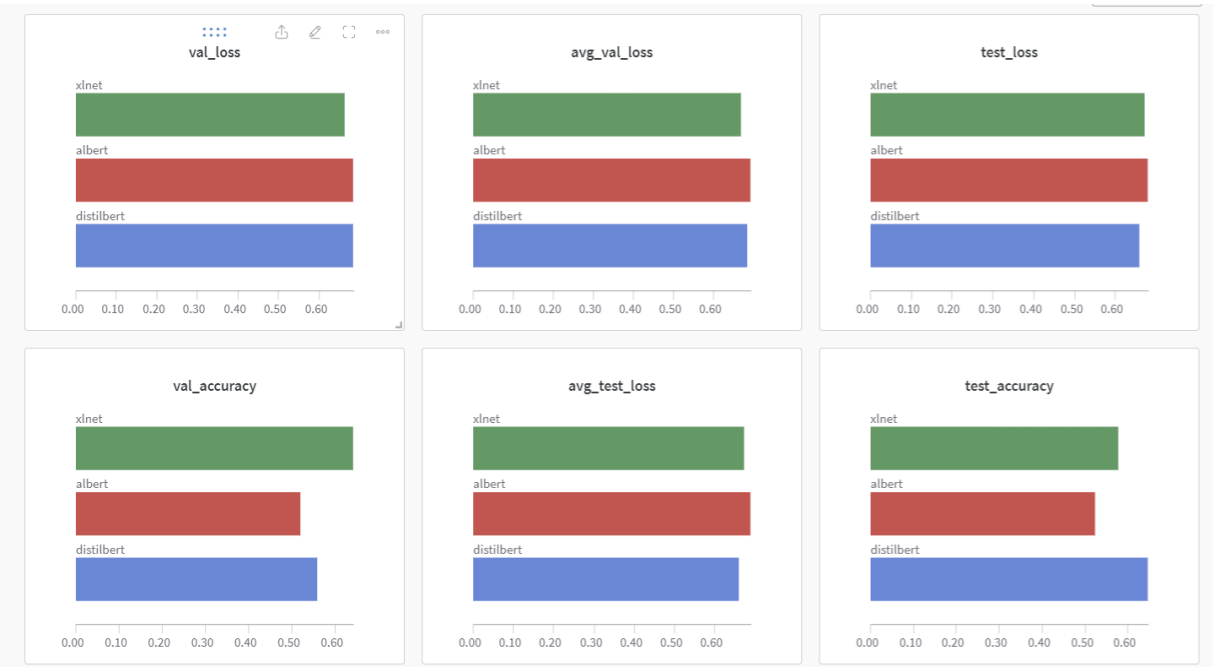
ALBERT 의 경우, 각 계층에서 같은 파라미터를 공유하기에 각 계층간 다양한 특징을 추출하기에는 한계가 있을 것이라고 생각하였다.

DistilBERT 는 기존 BERT 의 경량화 버전이기에, BERT 보다 더 간단해졌지만 유사한 방식으로 데이터를 처리하기에 성능이 높은 것이라고 추측한다. 특히 BERT 는 Transformer 의 Encoder 부분만을

사용하기에 기존 문장들의 문맥 정보를 잘 파악하여 감정 분류에는 더 좋은 성능을 보인다고 생각하였다.

XLNet 또한, 전체적인 문맥을 파악하는 데에 높은 성능을 보이지만, IMDB 리뷰 데이터셋에 정밀한 파라미터 튜닝이 수행되지 않아 성능이 낮게 나왔다고 생각하였다.

WANDB 결과



2. Hyperparameter 조정 (15 점)

* 1 번 과제의 가장 성능이 좋은 모델을 통해 hyperparameter 변화에 따른 성능을 비교하시오.
* Batch size, learning rate 에 대한 성능 비교 필수

1 번 실험

Epoch: 10

Batch_size: 16

Learning_rate: 2e-5

성능: 70.67%

2 번 실험

Epoch: 10

Batch_size: 16

Learning_rate: 1e-4

성능: 71.00%

3 번 실험

Epoch: 10

Batch_size: 32

Learning_rate: 2e-5

성능: 69.67%

4 번 실험

Epoch: 10

Batch_size: 64

Learning_rate: 1e-4

성능: 68.99%

※ weight_decay = 0.01, eps=1e-8 고정

1 번. 학습률 영향

배치 사이즈를 16 으로 고정해놓고 학습률만 수정하여 훈련시켰을 경우, 더 높은 학습률이 조금 더 나은 결과를 가져왔지만 엄청난 영향을 미치지 않은 것을 확인할 수 있다.

2 번. 배치 사이즈 영향

배치 사이즈를 16, 32, 64 로 다양하게 훈련시켰을 경우, 배치 사이즈가 가장 낮은 16 일 때 성능이 제일 좋았던 것을 확인할 수 있다.

배치 사이즈를 늘리면, 일반화된 학습은 가능하지만 각 문장들이 가지는 구체적인 문맥 정보들을 파악하는 것이 어렵다고 추측된다. (local minimum 문제)

3. T5 를 통한 Machine Translation 모델 학습 (40 점)

* Huggingface hub 의 T5 모듈 및 weight 을 통해 Seq2Seq 실습에 사용한 English-French MT 모델을 구현하시오.
* Seq2Seq 실습에 사용한 데이터셋 사용 (http://www.manythings.org/anki/fra-eng.zip)
* T5 모델은 "google-t5/t5-small"를 사용
* 참고: https://huggingface.co/docs/transformers/model_doc/t5#transformers.T5ForConditionalGeneration
* 실습에 사용한 모델과의 성능 비교

GRADING
* 학습 및 평가 (40 점)

Code

```
from transformers import AutoTokenizer, T5ForConditionalGeneration

plm_model = "google-t5/t5-small"

tokenizer = AutoTokenizer.from_pretrained(plm_model)
model = T5ForConditionalGeneration.from_pretrained(plm_model)
```

```python
class TranslationDataset(Dataset):
    def __init__(self, data, tokenizer, max_length=30):
        self.data = data
        self.tokenizer = tokenizer
        self.input_texts = [item[0] for item in data]  # 영어 문장
        self.target_texts = [item[1] for item in data]  # 프랑스어 문장
        self.max_length = max_length

        # Pre-tokenize all inputs and targets
        self.encoder_inputs = self.tokenizer(
            [f"translate English to French: {text}" for text in self.input_texts],
            max_length=max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )

        self.decoder_inputs = self.tokenizer(
            self.target_texts,
            max_length=max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )

        # Replace padding token id's in labels by -100 so they are ignored by the loss
        self.decoder_inputs['input_ids'][self.decoder_inputs['input_ids'] == self.tokenizer.pad_token_id] = -100

    def __len__(self):
        return len(self.input_texts)

    def __getitem__(self, idx):
        encoder_input = {key: val[idx] for key, val in self.encoder_inputs.items()}
        labels = self.decoder_inputs['input_ids'][idx]
        return encoder_input, labels
```

```python
import lightning as pl

class TranslateModelPL(pl.LightningModule):
    def __init__(self, translate_model):
        super(TranslateModelPL, self).__init__()
        self.model = translate_model
        self.loss = nn.CrossEntropyLoss()

        self.validation_step_outputs = []
        self.test_step_outputs = []
        self.save_hyperparameters()

    def forward(self, inputs):
        return self.model(inputs)

    def training_step(self, batch, batch_idx):
        encoder_input, labels = batch
        outputs = model(**encoder_input, labels=labels)
        loss = outputs.loss
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def validation_step(self, batch, batch_idx):
        encoder_input, labels = batch
        outputs = model(**encoder_input, labels=labels)
        loss = outputs.loss
        self.log("val_loss", loss)
        self.validation_step_outputs.append((loss, outputs, labels))
        return loss, outputs, labels

    def on_validation_epoch_end(self):
        outputs = self.validation_step_outputs
        avg_loss = torch.stack([x[0] for x in outputs]).mean()
        self.log("avg_val_loss", avg_loss)

```

```
        self.validation_step_outputs.clear()

    def test_step(self, batch, batch_idx):
        encoder_input, labels = batch
        outputs = model(**encoder_input, labels=labels)
        loss = outputs.loss
        self.log("test_loss", loss)
        self.test_step_outputs.append((loss, outputs, labels))
        return loss, outputs, labels

    def on_test_epoch_end(self):
        outputs = self.test_step_outputs
        avg_loss = torch.stack([x[0] for x in outputs]).mean()
        self.log("avg_test_loss", avg_loss)

        self.test_step_outputs.clear()

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.model.parameters(), lr=5e-6)
        return optimizer
```

```
import wandb
from lightning.pytorch.loggers import WandbLogger
from lightning.pytorch.callbacks import ModelSummary

wandb.login()

def check_performance(model,tokenizer, train_data, val_data, test_data, wandb_log_name):
    wandb_logger = WandbLogger(project="NLP", name=wandb_log_name, group="Lec07")

    pl_model = TranslateModelPL(model)

    train_dataset = TranslationDataset(train_data,tokenizer)
    train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
    val_dataset = TranslationDataset(val_data,tokenizer)
    val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
    test_dataset = TranslationDataset(test_data,tokenizer)
    test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

    trainer = pl.Trainer(
        max_epochs=1,
        accelerator="gpu",
        logger=wandb_logger,
        callbacks=[ModelSummary(max_depth=2)],
        precision=16
    )

    trainer.fit(
        model=pl_model,
        train_dataloaders=train_loader,
        val_dataloaders=val_loader
    )

    trainer.test(dataloaders=test_loader)

    wandb.finish()
```

```
pl_model = TranslateModelPL(model)
```

```
check_performance(pl_model, tokenizer, train, vali, test, "t5")
```

```
input_sentence = "Are we leaving soon?"

task_prefix = "translate English to French: "
input_text = task_prefix + input_sentence

inputs = tokenizer(input_text, return_tensors="pt")

outputs = model.generate(**inputs, max_length=30, num_beams=4, early_stopping=True)

translation = tokenizer.decode(outputs[0], skip_special_tokens=True)

print(f"Input: {input_sentence}")
print("Output: y allons nous bientot ?")
print(f"Translation: {translation}")
```

주의사항

Out Of Memory Issue 로 GPU 가 안돌아가서, Train, Vali, Test 데이터 개수를 대폭 줄여서 학습을 진행하였다.

Outputs

```
Input: Are we leaving soon?
Output: y allons nous bientot ?
Translation: sortons nous bientôt?
```

위의 결과는 마지막 코드의 출력 값이다.

이전의 Seq2Seq 모델을 훈련시켰을 때 제대로 된 결과가 나오지 않았다. (동일한 단어가 2 번 이상 나오거나 문장이 너무 짧거나 긴 이슈)

하지만, T5 모델로 훈련하였을 때 Epoch 를 1 번만 실행시켜본 것 치고는 엄청난 결과가 나왔다.

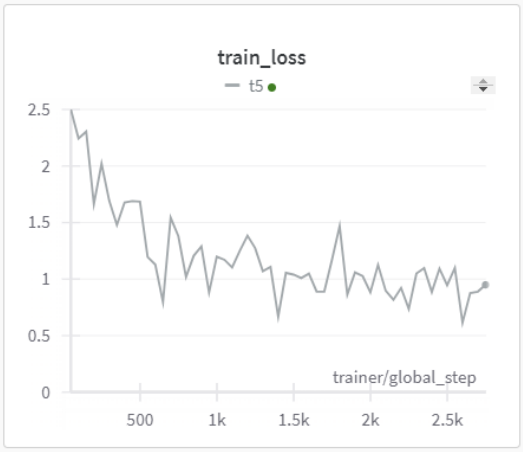
앞의 단어 2 개를 제외한 뒤의 단어를 모두 맞힌 것을 확인할 수 있다.

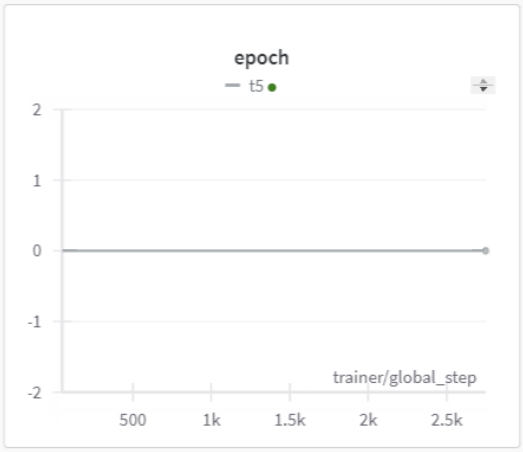
사전 훈련된 T5 모델은 기존의 Seq2Seq 의 모델과 달리 양방향 Encoder 를 사용한다. (Decoder 는 어쩔 수 없이 순차적으로 진행이 필수임) 또한, T5 모델의 특장점에 대해 살펴보니 모델이 다른 문장들(Input 값)에서 획득한 언어 패턴과 문맥 정보를 다른 문장에 사용하여 여러 문장에서의 문맥 정보를 더 풍부하게 획득할 수 있다는 장점이 있다.

이를 통해, 적절한 문맥과 뉘앙스 같은 것을 파악할 수 있다.

WANDB

Charts 2


train_loss
t5
2.5
2
1.5
1
0.5
0
500
1k
1.5k
2k
2.5k
trainer/global_step


epoch
t5
2
1
0
-1
-2
500
1k
1.5k
2k
2.5k
trainer/global_step