Week4 학습정리

Week4-1

라이브 Q&A

Q1

pooler output = last hidden state + linear + tanh activation이고 pooler ouput은 nsp 학습한 레이어이다.

다른 태스크를 풀고자 한다면 둘 다 적용 후 성능이 높은 것으로 사용하자.

- BERTforSequenceClassification 문장 분류에서는 pooler ouput 사용

Q2

head : 하나의 레이어에서 어텐션 행렬을 계산할 때 여러 방식이 문맥 임베딩을 생성하기 위해 사용

layer : 깊을수록 task 학습 잘하지만 복잡성 증가해 실제 프로덕션하기 힘듦

Week4-2

Tensorboard는 시각화 도구이고 시각화 가능한 요소는 아래와 같다.

- visualize metric (loss, acc)
- visualize graph
- visualize weights, biases, tensors
- visualize embeddings (dimesion reduction)
- visualize test, images, audio

설정

```
from torch.utils.tensorboard import SummaryWriter

logdir_path = "/content/drive/MyDrive/원티드AIML/week4_log" # log 파일을 저장할 경로를 지정
logdir_path = os.path.join(logdir_path, "summary/review_clf") # 이번 프로젝트 로그를 저장할 파일 경로 정의
writer = SummaryWriter(logdir_path) # 텐서 보드에 로그 남김

# tensorboard extension 설치
%load_ext tensorboard

# tensorboard를 web application 대신 주피터 노트북 내에서 실행
%tensorboard --logdir {logdir_path}


# add_graph의 파라미터는 tuple 형식을 받음. 따라서 인풋 데이터를 튜플 타입으로 변형
data0_tuple = tuple(data0[0][key] for key in data0[0].keys())

writer.add_graph(model, data0_tuple) # (모델, 입력 데이터)
```

학습 코드

```
def train(model, train_dataloader, valid_dataloader=None, epochs=2):

        loss_fct = CrossEntropyLoss()

        # train_dataloaer 학습을 epochs만큼 반복
        for epoch in range(epochs):
            print(f"*****Epoch {epoch} Train Start*****")
```

```
# 배치 단위 평균 loss와 총 평균 loss 계산하기위해 변수 생성
total_loss, batch_loss, batch_count = 0,0,0

# model을 train 모드로 설정 & device 할당
model.train()
model.to(device)

# data iterator를 돌면서 하나씩 학습
for step, batch in enumerate(train_dataloader):
    batch_count+=1

    # tensor 연산 전, 각 tensor에 device 할당
    batch = tuple(item.to(device) for item in batch)

    batch_input, batch_label = batch

    # batch마다 모델이 갖고 있는 기존 gradient를 초기화
    model.zero_grad()

    # forward
    logits = model(**batch_input)

    # loss
    loss = loss_fct(logits, batch_label)
    batch_loss += loss.item()
    total_loss += loss.item()

    # backward -> 파라미터의 미분(gradient)를 자동으로 계산
    loss.backward()

    # gradient clipping 적용
    clip_grad_norm_(model.parameters(), 1.0)

    # optimizer & scheduler 업데이트
    optimizer.step()
    scheduler.step()

    # 배치 10개씩 처리할 때마다 평균 loss와 lr를 출력
    if (step % 10 == 0 and step != 0):

        # 학습 loss 기록
        writer.add_scalar(
            tag = "Train Loss",
            scalar_value = batch_loss / batch_count,
            global_step = epoch * len(train_dataloader) + step
        )

        # 학습 learning rate 기록
        writer.add_scalar(
            tag = "Train LR",
            # y축
            scalar_value = optimizer.param_groups[0]['lr'],
            # x축
            global_step = epoch * len(train_dataloader) + step
        )

        # reset
        batch_loss, batch_count = 0,0

print(f"Epoch {epoch} Total Mean Loss : {total_loss/(step+1):.4f}")
print(f"*****Epoch {epoch} Train Finish*****\n")

if valid_dataloader is not None:
    print(f"*****Epoch {epoch} Valid Start*****")
    valid_loss, valid_acc = validate(model, valid_dataloader)

    # 검증 loss 기록
    writer.add_scalar(
        tag = "Valid Loss",
        scalar_value = valid_loss,
        global_step = epoch
    )

    # 검증 loss 기록
    writer.add_scalar(
        tag = "Valid Acc",
        scalar_value = valid_acc,
        global_step = epoch
    )

    print(f"*****Epoch {epoch} Valid Finish*****\n")
```

```
            # checkpoint 저장
            # save_checkpoint(".", model, optimizer, scheduler, epoch, total_loss/(step+1))

        print("Train Completed. End Program.")
```

텍스트 저장

```
predict_labels = np.argmax(probs, axis=1)
fn = 0
fp = 0

for wrong_idx in (labels != predict_labels).nonzero()[0]:
  if predict_labels[wrong_idx] == 0: # FN, 긍정인데 모델이 부정으로 분류함
    fn+=1
    writer.add_text(
        tag = "False Negative (0)",
        text_string = f"{data.test_dataset[wrong_idx][0]} Probability: {probs[wrong_idx][0]}",
        global_step = fn
    )
  else: # FP, 부정인데 모델이 긍정으로 분류함
    fp+=1
    writer.add_text(
        tag = "False Positive (1)",
        text_string = f"{data.test_dataset[wrong_idx][0]} Probability: {probs[wrong_idx][1]}",
        global_step = fp
    )
```

Week4-4

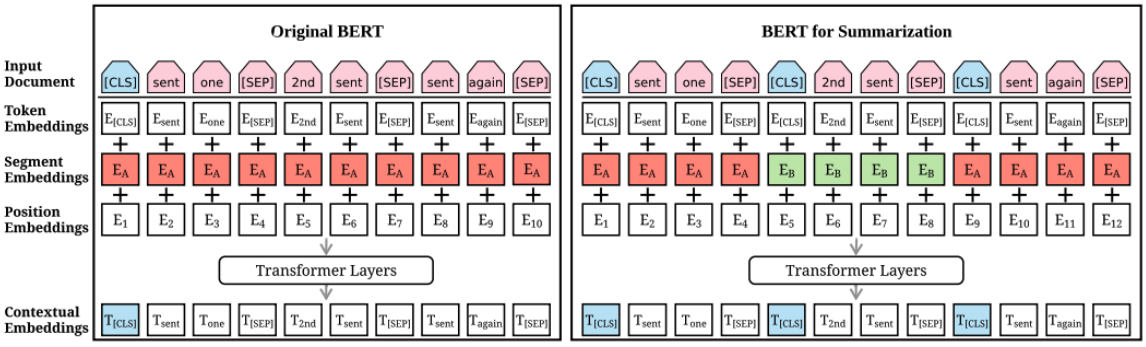
Extractive Summarization

참고 논문 및 소스 코드

- TextRank (2004)
  - 참고 한글 블로그
- BERTSUM (2019)
  - Paperhttps://arxiv.org/pdf/1908.08345v2.pdf
  - Performance

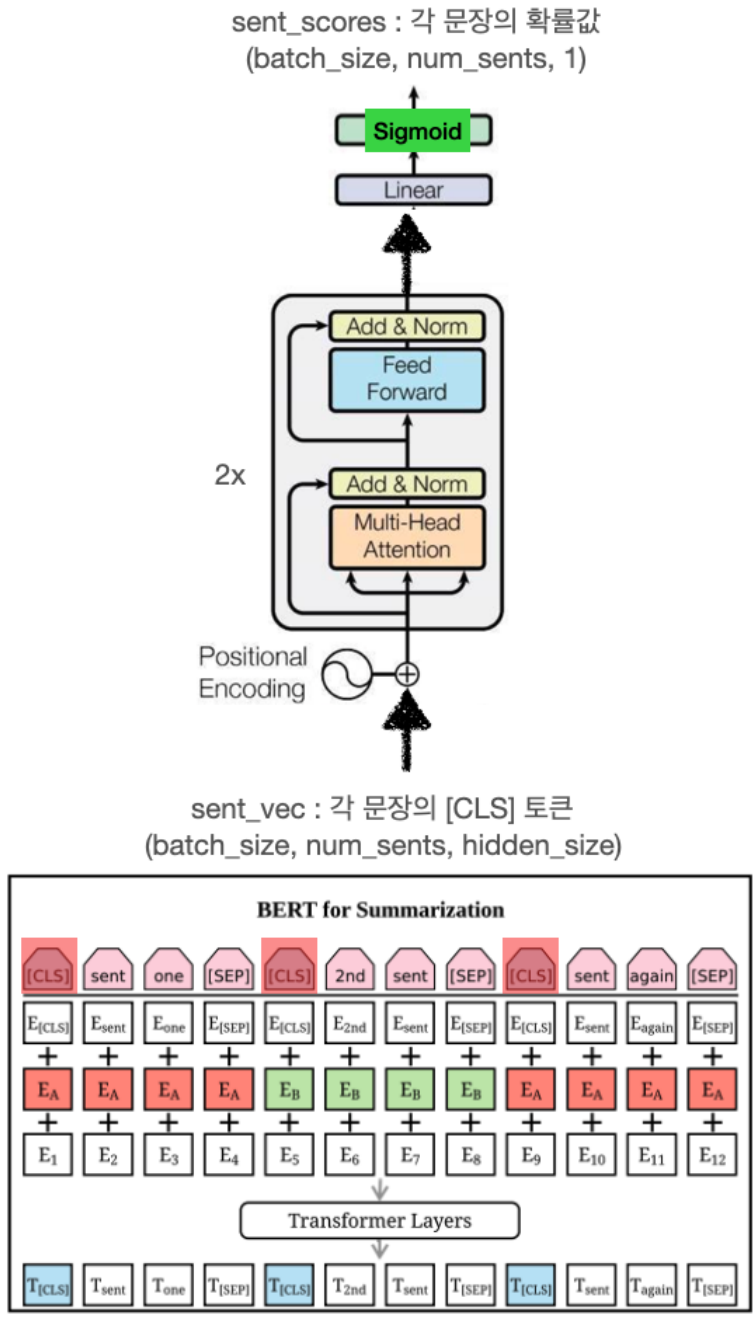
Model	R1	R2	RL
BertSumExt (BERT-base)	43.25	20.24	39.63
BertSumExt (BERT-large)	43.85	20.34	39.9

- ArchitectureDocument-level Encoder(BERT) 구조
  - 기존 BERT 모델을 사용해 문서의 각 문장이 summary 문장인지 아닌지를 판단하는 binary classification task를 수행함.
  - BERT의 input으로 문서(여러 문장의 집합)를 입력하기 위해 Embedding Layer의 구조를 약간 변화함.



- 위 그림에서 볼 수 있듯이 a) 모든 문장은 맨 앞에 “[CLS]” 토큰을 갖고 있으며 해당 토큰이 각 문장의 embedding 역할을 함 b) Segment embedding은 홀수 번째 문장은 EA, 짝수 번째 문장은 EB embedding을 부여해 홀수 번째와 짝수 번째 문장을 구분할 수 있게 함 c) Position embedding의 최대 길이 (기존 BERT는 512)을 늘림

BERTSUM 학습 방식



- 문서내 여러 문장을 “[CLS] 문장1 [SEP] [CLS] 문장2 [SEP] …” 구조로 BERT 모델에 입력해 마지막 layer에서 문장 별 [CLS] 토큰 값을 추출. (“sent_vec”)
- Transformer Encoder 2개를 합친 모델에 문장 별 [CLS] 토큰(“sent_vec”)을 입력. Transformer Encoder 위에 Linear Layer (shape: (768,1)) + Sigmoid 레이어를 쌓아 binary classifier를 구현. 결국 모델을 모두 통과하게 되면 output으로 문장 별 확률값이 나옴. (“sent_scores”)
- Sent_scores로 해당 문장이 summary 문장인지 아닌지 판별.
- Loss Function
  - Binary cross entropy loss
- Source Code

https://github.com/nlpyang/PreSumm

- Match Sum (2020)

- Paperhttps://arxiv.org/pdf/2004.08795v1.pdf
- Performance

Model	R1	R2	RL
MatchSum (BERT-base)	44.22	20.62	40.38
MatchSum (RoBERTa-base)	44.41	20.86	40.55

- Architecture
  - Siamese-BERT (RoBERTa) 학습 방식 하나의 BERT 모델에는 document(D)가, 다른 BERT 모델에는 summary 후보군 문장(C)이 입력됨.
  - 두 모델 (siamese network)은 서로 weight를 공유한다.
  - 두 모델은 모두 입력값의 embedding을 얻기 위해 마지막 layer의 첫 번째 토큰인 “[CLS]” 토큰을 사용한다.
  - 두 모델의 마지막 레이어의 “[CLS]” 토큰으로부터 각각 embedding을 구한 후 코사인 유사도를 계산한다.
  - document(D)와 코사인 유사도가 가장 높은 문장이 정답 summary가 된다.
- Loss Function
  - (Margin-based) Triplet Loss 계산 방식
    - 문서(D)와 gold summary 문장(gold summary, C*)간의 코사인 유사도를 f(D,C*)로 정의한다. (여기서 gold summary란 summary 후보군 문장 중 가장 정답 summary에 가까운 문장을 말한다.)
    - 문서(D)와 summary 후보군 문장(candidate summary, C)간의 코사인 유사도를 f(D,C)로 정의한다.
    - loss1을 아래 수식처럼 정의한다. 문서(D)와 gold summary(C*)의 거리가 문서(D)와 candidate summary(C)보다 더 가까워지도록 학습한다.

$$\mathcal{L}_1 = \max(0, f(D, C) - f(D, C^*) + \gamma_1), \quad (7)$$

    - loss2는 아래 수식처럼 정의한다. Loss1은 모든 candidate summary가 같은 margin값(r1)을 갖는다는 한계점이 있다. 이를 보강하기 위해 candidate summary중에서도 summary가 될 확률이 높은 문장은 작은 margin을 갖고 반대로 확률이 적은 문장은 높은 margin을 가져 문서와 거리가 더 멀어지도록 학습하는 loss2를 함께 사용한다. (i,j는 summary candidate 문장의 랭킹을 의미한다. Document(D)와 summary candidate(C)간 코사인 유사도를 구한 후 내림차순했을 때 summary candidate의 index를 랭킹이라고 정의한다. 따라서 i<j는 Ci summary 문장이 Cj summary 문장보다 랭킹이 높고 따라서 코사인 유사도가 높다고 보면 된다.)

      loss = loss1 + loss2

$$\mathcal{L}_2 = \max(0, f(D, C_j) - f(D, C_i) + (j - i) * \gamma_2) \quad (i < j),$$

- Source CodeGitHub - maszhongming/MatchSum: Code for ACL 2020 paper: "Extractive Summarization as Text Matching"

Huggingface 오픈 모델

huggingface에서 학습이 완료된 모델을 가져와 사용해볼 수 있다.

- URL

Pipelines

- 제공 모델
  - bart-large-cnn
  - t5-small
  - t5-base
  - t5-large
  - t5-3b
  - t5-11b