7주차

WMT 2016 번역모델 구현

7주차: Transformer를 활용한 번역 모델 구현

- 목적: Text classification과 같은 단일 label에 대한 학습이 아닌 machine translation처럼 문장을 생성할 수 있는 모델 구현
- 내용:
  - WMT2016 내 Multi-modal 데이터셋을 Custom Dataset을 활용하여 load
    (데이터: https://huggingface.co/datasets/bentrevett/multi30k)
  - 모델은 Transformer를 사용 (4주차 때 직접 구현한 Transformer or Huggingface 모델 불러와서 사용 → huggingface 라이브러리 활용시 pre-train되지 않은 모델을 불러와서 학습 진행)
  - Test 데이터셋에 대한 BLEU-4 Score 0.15 이상 목표
- 참고 사이트
  - WMT2016 내 Multi-modal 데이터셋
    https://www.statmt.org/wmt16/multimodal-task.html

WMT2016 번역 모델 구현

1. 데이터 로드

```python
#토크나이저
model_name = "facebook/mbart-large-50-many-to-many-mmt"
```

#⟨BOS⟩ 250004 / ⟨EOS⟩ 2 / ⟨PAD⟩ 1

```python
#데이터셋 로드
ds = load_dataset("bentrevett/multi30k")

# 데이터셋 클래스 정의
class TranslationDataset(Dataset):
    def __init__(self, dataset, tokenizer, max_len=128):
        self.dataset = dataset
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, idx):
        src_text = self.dataset[idx]['en']
        tgt_text = self.dataset[idx]['de']
        src_tokens = self.tokenizer(src_text, padding='max_length', truncation=True, max_length=self.max_len, return_tensors="pt", add_special_tokens=True)
        tgt_tokens = self.tokenizer(tgt_text, padding='max_length', truncation=True, max_length=self.max_len, return_tensors="pt", add_special_tokens=True)

        return {
            'src_input_ids': src_tokens['input_ids'].squeeze(),
            'tgt_input_ids': tgt_tokens['input_ids'].squeeze(),
            'src_attention_mask': src_tokens['attention_mask'].squeeze(),
            'tgt_attention_mask': tgt_tokens['attention_mask'].squeeze()
        }


#데이터셋-> 전처리된 데이터셋으로 변환
train_dataset = TranslationDataset(ds['train'],tokenizer)
val_dataset = TranslationDataset(ds['validation'],tokenizer)
test_dataset = TranslationDataset(ds['test'],tokenizer)

train_dataloader = DataLoader(train_dataset, batch_size=8, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=8, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=8, shuffle=False)
```

```
{'src_input_ids': tensor([250004,  32964,  27150,      4,  22392,  11280,      7,    621,  50782,
         43573,   5941,    373, 114942,      5,      2,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1]), 'tgt_input_ids': tensor([250004,  58320,  95350,  23739,     13,  63804,   1276,    566,  36443,
            33,     23,    122,  42231,   6179,     56,  57891,   7560,      5,
             2,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1,      1,      1,      1,      1,      1,      1,      1,
             1,      1]), 'src_attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0]), 'tgt_attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0])}
```

WMT2016 번역 모델 구현

2. 트랜스포머 모델 정의

```
SRC_VOCAB_SIZE = tokenizer.vocab_size
TGT_VOCAB_SIZE = tokenizer.vocab_size
EMB_SIZE = 512#512
NHEAD = 4#8
FFN_HID_DIM = 512#2048
BATCH_SIZE = 8
NUM_ENCODER_LAYERS = 2 #6
NUM_DECODER_LAYERS = 2 #6
```

```
class Seq2SeqTransformer(nn.Module):
    def __init__(self,
                 num_encoder_layers: int,
                 num_decoder_layers: int,
                 emb_size: int,
                 nhead: int,
                 src_vocab_size: int,
                 tgt_vocab_size: int,
                 dim_feedforward: int = 512,
                 dropout: float = 0.1):
        super(Seq2SeqTransformer, self).__init__()
        self.transformer = Transformer(d_model=emb_size,
                                       nhead=nhead,
                                       num_encoder_layers=num_encoder_layers,
                                       num_decoder_layers=num_decoder_layers,
                                       dim_feedforward=dim_feedforward,
                                       dropout=dropout)
        self.generator = nn.Linear(emb_size, tgt_vocab_size)#출력 생성기
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)#source 토큰 임베딩
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)#target 토큰 임베딩
        self.positional_encoding = PositionalEncoding(#위치 인코딩
            emb_size, dropout=dropout)
```

```
    #인코더-디코더 각 과정/훈련데이터 처리
    def forward(self,
                src: Tensor,
                trg: Tensor,
                src_mask: Tensor,
                tgt_mask: Tensor,
                src_padding_mask: Tensor,
                tgt_padding_mask: Tensor,
                memory_key_padding_mask: Tensor):
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        #source 문장을 처리한 후, target문장을 디코더로 처리?
        outs = self.transformer(src_emb, tgt_emb, src_mask, tgt_mask, None,
                                src_padding_mask, tgt_padding_mask, memory_key_padding_mask)#mask 적용
        return self.generator(outs)#최종적으로 각 타겟 단어에 대한 확률분포 반환/임베딩 차원->선형변환 적용->확률분포
```

```
    #추론(예측) 과정
    #인코딩
    def encode(self, src: Tensor, src_mask: Tensor):#source 문장 인코딩
        return self.transformer.encoder(self.positional_encoding(
                            self.src_tok_emb(src)), src_mask)
    #디코딩
    def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):#target 문장 디코딩
        return self.transformer.decoder(self.positional_encoding(
                          self.tgt_tok_emb(tgt)), memory,
                          tgt_mask)
```

```
#마스킹 생성
def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)#상삼각행렬 생성(순차적인 마스킹 적용)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))#-무한대로 마스킹 적용
    return mask

#마스크 설정
def create_mask(src, tgt):
    #[0]: 배치크기, [1]:시퀀스 길이
    tgt_seq_len = tgt.shape[0]-1#128-1


    #어텐션 masking(디코더)
    tgt_mask = generate_square_subsequent_mask(tgt_seq_len).type(torch.float32).to(DEVICE)

    #패딩 토큰에 대한 masking(인코더/디코더)
    src_padding_mask = (src == tokenizer.pad_token_id).to(torch.bool).transpose(0,1)#[src_seq_len, batch_size]
    tgt_padding_mask = (tgt[:-1,:] == tokenizer.pad_token_id).to(torch.bool).transpose(0,1)

    #차원 확인용
    #print('tgt_seq_len',tgt_seq_len, '  tgt_mask',tgt_mask.shape )
    #print('₩n',src_padding_mask.shape, '₩n',tgt_padding_mask.shape)

    return None, tgt_mask, src_padding_mask, tgt_padding_mask #src_mask는 반환하지 않음
```

https://tutorials.pytorch.kr/beginner/translation_transformer.html

WMT2016 번역 모델 구현

3. 학습 진행

1) 손실함수에 특정 토큰에 대한 가중치 적용

```
#손실 함수에 EOS 토큰 가중치 적용
def create_weighted_loss(vocab_size, eos_token_id, pad_token_id, device):
    weights = torch.ones(vocab_size).to(device)  #GPU로 옮김
    weights[eos_token_id] = 0.1 #<EOS> 토큰에 대한 가중치를 낮춤
    #weights[pad_token_id] = 0.0#<PAD> 토큰은 무시

    return nn.CrossEntropyLoss(weight=weights, ignore_index=pad_token_id)
```

```
loss_fn = create_weighted_loss(vocab_size=tokenizer.vocab_size, eos_token_id=eos_token_id, pad_token_id=pad_token_id, device=DEVICE)
```

2) 자동 혼합 정밀도(Amp, Automatic Mixed Precision)

모델을 학습 및 추론 시, 일부 연산을 16-bit(반정밀도, FP16)로 처리,
중요한 연산은 32-bit(단정밀도, FP32)로 처리하여 **성능을 최적화**

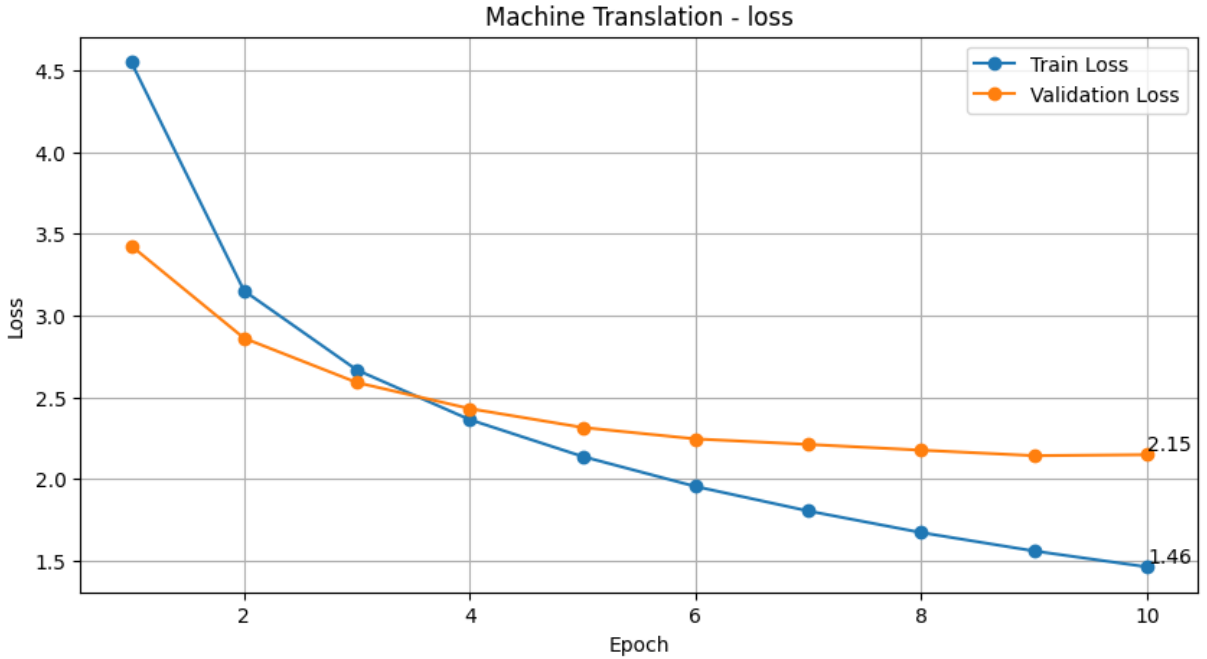
```
with amp.autocast():
    #모델의 출력 계산
    output = model(src, tgt[:-1, :], None, tgt_mask, src_padding_mask, tgt_padding_mask, src_padding_mask)

    #손실 계산 (output을 [batch_size, sequence_len, vocab_size]로 변환)
    tgt_out = tgt[1:, :]  #<BOS> 토큰 제외
    loss = loss_fn(output.reshape(-1, output.shape[-1]), tgt_out.reshape(-1))
```

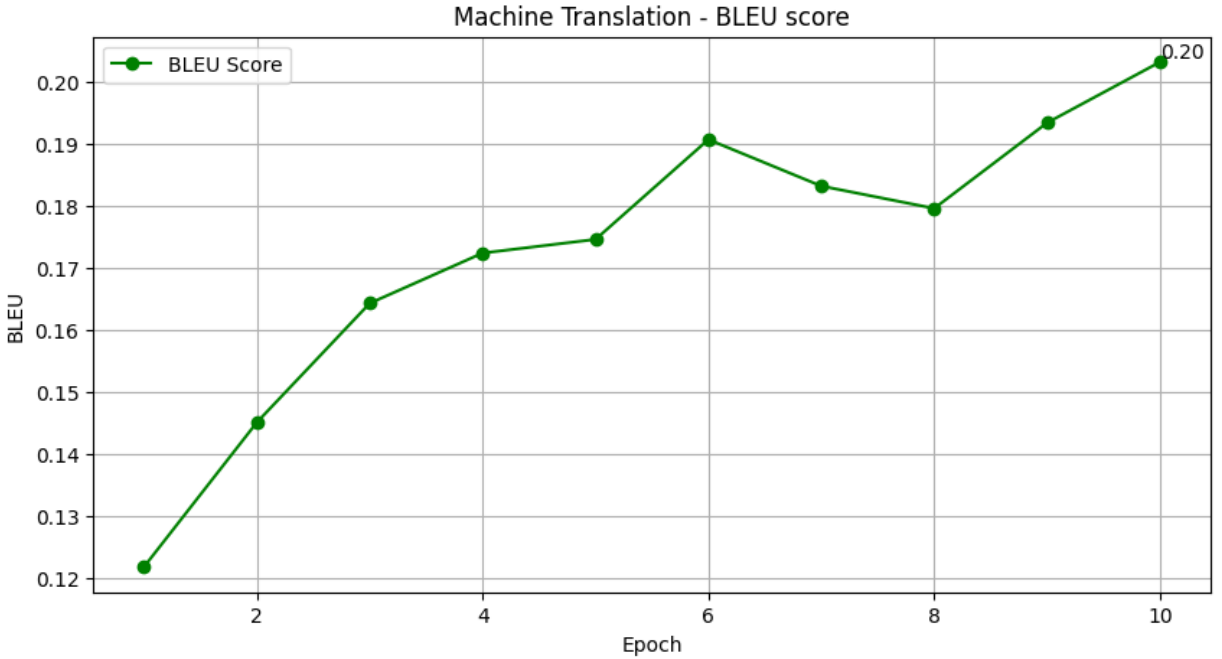
WMT2016 번역 모델 구현

4. 학습 결과

TEST Result

```
Evaluating on Test dataset :
Test loss: 2.0736761302947997, Test BLEU-4: 0.200507296805846
```





5. Inference

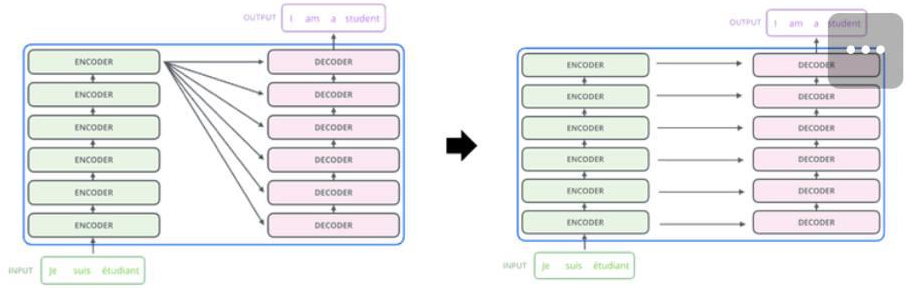
```
sentence : I love you.
translated_sentence : mit einem linghelm.
```

8주차

P-Transformer 모델 구현

8주차: P-Transformer 모델 구현

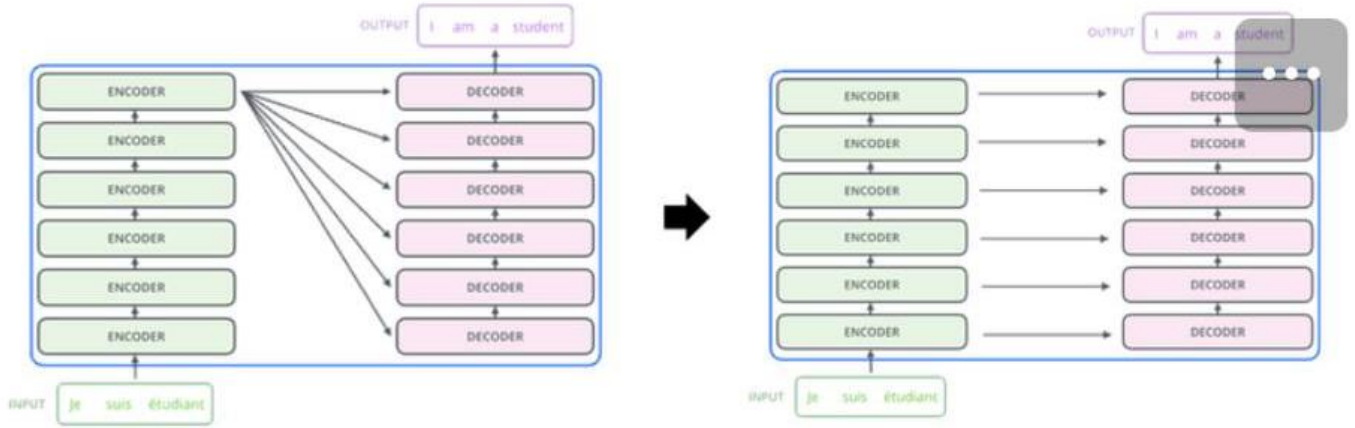
- 목적: Transformer모델을 원하는 방식대로 변형해서 쓸 수 있도록 구현 연습
- 내용:
  - Original Transformer를 P-Transformer로 변형



(왼쪽: Original Transformer; 오른쪽: Parallel Transformer (P-Transformer)

  - 5주차와 동일한 데이터셋에 대해서도 BLEU-4 Score 0.15 이상 목표

P-Transformer



(왼쪽: Original Transformer; 오른쪽: Parallel Transformer (P-Transformer)

1. 멀티 GPU를 활용한 트랜스포머 병렬 학습

각 레이어를 다른 GPU에 배치하여
동시에 여러 레이어를 병렬로 연산

2. 인코더와 디코더의 병렬학습 진행

인코더와 디코더의 동시 처리.
인코더에서 모든 입력을 처리하기 전에
디코더가 일부 연산을 미리 시작

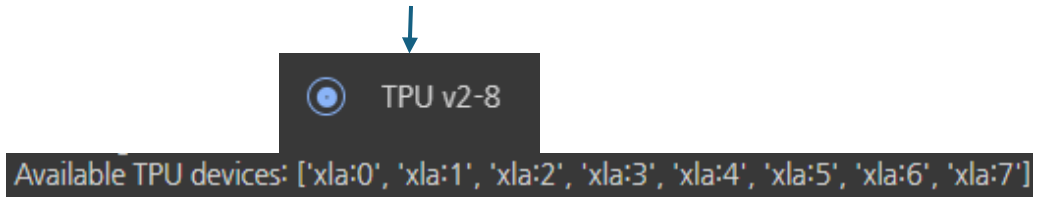
1. 멀티 TPU를 활용한 병렬 학습

1. DataParallel

배치 데이터를 **자동으로 여러 GPU에 나누어 할당**

```
model = nn.DataParallel(transformer)#병렬로 처리
model = model.to(DEVICE)
```

batch = 8 / TPU 코어(8개) = **1개 샘플씩 처리**



2. TPU 병렬 처리

```
for batch in dataloader:
    batch = prepare_batch(batch, device)#데이터 로드 및 분배

    # 데이터 로드
    src = batch['src_input_ids'].to(device).transpose(0,1)
    tgt = batch['tgt_input_ids'].to(device).transpose(0,1)

    #마스크 생성
    src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt)

    optimizer.zero_grad()

    #모델 출력
    output = model(src, tgt[:-1, :], None, tgt_mask, src_padding_mask, tgt_padding_mask, src_padding_mask)

    #손실 계산 (output을 [batch_size, sequence_len, vocab_size]로 변환)
    tgt_out = tgt[1:, :]  # <sos> 토큰 제외
    loss = loss_fn(output.reshape(-1, output.shape[-1]), tgt_out.reshape(-1))

    #역전파 및 파라미터 업데이트
    loss.backward()
    xm.optimizer_step(optimizer)# 병렬처리된 각 코어의 기울기를 동기화, 파라미터 업데이트
    xm.mark_step()# TPU 메모리 동기화

    total_loss += loss.item()

    #BLEU-4 스코어 계산
    output_ids = output.argmax(dim=-1).transpose(0,1)#예측 토큰 -> [batch_size, sequence_length]
    avg_bleu = calculate_bleu(output_ids, tgt[1:, :].transpose(0, 1), tokenizer)  # BLEU 계산
    total_bleu += avg_bleu

    #tqdm 업데이트
    pbar.update(1)

#평균계산
avg_loss = total_loss/len(dataloader)
avg_bleu = total_bleu/len(dataloader)

return avg_loss, avg_bleu
```

1. 멀티 TPU를 활용한 병렬 학습

```
Epoch 1/3
Training Epoch 1/3: 100%|████████████████| 3625/3625 [2:05:56<00:00,  2.08s/batch]
  Train loss: 4.230305880974079, Train Bleu: 0.07396006449481067
TPU Metrics after training epoch:
Metric: DeviceLockWait
  TotalSamples: 7928
  Accumulator: 043ms727.429us
  ValueRate: 004.976us / second
  Rate: 0.950413 / second
  Percentiles: 1%=001.695us; 5%=001.823us; 10%=001.908us; 20%=002.006us; 50%=007.092us; 80%=008.193us; 90%=008.560us; 95%=008.859us; 99%=011.356us
Counter: RegisterXLAFunctions
  Value: 1
Counter: MarkStep
  Value: 3964

Current TPU Device: xla:0
Validation Epoch 1/3: 100%|████████████████| 127/127 [01:08<00:00,  1.85batch/s]
  Validation loss: 3.395407136031023, Validation Bleu: 0.12358588276849701

TPU Metrics after validation epoch:
Metric: DeviceLockWait
  TotalSamples: 7930
  Accumulator: 043ms741.416us
  ValueRate: 004.690us / second
  Rate: 0.895039 / second
  Percentiles: 1%=001.695us; 5%=001.823us; 10%=001.908us; 20%=002.006us; 50%=007.092us; 80%=008.197us; 90%=008.560us; 95%=008.862us; 99%=011.399us
Counter: RegisterXLAFunctions
  Value: 1
Counter: MarkStep
  Value: 3965

Current TPU Device: xla:0
```

샘플을 처리하는 데 소요된 누적 시간

TPU가 초당 처리한 샘플 수

TPU에서 작업을 처리하는 데 걸린 시간을 퍼센타일로 나눈 값
(**99%의 작업**이 **11.356 마이크로초** 안에 완료되었다)

1. 멀티 TPU를 활용한 병렬 학습

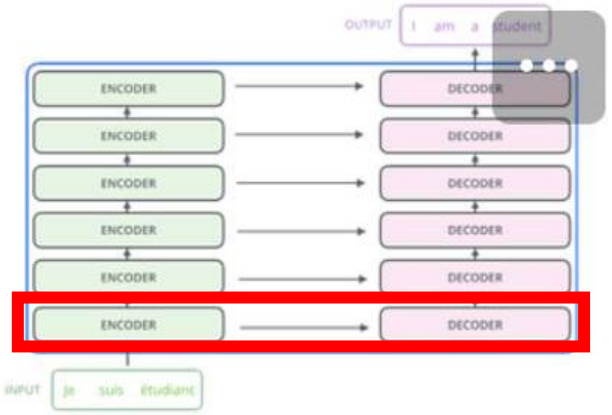
TEST Result

```
Evaluating on Test dataset :
Test loss: 2.5256736736297607, Test BLEU-4: 0.1553880677634969
```

Inference

```
sentence : You will face many defeats in life, but never let yourself be defeated.
translated_sentence : T-Shirt mit Blumenladen.
```

2. 인코더와 디코더의 병렬학습 진행



- 인코더의 첫 번째 레이어가 완료된 후, 즉시 디코더 시작
- 인코더의 나머지 레이어들이 병렬적으로 실행되며, 디코더도 계속해서 동작

- 디코더의 입력은 Teacher Forcing 사용

```
#파이프라인으로 처리
#인코더-디코더 각 과정/훈련데이터 처리
def forward(self,
        src: Tensor,
        trg: Tensor,
        src_mask: Tensor,
        tgt_mask: Tensor,
        src_padding_mask: Tensor,
        tgt_padding_mask: Tensor,
        memory_key_padding_mask: Tensor,
        teacher_forcing_ratio=1.0):#교사강요 비율

  #1. 인코더 첫번째 레이어
  src_emb = self.positional_encoding(self.src_tok_emb(src))
  memory = self.transformer_encoder.layers[0](src_emb, src_key_padding_mask=src_padding_mask)

  # 2. 디코더의 첫번째 레이어
  tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
  tgt_output = torch.zeros_like(tgt_emb).to(tgt_emb.device)#예측값을 저장할 tensor

  #첫번째 단어 입력 (<BOS>)
  tgt_output[:, 0, :] = self.transformer_decoder.layers[0](tgt_emb[:, 0, :], memory, tgt_mask=tgt_mask, memory_key_padding_mask=memory_key_padding_mask)#.unsqueeze(1)

  #나머지 단어 입력 (교사강요)
  for i in range(1, trg.size(1)):  # (1, 문장길이)
     for layer_idx in range(len(self.transformer_decoder.layers)):  # 모든 디코더 레이어에 대해 반복
        if torch.rand(1).item() < teacher_forcing_ratio:  # 교사강요 여부 결정
           # 교사강요: 정답 토큰 사용
           tgt_output[:, i, :] = self.transformer_decoder.layers[layer_idx](tgt_emb[:, i, :], memory, tgt_mask=tgt_mask, memory_key_padding_mask=memory_key_padding_mask)
        else:
           # 이전 예측값 사용
           tgt_output[:, i, :] = self.transformer_decoder.layers[layer_idx](tgt_output[:, i - 1, :], memory, tgt_mask=tgt_mask, memory_key_padding_mask=memory_key_padding_mask)

  #3. 인코더 나머지 레이어 병렬처리
  for i in range(1,len(self.transformer_encoder.layers)):
    memory = self.transformer_encoder.layers[i](memory, src_key_padding_mask=src_padding_mask)
  #4. 디코더 나머지 레이어 병렬처리
  for i in range(1,len(self.transformer_decoder.layers)):
    tgt_output  = self.transformer_decoder.layers[i](tgt_output, memory, tgt_mask=tgt_mask, memory_key_padding_mask=memory_key_padding_mask)

  return self.generator(tgt_output)
```

결과

1. [7주차] – 번역 모델 구현 완료.

   –〉 좋지 않은 번역 품질

2. [8주차] – Parallel Transformer 학습

   –〉 ‘2번 방법’에 대해 차주 구현하여 발표희망, 간단논세 병행