Homework #3

Report

- LSTM, Attention, Transformer 비교 실험 -

전공 컴퓨터공학전공

이름 한사랑

학번 2271064

1. 서론

본 보고서에서는 영어-프랑스어(EN-FR) 기계번역을 위해 LSTM, 어텐션 기반 LSTM, 그리고 트랜스포머(Transformer) 모델을 구현하고, 각 모델의 구조와 성능을 비교 분석합니다.
특히 자연어처리(NLP) 파이프라인의 전처리, 토크나이저 구축, 데이터셋 구성부터, 각 모델별 학습 및 검증 루프, 그리고 실제 번역 결과까지의 전 과정이 구현되었습니다.

구현된 주요 요소는 다음과 같습니다:

- SentencePiece 기반 서브워드 토크나이저 및 데이터 전처리
- LSTM, LSTM with Attention, Transformer 모델 클래스 정의
- 각 모델별 학습 및 검증 루프
- 번역 sample 출력 및 모델 파라미터/연산량 요약

실험에는 IWSLT 2017 EN-FR 병렬 코퍼스(학습 5만 문장, 검증 1천 문장)가 사용되었습니다.
모든 실험은 PyTorch 라이브러리를 활용하였고, Google Colab - T4 GPU 환경에서 수행되었습니다.

전체 코드가 담긴 깃허브 저장소 링크:

https://github.com/Sarang-Han/Artificial-Intelligence/tree/main/HW3

2. 실험 설계

2.1. LSTM Baseline 구현

1. Encoder(nn.Module)

```
class Encoder(nn.Module):
    def __init__(self, vocab, hidden):
        super().__init__()

        # ---- TODO students implement below ---- #
        # Token ID → embedding vector conversion
        self.embedding = nn.Embedding(vocab, hidden)
        # LSTM layer with batch_first=True
        self.lstm = nn.LSTM(input_size=hidden,
                            hidden_size=hidden,
                            num_layers=lstm_layers,
                            dropout=lstm_dropout if lstm_layers > 1 else 0,
                            batch_first=True)
        self.dropout = nn.Dropout(lstm_dropout)

    def forward(self, x):
        emb = self.embedding(x)  # [B, T] -> [B, T, H]
        emb = self.dropout(emb)
        outputs, (h_n, c_n) = self.lstm(emb)
        return outputs, (h_n, c_n)
```

2. Decoder(nn.Module)

```
class Decoder(nn.Module):
    def __init__(self, vocab, hidden):
        super().__init__()

        # ---- TODO students implement below ---- #
        self.embedding = nn.Embedding(vocab, hidden)
        self.lstm = nn.LSTM(input_size=hidden,
                            hidden_size=hidden,
                            num_layers=lstm_layers,
                            dropout=lstm_dropout if lstm_layers > 1 else 0,
                            batch_first=True)
        self.dropout = nn.Dropout(lstm_dropout)
        # Output layer: maps hidden state to vocab-sized logits
        self.fc = nn.Linear(hidden, vocab)

    def forward(self, y, hidden):
        emb = self.embedding(y)  # [B, T] -> [B, T, H]
        emb = self.dropout(emb)
        # Use encoder's hidden state as initial state
        outputs, (h_n, c_n) = self.lstm(emb, hidden)
        outputs = self.dropout(outputs)
        logits = self.fc(outputs)  # [B, T, H] -> [B, T, V]
        return logits, (h_n, c_n)
```

2.2. LSTM with Attention 구현

```
# ---------------------------------------------------------------
# Scaled dot-product Attention (single-head, batched)
# ---------------------------------------------------------------
class Attention(nn.Module):
    def __init__(self, hidden):
        super().__init__()
        self.W = nn.Linear(hidden, hidden, bias=False)

    def forward(self, decoder_hidden, encoder_out):
        # ---- TODO students implement below ---- #

        # 1. Project decoder hidden through self.W
        query = self.W(decoder_hidden)  # [B, 1, H]

        # 2. Dot-product with encoder_out^T via torch.bmm
        # scores = Q · K^T
        scores = torch.bmm(query, encoder_out.transpose(1, 2))  # [B, 1, T_src]

        # 3. Softmax over T_src dimension to turn scores → probs
        attn_weights = F.softmax(scores, dim=-1)  # [B, 1, T_src]

        # 4. Weighted sum (context) = probs · V via torch.bmm
        context = torch.bmm(attn_weights, encoder_out)  # [B, 1, H]
        return context, attn_weights

# ---------------------------------------------------------------
# Attention-augmented Decoder (one token at a time)
# ---------------------------------------------------------------
class AttnDecoder(nn.Module):
    def __init__(self, vocab, hidden):
        super().__init__()
        # ---- TODO students implement below ---- #

        self.embedding = nn.Embedding(vocab, hidden)
        self.lstm = nn.LSTM(input_size=hidden,
                            hidden_size=hidden,
                            num_layers=lstm_layers,
                            dropout=lstm_dropout if lstm_layers > 1 else 0,
                            batch_first=True)
        self.attention = Attention(hidden)
        self.dropout = nn.Dropout(lstm_dropout)
        self.out_dropout = nn.Dropout(lstm_dropout)
        self.fc = nn.Linear(hidden, vocab)

    def forward(self, y, hidden, enc_outputs):
        # ---- TODO students implement below ---- #

        # 1. Embed y and apply dropout
        emb = self.embedding(y)  # [B, T_dec, H]
        emb = self.dropout(emb)

        batch_size = emb.size(0)
        seq_len = emb.size(1)
        hidden_size = emb.size(2)
```

```
        # Storage for collecting outputs
        outputs = []

        # 2. & 3. Process each timestep to use previous decoder state
        for t in range(seq_len):
            # Current timestep embedding
            emb_t = emb[:, t:t+1, :]  # [B, 1, H]

            # LSTM step
            lstm_out, hidden = self.lstm(emb_t, hidden)  # [B, 1, H]

            # Apply attention mechanism
            context, _ = self.attention(lstm_out, enc_outputs)

            # Combine context and LSTM output (simple addition)
            combined = lstm_out + context  # [B, 1, H]

            # Store result
            outputs.append(combined)

        # 4. Concatenate all outputs
        outputs = torch.cat(outputs, dim=1)  # [B, T_dec, H]

        # 5. Apply output dropout and projection
        outputs = self.out_dropout(outputs)
        logits = self.fc(outputs)  # [B, T_dec, vocab]

        # 6. Return logits and final hidden state
        return logits, hidden
```

기본 LSTM 구조에 어텐션 메커니즘을 추가하여, Decoder가 매 시점마다 Encoder의 전체 출력(hidden states) 중에서 중요한 부분에 집중할 수 있도록 구현되었습니다.

`Attention` 클래스

- 디코더의 현재 hidden state를 선형 변환하여 쿼리로 사용하고, 인코더의 모든 hidden state와 dot-product를 통해 어텐션 score를 계산함
- Softmax를 적용해 attention weights를 얻고, 이를 인코더 출력에 곱해 context vector를 만듦

`AttnDecoder` 클래스

- 입력 토큰을 임베딩 후, 각 시점마다 LSTM 셀을 한 번씩 실행함
- 각 시점의 LSTM 출력과 인코더 출력을 Attention 모듈에 입력하여 컨텍스트 벡터를 계산하고, 이를 LSTM 출력과 더해 최종 출력으로 사용함
- 모든 시점의 출력을 모아 최종적으로 선형 계층을 거쳐 각 토큰의 예측 확률을 산출함

2.3. Transformer 구현

```
# ----------------------------------------------------------------
# PositionalEncoding - sinusoidal schedule explained
# ----------------------------------------------------------------
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model) # [T, D]

        # Positions: 0, 1, 2, ..., T-1  → shape [T, 1] so broadcasting works
        pos = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) # 0,2,4,...
                             * -(math.log(10000.0) / d_model) # exponent factor
                             ) # shape [D/2]

        pe[:, 0::2] = torch.sin(pos * div_term) # even indices  (0,2,...)
        pe[:, 1::2] = torch.cos(pos * div_term) # odd  indices  (1,3,...)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
# ----------------------------------------------------------------
# TransformerModel - step-by-step TODOs
# ----------------------------------------------------------------
class TransformerModel(nn.Module):
    def __init__(self, vocab, d_model=256, nhead=4, nlayers=2):
        super().__init__()
        # -------- TODO students implement below -------- #
        # 1. Token embedding with `padding_idx=PAD_ID` and embed_dim = d_model
        self.embed = nn.Embedding(vocab, d_model, padding_idx=PAD_ID)

        # 2. PositionalEncoding instance (no learnable params)
        self.pos = PositionalEncoding(d_model)

        # 3. Encoder-decoder stack
        # Encoder layers with batch_first=True
        enc_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=4*d_model,
            batch_first=True
        )
        # Stack encoder layers together
        self.encoder = nn.TransformerEncoder(enc_layer, nlayers)

        # Decoder layers with batch_first=True
        dec_layer = nn.TransformerDecoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=4*d_model,
            batch_first=True
        )
        # Stack decoder layers together
        self.decoder = nn.TransformerDecoder(dec_layer, nlayers)
```

```
        # 4. Final linear layer maps D → vocab logits
        self.fc = nn.Linear(d_model, vocab)
    def forward(self, src, tgt,
                src_key_padding_mask=None,
                tgt_key_padding_mask=None,
                tgt_mask=None,
                memory_key_padding_mask=None):

        # -------- TODO students implement below -------- #
        # 1. Embed + add positional encodings
        src_emb = self.pos(self.embed(src))  # [B, T_src, D]
        tgt_emb = self.pos(self.embed(tgt))  # [B, T_tgt, D]

        # If memory padding mask not provided, default to src padding mask
        if memory_key_padding_mask is None:
            memory_key_padding_mask = src_key_padding_mask

        # 2. Encoder produces memory [B, T_src, D]
        memory = self.encoder(
            src_emb,
            src_key_padding_mask=src_key_padding_mask
        )

        # 3. Decoder consumes (tgt, memory) and returns hidden states [B, T_tgt, D]
        # Pass all masking arguments to ensure padding & causality
        out = self.decoder(
            tgt_emb,
            memory,
            tgt_mask=tgt_mask,
            tgt_key_padding_mask=tgt_key_padding_mask,
            memory_key_padding_mask=memory_key_padding_mask
        )
        # 4. Project decoder outputs through self.fc → logits [B, T_tgt, vocab]
        logits = self.fc(out)

        # 5. Return logits
        return logits
```

Transformer 구조는 RNN이나 CNN 없이 어텐션 메커니즘만으로 시퀀스 간의 관계를 학습할 수 있도록 구현되었습니다.

`PositionalEncoding` 클래스

- 토큰 임베딩에 사인/코사인 기반의 위치 정보를 더해, 순서 정보가 없는 어텐션 구조에 위치 정보를 제공

`TransformerModel` 클래스

- 입력 시퀀스가 임베딩과 포지셔널 인코딩을 거쳐 인코더에 전달됨
- 인코더와 디코더는 각각 여러 층의 Transformer 레이어로 구성되고, 디코더에는 causal mask와 패딩 마스크를 적용해 미래 토큰을 참조하지 못하도록 함
- 디코더의 출력을 선형 계층에 통과시켜 각 시점별로 전체 단어 집합에 대한 예측 확률을 산출함

3. 실험 결과

3.1. 모델별 결과 요약

실험에서는 총 세 가지 모델(LSTM, LSTM with Attention, Transformer)을 동일한 데이터셋에 대해 15 에포크 동안 학습했습니다. 또한 학습 과정에서 출력된 train loss, val loss, val ppl 데이터를 기반으로, 각 모델의 학습 과정과 최종 성능을 시각화하였습니다.

사용한 시각화 코드:
https://github.com/Sarang-Han/Artificial-Intelligence/blob/main/HW3/Visualize.ipynb

1. 주요 결과 요약

최종 검증 성능 순위

- 1위: **LSTM with Attention** (Val PPL: 17.11)
- 2위: **Transformer** (Val PPL: 26.21)
- 3위: **LSTM** (Val PPL: 59.20)

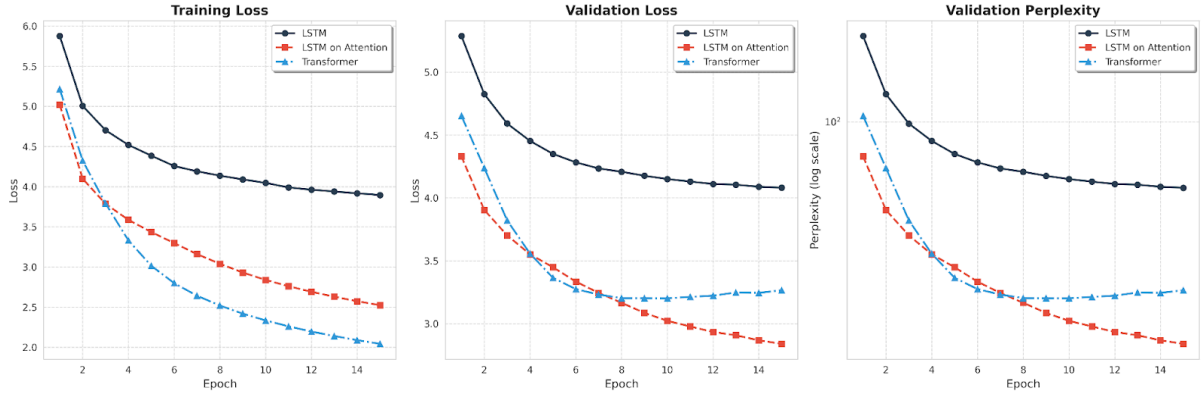
모델별 특징

- **LSTM**: 학습이 느리고 최종 성능이 가장 낮음
- **LSTM with Attention**: 안정적인 학습 곡선과 높은 검증 성능
- **Transformer**: 훈련 데이터에서 가장 빠른 학습과 낮은 훈련 손실을 보이지만, 후반부에 과적합을 보임

성능 차이 원인 분석:

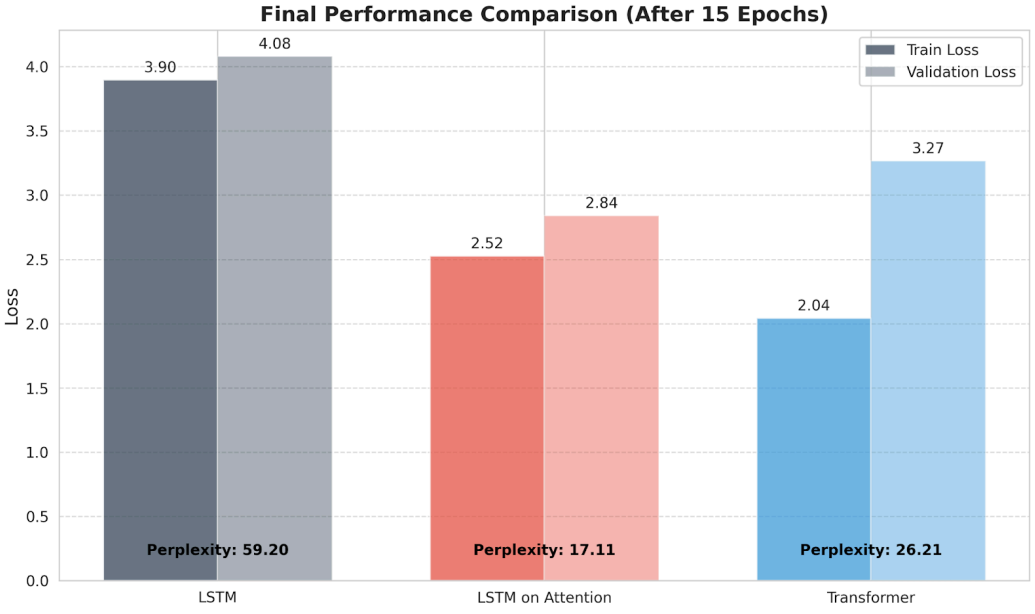
- Attention 메커니즘이 번역 작업에서 문맥 이해에 결정적 역할을 함
- Transformer는 훈련 데이터에 과적합되는 경향이 있어 검증 성능이 LSTM with Attention보다 떨어짐
- 일반 LSTM은 번역 작업의 복잡한 의존성을 포착하는 능력이 제한적으로 보여짐

2. 모델 학습 곡선 비교



- **일반 LSTM vs 다른 모델들**: 일반 LSTM은 다른 두 모델에 비해 전반적으로 손실이 높고 수렴 속도가 느렸음. 15 에포크 후에도 손실이 크게 감소하지 않았음.
- **학습 속도**: LSTM with Attention과 Transformer는 초기 에포크에서 손실이 급격히 감소하였고, 특히 Transformer가 학습 데이터에서 가장 빠르게 손실을 줄이는 경향을 보였음.
- **과적합 현상**: Transformer 모델은 약 8 에포크 이후부터 훈련 손실은 계속 감소하는 반면, 검증 손실이 약간 증가하는 과적합 현상을 보임.
- **LSTM with Attention의 안정성**: LSTM with Attention 모델은 학습이 진행됨에 따라 검증 손실이 꾸준히 감소하여 가장 안정적인 학습 곡선을 보임.

3. 최종 성능 비교



3.2. 번역 Sample 비교

영어 source	프랑스어 target
A person is wearing a hat.	Une personne porte un chapeau.

모델	번역 결과	주요 오류	정확도
LSTM	itablement que c'est un peu	- 단어 선택 오류 - 문장 구조 부적절 - 의미 전달 실패	**매우 낮음** 원문 내용과 무관한 번역
LSTM with Attention	personne ne s'est mis un pied.	- 부정문으로 변환 - 핵심 객체 변경 (모자→발) - 동사 선택 부적절	**중간** 주어(personne)는 맞지만 의미가 변형됨 "사람이 발을 놓지 않았다" 뜻으로 변형
Transformer	mons un chapitre.	- 주어 오류 - 핵심 객체 변경 (모자→장) - 문장 불완전	**낮음** "우리는 한 장(chapter)을" 정도로 번역되어 원문과 관련 없음

3.3. 종합적 분석 및 논의

1. 연산량 및 번역 성능 관점에서의 효율성

모델	총 연산량(Mult-Adds)	검증 PPL	효율성
LSTM	1,498.53 MB	59.20	낮음
LSTM with Attention	1,528.94 MB	17.11	중간
Transformer	22.97 MB	26.21	높음

Transformer 모델은 다른 두 모델에 비해 연산량이 약 1/65 수준으로 낮으면서도 LSTM보다 훨씬 우수한 번역 성능을 보여줌. 비록 LSTM with Attention보다 검증 PPL이 다소 높지만, 연산량 대비 성능을 고려하면 Transformer가 가장 효율적인 모델이라고 볼 수 있음. 반면, LSTM with Attention은 가장 높은 번역 정확도를 보여주지만 이를 달성하기 위해 LSTM과 비슷한 수준의 높은 연산량이 필요함. 특히 어텐션 메커니즘이 추가되면서 추가적인 연산 부담이 발생했지만, 성능적으로는 우수함을 알 수 있음.

2. 모델 크기 및 번역 성능 관점에서의 효율성

모델	파라미터 수	모델 크기(MB)	검증 PPL	효율성
LSTM	62,672,800	251.52	59.20	낮음
LSTM with Attention	63,721,376	255.16	17.11	중간
Transformer	33,525,664	83.67	26.21	높음

Transformer는 파라미터 수가 가장 적으면서도(LSTM 대비 약 53%), 모델 크기도 가장 작음(LSTM 대비 약 33%). 동시에 LSTM보다 훨씬 우수한 번역 성능을 달성하여, 모델 크기 대비 성능 측면에서 매우 효율적임. LSTM with Attention은 가장 많은 파라미터를 가지고 있지만, 이를 통해 가장 높은 성능을 보여줌. 그러나 모델 크기가 크기 때문에 자원의 제약이 있는 환경에서는 적합하지 않을 수 있음.

3. 학습 시간 및 번역 성능 관점에서의 효율성

모델	학습 시간(분)	검증 PPL	효율성
LSTM	81.06	59.20	낮음
LSTM with Attention	109.66	17.11	중간
Transformer	47.83	26.21	높음

Transformer는 병렬 처리가 가능한 구조 덕분에 학습 시간이 가장 짧으면서도 LSTM보다 훨씬 우수한 번역 성능을 달성함. LSTM with Attention은 가장 긴 학습 시간이 필요하지만, 최고의 번역 성능을 보여주어 학습 시간과 성능 간의 trade-off가 있음.

4. 결론

본 실험에서는 기계번역 작업에 사용되는 세 가지 주요 신경망 아키텍처(LSTM, LSTM with Attention, Transformer)를 IWSLT 2017 영어-프랑스어 데이터셋에 적용하고, 그 성능과 효율성을 다양한 관점에서 비교 분석했습니다. 각 모델의 훈련/검증 손실, 복잡도(Perplexity), 학습 시간, 파라미터 수, 연산량 등 여러 지표를 측정하여 종합적인 평가를 수행했습니다.

실험의 주요 관찰 결과는 다음과 같습니다:

번역 성능 관점

- LSTM with Attention 모델이 가장 낮은 검증 복잡도(17.11)를 달성하여 번역 정확도 측면에서 최고 성능을 보여줌
- 기본 LSTM은 가장 높은 복잡도(59.20)로 번역 품질이 현저히 낮음
- Transformer는 중간 수준의 복잡도(26.21)로 준수한 번역 성능

효율성 관점

- Transformer 모델이 가장 적은 연산량(22.97MB), 가장 작은 모델 크기(83.67MB), 가장 짧은 학습 시간(47.83분)으로 효율성 측면에서 우위
- LSTM과 LSTM with Attention은 모델 크기와 연산량이 약 3배 이상 크며, Transformer보다 훨씬 긴 학습 시간 필요
- LSTM with Attention은 추가된 어텐션 메커니즘으로 인해 LSTM보다 약 35% 더 긴 학습 시간 소요

모델 특성

- LSTM: 시퀀스 모델링의 기본 구조로서 의미가 있으나, 효율성과 성능 모두 낮음
- LSTM with Attention: 어텐션 메커니즘의 도입으로 번역 품질이 크게 향상되었으나, 계산 효율성 측면에서 비용이 높음
- Transformer: 병렬 처리가 가능한 구조로 효율성이 뛰어나며, 중간 수준의 성능으로 자원 제약 환경에서 적합함