KOREATECH 한국기술교육대학교

7 주차 과제

학번	2020136087
이름	윤아현
수강 과목	자연어처리
담당 교수	임상훈 교수님
제출일	2024.04.30

문제 및 해결 방법 (구현 코드)

1. ENCODER 개선 (30 점)

실습수업에 사용한 Seq-to-Seq 모델의 Encoder 를 개선하시오.
* 합리적 이유에 기반해 개선 방법을 찾고 구현 및 실험 하시오.
-여러 제약사항(컴퓨팅, 메모리 등)이 있으므로 꼭 성능이 높아져야 하는 것은 아님*
* 왜 그런 모델 구성을 생각했는지, 그 결과가 어떻게 나타났는지 기술하시오.
* 성능이 높아졌다면 왜 그렇다고 생각하는지, 낮아졌다면 무엇이 문제인 것 같은지.

Hint
* 꼭 Encoder 의 구조가 RNN 계열의 모델이어야 하는가?
* Bi-directional RNN 을 사용한다면?

GRADING
적용한 방법 1 개당 (+15)

Encoder 개선 방법
1 번. 현재는 단방향 RNN model 을 사용하고 있으므로, 단방향의 문맥 정보만 파악이 가능하다.
양방향 RNN model 을 통해서, 이전 시점과 이후 시점의 문맥 정보를 모두 파악할 수 있도록 한다.
2 번. LSTM 은 여러 게이트(input, forget, output, gate gate)로 구성되어 있는데, 이는 모델을 복잡하게 만든다.
LSTM 모델보다 더 간단한 구조를 가지고 있는 GRU 모델을 사용해본다.

Code

```
class ResGRU_Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.gru = nn.GRU(emb_dim, hid_dim, n_layers, dropout=dropout, bidirectional=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        outputs, hidden = self.gru(embedded)
        hidden = self.reformat_hidden(hidden, num_directions=2)

        if outputs.size(-1) == embedded.size(-1):
            outputs += embedded

        return outputs, hidden

    def reformat_hidden(self, hidden, num_directions):
        new_shape = (self.gru.num_layers, num_directions, hidden.size(1), hidden.size(2))
        hidden = hidden.view(*new_shape)
        hidden = torch.cat([hidden[:, i, :, :] for i in range(num_directions)], dim = 2)
        return hidden
```

Code 설명

양방향 GRU 모델을 사용하였다.

이에 추가로, 기존의 embedding Vector 에 outputs(최종 출력된 context vectors)를 더하여 기울기 소실 문제를 완화시키는 **잔차 연결 기법을** 사용하였다.

GRU 모델은 cell state 가 없고 outputs 와 최종 hidden vector 만 반환한다.

2. DECODER 개선 (30 점)

* 실습수업에 사용한 Seq-to-Seq 모델의 Decoder 를 개선하시오.
* 합리적 이유에 기반해 개선 방법을 찾고 구현 및 실험 하시오
* 여러 제약사항(컴퓨팅, 메모리 등)이 있으므로 꼭 성능이 높아져야 하는 것은 아님
* 왜 그런 모델 구성을 생각했는지, 그 결과가 어떻게 나타났는지 기술하시오
* 성능이 높아졌다면 왜 그렇다고 생각하는지, 낮아졌다면 무엇이 문제인 것 같은지

* Hint
* 최종 output 을 만들 때 마지막 layer 의 hidden vector 만 사용하는게 최선인가? 이전 layer 의 hidden vector 도 같이 사용한다면?
* 왜 encoder 와 decoder 의 크기 차이가 많이 발생하는가? 이를 해결할 수 없는가?
* 현재 Encoder 의 마지막 hidden vector 를 사용하는데 대부분 [PAD] 토큰이다. [PAD] 토큰의 hidden vector 를 사용하는게 맞는가?

GRADING
* 적용한 방법 1 개당 (+15)

Decoder 개선 방법

1 번. 현재 Context Vector 는 Encoder 의 마지막 Layer 을 사용하여 Encoder Output 을 출력한다. 이 대신 모든 은닉층에 대해 평균을 내어 Context Vector 를 생성한다.

2 번. Encoder 와 동일하게 복잡한 모델을 GRU 모델로 경량화한다.

Code

```python
class AttentionDecoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        self.output_dim = output_dim
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim + hid_dim*2, hid_dim*2, n_layers, dropout=dropout)
        self.fc_out = nn.Linear(hid_dim*4, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        input = input.unsqueeze(0)  # [1, batch_size]
        embedded = self.dropout(self.embedding(input))

        hidden_last = hidden[-1].unsqueeze(0).transpose(0, 1)  # [batch_size, 1, hidden_size]
        print(hidden_last.shape)
        attention_scores = torch.bmm(hidden_last, encoder_outputs.permute(1, 2, 0)) # [batch_size, 1, seq_len]
        # attention 분포 생성
        attention_distribution = torch.softmax(attention_scores.squeeze(1), dim=1)
        # context 벡터 계산
        context = torch.bmm(attention_distribution.unsqueeze(1), encoder_outputs.permute(1, 0, 2)).squeeze(1)

        # GRU 실행
        rnn_input = torch.cat((embedded.squeeze(0), context), dim=1).unsqueeze(0)
        output, hidden = self.rnn(rnn_input, hidden)

        # 최종 단어 예측
        prediction = self.fc_out(torch.cat((output.squeeze(0), context), dim=1))

        return prediction, hidden
```

```python
class AttentionDecoder2(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        self.output_dim = output_dim
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim + hid_dim*2, hid_dim*2, n_layers, dropout=dropout)
        self.fc_out = nn.Linear(hid_dim*4, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        input = input.unsqueeze(0)  # [1, batch_size]
        embedded = self.dropout(self.embedding(input))

        hidden_all = hidden.transpose(0, 1)
        hidden_mean = hidden_all.mean(dim=1, keepdim=True)

        attention_scores = torch.bmm(hidden_mean, encoder_outputs.permute(1, 2, 0)) # [batch_size, 1, seq_len]
        # attention 분포 생성
        attention_distribution = torch.softmax(attention_scores.squeeze(1), dim=1)
        # context 벡터 계산
        context = torch.bmm(attention_distribution.unsqueeze(1), encoder_outputs.permute(1, 0, 2)).squeeze(1)

        # GRU 실행
        rnn_input = torch.cat((embedded.squeeze(0), context), dim=1).unsqueeze(0)
        output, hidden = self.rnn(rnn_input, hidden)

        hidden_mean = hidden_mean.squeeze(1)

        prediction = self.fc_out(torch.cat((hidden_mean, context), dim=1))

        return prediction, hidden
```

Code 설명

AttentionDecoder2 를 기준으로,

Encoder 와 동일하게 RNN model 대신 GRU 모델을 사용하였다.

마지막 은닉층만 사용하는 것이 아닌, 여러 은닉층의 평균을 사용하여 Context Vector 를 만든다. 또한, 은닉층과 Context Vector 차원 모두 고려하여 더 많은 정보를 모델 출력에 반영한다.

Encoder and Decoder Output (AttentionDecoder2 기준)

```
test_data2 = test_dataset.__getitem__(400)
a2 = model.decode(test_data2["src"])

input2 = " ".join([list(en_vocab.keys())[list(en_vocab.values()).index(i)] for i in test_data2["src"]])
target2 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in test_data2["trg"]])
model_output2 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in a2])

print("400번째 index의 입력값 (english) :", input2) # 입력
print("400번째 index의 정답값 (france) :", target2) # 정답
print("400번째 index의 예측값 (france) :", model_output2) # 모델 예측값
```

```
400번째 index의 입력값 (english) : that s what i thought at first . [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
400번째 index의 정답값 (france) : [SOS] c est ce que j ai pense tout d abord . [EOS] [PAD] [PAD] [PAD] [PAD] [PAD] [
400번째 index의 예측값 (france) : [SOS] c est ce que je ai ai . . . [EOS]
```

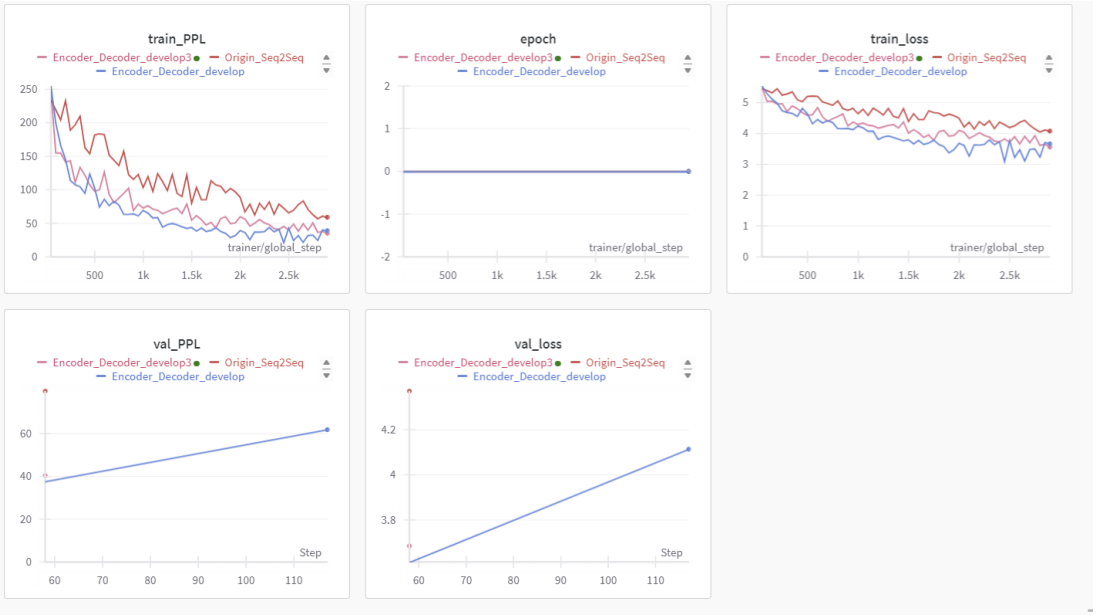
```
test_data3 = test_dataset.__getitem__(260)
a3 = model.decode(test_data3["src"])

input3 = " ".join([list(en_vocab.keys())[list(en_vocab.values()).index(i)] for i in test_data3["src"]])
target3 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in test_data3["trg"]])
model_output3 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in a3])

print("400번째 index의 입력값 (english) :", input3) # 입력
print("400번째 index의 정답값 (france) :", target3) # 정답
print("400번째 index의 예측값 (france) :", model_output3) # 모델 예측값
```

```
400번째 index의 입력값 (english) : i think we re going to have a very strong team . [PAD] [PAD] [PAD] [PAD] [PAD]
400번째 index의 정답값 (france) : [SOS] je pense que nous allons disposer d une equipe tres forte . [EOS] [PAD] [
400번째 index의 예측값 (france) : [SOS] je pense que nous nous une un un de . . . [EOS]
```

Origin Seq2Seq VS 개선한 Seq2Seq(Encoder_Decoder_develop) 1, 2 성능평가 Graph



GRU 모델과 모든 은닉층 평균을 통한 Context Vector 를 만드는 것보다, GRU 모델과 마지막 은닉층만을 사용하였을 때 더 좋은 성능을 보였다.

왜 그런지 생각을 해보았을 때, 마지막 은닉층에는 최종적으로 나온 (가장 최근) 정보를 포함하고 있기 때문에 이 은닉층이 제일 많은 정보를 포함하며 가장 관련성이 높은 문맥을 가지고 있을 것이다. 은닉층의 평균 같은 경우, 중간 시점에서 중요하지 않은 정보가 중요한 정보로 잘못 나오는 경우가 발생할 수도 있을 것이다. 그렇기 때문에 중간 시점의 은닉층도 포함하여 최종 은닉층을 만드는 것보다 마지막 은닉층만을 사용하는 것이 더 좋을 수도 있다.

3. ENC-DEC 개선 (40 점)

* 실습수업에 사용한 Seq-to-Seq 모델의 Encoder-Decoder 의 연결부분을 개선하시오.
* 합리적 이유에 기반해 개선 방법을 찾고 구현 및 실험 하시오
* 여러 제약사항(컴퓨팅, 메모리 등)이 있으므로 꼭 성능이 높아져야 하는 것은 아님
* 왜 그런 모델 구성을 생각했는지, 그 결과가 어떻게 나타났는지 기술하시오
* 성능이 높아졌다면 왜 그렇다고 생각하는지, 낮아졌다면 무엇이 문제인 것 같은지

* Hint
* Attention 을 개선할 수 없을까? (Dot attention 을 QKV attention 으로 개선, weighted attention 등)
* Enc-DEC 의 layer 수가 다른 경우는 어떻게 처리할 것인가?

GRADING
* 적용한 방법 1 개당 (+15) (최대 40 점)

Encoder-Decoder 개선 방법

1 번. 단순한 Dot Product Attention 이 아닌, Query, key, Value 를 사용한 Attention 을 사용하였다. Decoder 가 Encoder 에 Query 를 날려, Decoder 가 필요한 정보, 중요한 정보에 더 잘 접근할 수 있도록 하였다.

2 번. 만일 Encoder 와 Decoder 의 Layer 수가 다르면, Encoder 의 은닉 상태를 Decoder 의 레이어 수와 크기에 맞게 변환해주는 State_mapper 기능을 사용하였다.

Code

```python
class QKVAttention(nn.Module):
    def __init__(self, enc_dim, dec_dim):
        super().__init__()
        self.key_layer = nn.Linear(enc_dim, dec_dim)
        self.query_layer = nn.Linear(dec_dim, dec_dim)
        self.value_layer = nn.Linear(enc_dim, dec_dim)
        self.scale = torch.sqrt(torch.FloatTensor([dec_dim])).to('cuda')

    def forward(self, hidden, encoder_outputs):
        keys = self.key_layer(encoder_outputs)
        queries = self.query_layer(hidden)
        values = self.value_layer(encoder_outputs)

        queries = queries.unsqueeze(1)
        keys_transposed = keys.permute(1, 2, 0)
        energy = torch.bmm(queries, keys_transposed) / self.scale
        attention = torch.softmax(energy, dim=2)

        values_permuted = values.permute(1, 0, 2)
        weighted = torch.bmm(attention, values_permuted)
        return weighted, attention
```

```python
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout=dropout, bidirectional=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        outputs, (hidden, cell) = self.rnn(embedded)
        return outputs, hidden, cell
```

```python
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, n_layers, dropout, attention):
        super().__init__()
        self.output_dim = output_dim
        self.attention = attention
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.LSTM(enc_hid_dim + emb_dim, dec_hid_dim, n_layers, dropout=dropout)
        self.fc_out = nn.Linear(enc_hid_dim + dec_hid_dim + emb_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell, encoder_outputs):
        input = input.unsqueeze(0)  # [1, batch_size]
        embedded = self.dropout(self.embedding(input))
        attention_output, _ = self.attention(hidden[-1], encoder_outputs)

        attention_output = attention_output.permute(1, 0, 2)

        rnn_input = torch.cat((embedded, attention_output), dim=2)
        output, (hidden, cell) = self.rnn(rnn_input, (hidden, cell))
        output_squeezed = output.squeeze(0)
        prediction = self.fc_out(torch.cat((output_squeezed, attention_output.squeeze(0), embedded.squeeze(0)), dim=1))
        return prediction, hidden, cell
```

```python
class Decoder2(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, n_layers, dropout, attention):
        super().__init__()
        self.output_dim = output_dim
        self.attention = attention
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.state_mapper = nn.Linear(enc_hid_dim, dec_hid_dim * n_layers)  # 상태 변환
        self.rnn = nn.LSTM(emb_dim + enc_hid_dim, dec_hid_dim, n_layers, dropout=dropout)
        self.fc_out = nn.Linear(enc_hid_dim + dec_hid_dim + emb_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell, encoder_outputs):
        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))

        if hidden.shape[0] != self.rnn.num_layers:
            # 인코더의 은닉 상태를 디코더의 레이어 수에 맞게 조정
            print("Hidden.shape : ", hidden.shpape)
            print("rnn.num_layers :", self.rnn.num_layers)
            print("OMG Layer Different..")
            hidden = self.state_mapper(hidden.transpose(0, 1)).transpose(0, 1).view(self.rnn.num_layers, -1, self.rnn.hidden_size)
            cell = self.state_mapper(cell.transpose(0, 1)).transpose(0, 1).view(self.rnn.num_layers, -1, self.rnn.hidden_size)

        attention_output, _ = self.attention(hidden[-1], encoder_outputs)
        attention_output = attention_output.permute(1, 0, 2)
        rnn_input = torch.cat((embedded, attention_output), dim=2)

        output, (hidden, cell) = self.rnn(rnn_input, (hidden, cell))
        output_squeezed = output.squeeze(0)
        prediction = self.fc_out(torch.cat((output_squeezed, attention_output.squeeze(0), embedded.squeeze(0)), dim=1))
        return prediction, hidden, cell
```

Code 설명

Q, K, V 를 사용하여 Encoder 에서는 key, value 를, Decoder 에서는 Query 를 수행하였다. 이를 통해 나온 가중합 (Weighted Sum)을 Decoder 로 보내 연산을 수행하였다.

또한, Decoder 에서 현재의 Embedding Token 과 Encoder 의 마지막 은닉 상태를 고려해서 가중 평균을 계산하였다. 이 두 가지를 모두 사용하여, 중요한 부분에 대한 정보를 동시에 고려하였다

Encoder and Decoder Output (Decoder2 기준)

```python
test_data1 = test_dataset.__getitem__(150)
test_data1["src"] = test_data1["src"].to('cuda')
test_data1["trg"] = test_data1["trg"].to('cuda')
a1 = model.decode(test_data1["src"])

input1 = " ".join([list(en_vocab.keys())[list(en_vocab.values()).index(i)] for i in test_data1["src"]])
target1 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in test_data1["trg"]])
model_output1 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in a1])

print("234번째 index의 입력값 (english) :", input1) # 입력
print("234번째 index의 정답값 (france) :", target1) # 정답
print("234번째 index의 예측값 (france) :", model_output1) # 모델 예측값
```

```
234번째 index의 입력값 (english) : do you like roller [UNK] ? [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [
234번째 index의 정답값 (france) : [SOS] est ce que tu aimes les montagnes [UNK] ? [EOS] [PAD] [PAD] [PAD] [PAD]
234번째 index의 예측값 (france) : [SOS] tu vous tu ? ? ? ? ? ? ? ? ? ? ? ? ? [EOS]
```

```python
test_data2 = test_dataset.__getitem__(300)
test_data2["src"] = test_data2["src"].to('cuda')
test_data2["trg"] = test_data2["trg"].to('cuda')
a2 = model.decode(test_data2["src"])

input2 = " ".join([list(en_vocab.keys())[list(en_vocab.values()).index(i)] for i in test_data2["src"]])
target2 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in test_data2["trg"]])
model_output2 = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in a2])

print("400번째 index의 입력값 (english) :", input2) # 입력
print("400번째 index의 정답값 (france) :", target2) # 정답
print("400번째 index의 예측값 (france) :", model_output2) # 모델 예측값
```

```
400번째 index의 입력값 (english) : you re not very good . [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
400번째 index의 정답값 (france) : [SOS] vous n etes pas tres bonne . [EOS] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [
400번째 index의 예측값 (france) : [SOS] vous n ne pas ne . . . . . [EOS]
```

Seq2Seq(Encoder_Decoder_develop) VS Q, K, V 를 활용한 Seq2Seq 성능평가 Graph

출력된 Output 을 보면, 단어가 제대로 출력되지 않는 것을 볼 수 있다. (물음표 혹은 점이 많이 나오는 오류 발생) 코드를 구현할 때, cuda 관련 오류가 계속 발생하였는데, Loss 값은 제대로 떨어지는 것을 보니 cuda 의 문제로 인해 제대로 예측이 안 된 것 같다. Q, K, V Attention mechanism 을 사용하였을 때 제일 좋은 성능이 나올 것이라고 생각하였는데, 실제로는 Encoder_Decoder 를 각각 개선했을 때 더 좋은 성능이 나온 것을 볼 수 있다. GRU 모델 및 양방향 학습이 모델 성능에 많은 영향을 미친 것 같다.

또한, 예상을 해보았을 때 Query, Key, Value 라는 여러 가중치들에 대한 학습을 수행해야 하기 때문에 epoch 를 1 번 돌림으로써 얻을 수 있는 모델의 성능이 좋지 못했을 거라고 추측한다.