Neural Machine Translation
by Jointly Learning to Align and Translate

120220225 조유빈

1. 논문 내용 요약

A. Traditional machine translation 모델의 문제점

i. Source sentence 정보를 고정된 길이의 context vector로 나타내야 함

ii. Source sentence의 길이가 길 경우에는 고정된 길이로 정보를 압축하는 것이 bottleneck 역할을 하여 long sentence를 처리하는데 어려움이 존재

B. 저자가 제안하는 사항

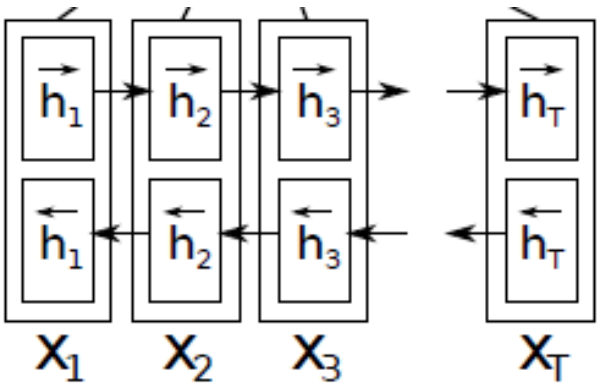
i. 예측할 target word와 관련된 source sentence의 일부 정보를 모델이 자동으로 찾을 수 있도록 설계

ii. Attention mechanism을 사용하여 decoder가 어떤 source sentence에 집중해야 하는지 결정하도록 함

- ✓ Encoder는 source sentence의 모든 정보를 고정된 길이의 벡터로 표현해야 할 필요 없음
- ✓ 다음 target 단어의 생성과 관련 있는 정보들(특정 source positions와 이전 states에서 예측된 target words)에만 집중하도록 함

2. 수식 및 네트워크 설명 (with 코드 분석)

A. Encoder



```python
# Building the Seq2Seq Model - Encoder
"""
Bidirectional RNN : the annotation of each word to summarize not only the preceding words, but also the following words
z = s0 (hidden) : concatenating two context vectors (a forward and a backward one) together => initial hidden state of decoder
"""
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        # src : [src len, batch size]

        # Embedding the input src
        embedded = self.dropout(self.embedding(src)) # [src len, batch size, emb dim]
```

i. Bidirectional RNN (BiRNN)을 통해 forward hidden states $(\vec{h}_1, \vec{h}_2, \cdots, \vec{h}_T)$와 backward hidden states $(\overleftarrow{h}_1, \overleftarrow{h}_2, \cdots, \overleftarrow{h}_T)$를 얻음

```python
        # Perform Bidirectional RNN
        outputs, hidden = self.rnn(embedded) # [src len, batch size, hid dim * num directions] , [n layers * num directions, batch size, hid dim]

        # hidden is stacked [forward_1, backward_1, forward_2, backward_2, ...]
        # outputs are always from the last layer
```

- ✓ Forward RNN은 source sentence를 정방향 순서로 읽어내고 backward RNN은 역방향 순서로 읽어 냄
- ✓ BiRNN을 통해 단어 x_i의 이전에 위치한 단어의 정보와 이후에 위치한 단어의 정보를 모두 활용하여 인코딩할 수 있음
- ✓ 단어 x_i의 주변 단어들에 집중할 수 있음
- ✓ 출력값
  1. output : 각 i번째 forward hidden states와 backward hidden states이 결합되어 담긴 $H = \{ h_1, h_2, \cdots, h_T \}$ ($h_i = [\vec{h}_i ; \overleftarrow{h}_i]$)
     - Shape : (입력 문장 내 단어 수, batch size, 2 * hidden_dim)
  2. hidden : layer의 forward hidden states와 backward states가 담긴 stack, 즉, $[\vec{h}_T, \overleftarrow{h}_T]$
     - Shape : (2 * layer 수, batch size, hidden_dim)

ii. Decoder의 initial hidden state s_0로 사용되는 context vector 생성

```python
        # hidden [-2, :, : ] is the last of the forwards RNN
        # hidden [-1, :, : ] is the last of the backwards RNN

        # initial decoder hidden is final hidden state of the forwards and backwards
        # encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))) # [batch size, dec hid dim]

        # outputs : H = {h1, h2, ... , hT} , h1 = [h1->,h1<-] ...

        return outputs, hidden
```

- ✓ BiRNN에서 마지막 states에 대하여 두 개(forward & backward)의 hidden states가 얻어졌지만, decoder는 bidirectional 구조가 아니므로 single context vector가 필요

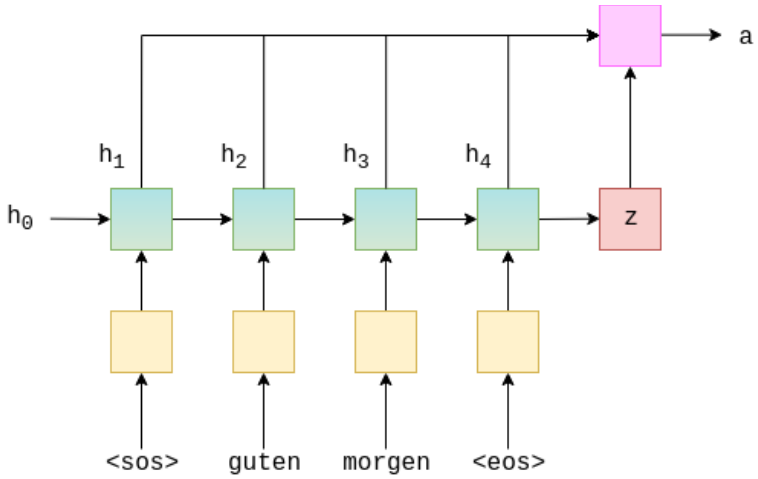
✓ 마지막 state의 두 hidden states를 결합하여 아래 식을 통해 decoder의 initial hidden state s_0를 얻음

$$s_0 = \tanh\left(g\left(concat\left(\vec{h}_T, \overleftarrow{h}_T\right)\right)\right), \qquad (\, g(\cdot) : \, linear\ layer\,)$$

✓ 출력값

1. hidden : decoder로 들어가는 initial hidden state s_0
   - Shape : (batch size, decoder hidden_dim)

B. Attention




```python
# Attention mechanism
"""
Inputs : the previous hidden state of the decoder s_(t-1) and all of the stacked hidden states from encoder H.
Outputs : an attention vector a_t that is the length of the source sentence.
          Each element is between 0 and 1 and the entire vector sums to 1.

a_t represents which words in the source sentence we should pay the most attention to
in order to correctly predict the next word to decode y_(t+1).
"""
class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim)
        self.v = nn.Linear(dec_hid_dim, 1, bias = False)

    def forward(self, hidden, encoder_outputs):
        # hidden (z) : [batch size, dec hid dim]
        # encoder_outputs : [src len, batch size, enc hid dim * 2]

        batch_size = encoder_outputs.shape[1]
        src_len = encoder_outputs.shape[0]

        # repeat decoder hidden state src_len times
        # encoder_outputs와 shape를 맞춰주기 위함
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1) # [batch size, src len, dec hid dim] s0

        encoder_outputs = encoder_outputs.permute(1, 0, 2) # [batch size, src len, enc hid dim * 2]

        # Calculate energy between the previous decoder hidden state and the encoder hidden states
        energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim = 2))) # [batch size, src len, dec hid dim]

        attention = self.v(energy).squeeze(2) # [batch size, src len]

        return F.softmax(attention, dim=1)
```

i. Attention 과정은 decoder가 다음 단어 y_i를 올바르게 예측하기 위해 source sentence의 어느 위치에 주목해야 하는지를 구하는 과정임

ii. Decoder의 이전 hidden state s_{i-1}와 모든 encoder hidden states H를 입력 받아 아래와 같은 수식을 통해 energy E_i를 구함

$$E_i = \tanh\big(attn(concat(s_{i-1}, H))\big), \quad (attn(\cdot) : linear\ layer)$$

✓ E_{ij}는 이전 hidden state s_{i-1}에 대한 encoder hidden states h_j의 중요도를 반영함

✓ 출력값

1. energy E_i : 이전 hidden state s_{i-1}에 대한 각 encoder hidden states의 중요도를 계산한 값
   - Shape : (batch_size, 입력 문장 내 단어 수, decoder hidden_dim)

iii. (1, decoder hidden_dim) 형태의 tensor v를 E_i에 곱하여 attention 벡터 $\hat{a}_i$를 구함

$$\hat{a}_i = vE_i$$

✓ 출력값

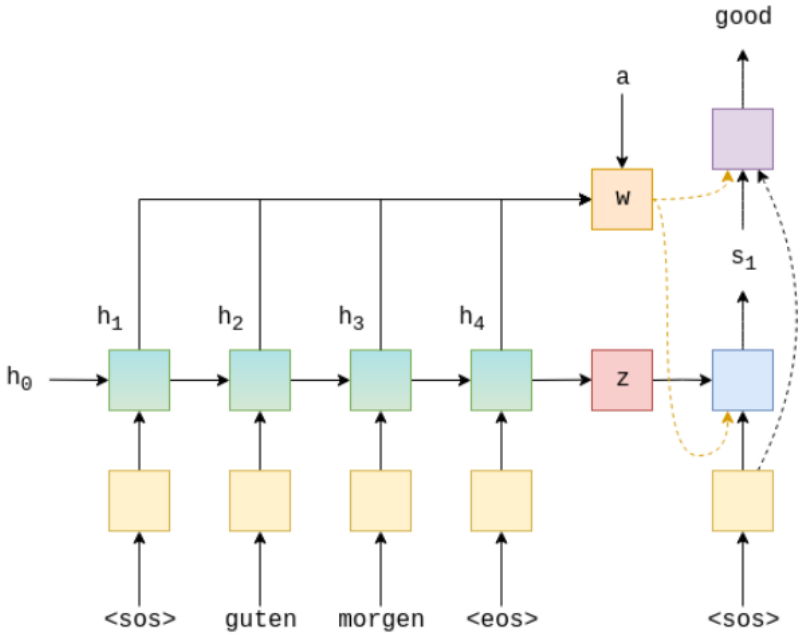
1. attention 벡터 $\hat{a}_i$: 각 encoder hidden states에 대한 energy E_{ij}에 weighted sum을 적용한 값
   - Shape : (batch_size, 입력 문장 내 단어 수)

iv. Attention vector $\hat{a}_i$에 softmax를 취하여 최종적인 attention vector a_i를 구함

$$a_i = softmax(\hat{a}_i) = \frac{\exp(\hat{a}_i)}{\sum_{k=1}^{T} \exp(\hat{a}_{ik})}$$

✓ a_i의 각 요소들은 모두 0과 1사이의 값을 갖게 되고, 모든 요소들의 합은 1이 됨

C. Decoder




```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()

        self.output_dim = output_dim
        self.attention = attention

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim)

        self.fc_out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        # input : [batch size]
        # hidden : [batch size, dec hid dim]
        # encoder_outputs : [src len, batch size, enc hid dim * 2]

        input = input.unsqueeze(0) # [1, batch size]
```

i. Decoder의 입력 토큰(즉, 이전 예측 단어 y_{i-1})을 embedding 함

```
        embedded = self.dropout(self.embedding(input)) # [1, batch size, emb dim]
```

ii. 이전 hidden state s_{i-1}와 모든 encoder hidden states H를 받아 앞서 설명한 attention 과정을 통해 attention vector a_i를 구함

```
        # Get an attention vector
        a = self.attention(hidden, encoder_outputs) # [batch size, src len]
        a = a.unsqueeze(1) # [batch size, 1, src len]
```

iii. attention vector a_i와 모든 encoder hidden states H의 행렬 곱 연산을 통해 weighted source vector w_i를 구함

$$w_i = c_i = \sum_{j=1}^{T} a_{ij} h_j$$

```
# Matrix 연산을 통해 weighted source vector w_i를 구함. context vector c_i와 같음
weighted = torch.bmm(a, encoder_outputs) # [batch size, 1, enc hid dim * 2]
weighted = weighted.permute(1, 0, 2) # [1, batch size, enc hid dim * 2]
```

- ✓ weighted source vector w_i는 context vector c_i와 같음
- ✓ 즉, source sentence의 정보를 지닌 벡터

iv. weighted source vector w_i는 embedded input word $d(y_{i-1})$와 결합되어 RNN으로 들어감

$$s_i = RNN(s_{i-1}, d(y_{i-1}), c_i)$$

```
rnn_input = torch.cat((embedded, weighted), dim = 2) # [1, batch size, (enc hid dim * 2) + emb dim]

# 현재 hidden state s_i 구함
output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))
# [seq len, batch size, dec hid dim * n directions], [n layers * n directions, batch size, dec hid dim]
# seq len, n layers and n directions will always be 1 in this decoder, therefore:
# output : [1, batch size, dec hid dim]
# hidden : [1, batch size, dec hid dim]
# this also means that output == hidden
assert (output == hidden).all()
```

- ✓ 이때 이전 hidden state s_{i-1}도 RNN으로 들어가 이전 단어에 대한 정보를 넘겨줌
- ✓ 출력값
  1. output : 현재 hidden state s_i
     - Shape : (1, batch size, decoder hidden_dim)
  2. hidden : output 값과 동일한 s_i값을 가짐

v. Embedded input word $d(y_{i-1})$, weighted source vector w_i와 현재 hidden state s_i를 모두 결합하여 fully connected layer를 통과시켜 다음에 올 예측 단어 $\hat{y}_i$를 구함

$$\hat{y}_i = f(d(y_{i-1}), w_i, s_i), \qquad (f : linear\ layer)$$

```
# 다음에 올 단어 y_i 예측
prediction = self.fc_out(torch.cat((output, weighted, embedded), dim = 1)) # [batch size, output dim]

return prediction, hidden.squeeze(0)
```

D. Seq2Seq

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):
        # src : [src len, batch size]
        # trg : [trg len, batch size]
        # teacher_forcing_ratio is probability to use teacher forcing
        # e.g. if teacher_forcing_ratio is 0.75 we use teacher forcing 75% of the time

        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        # tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)
```

i. Source sentence가 먼저 encoder를 통과하여 모든 encoder hidden states H와 decoder의 initial hidden state s_0를 반환

```
        # encoder_outputs is all hidden states of the input sequence, back and forwards
        # hidden is the final forward and backward hidden states, passed through a linear layer
        encoder_outputs, hidden = self.encoder(src)
```

ii. Decoder의 첫번째 입력 값 y_0으로는 <sos> 토큰이 들어감

```
        # first input to the decoder is the <sos> tokens
        input = trg[0,:]
```

iii. Input token embedding y_{i-1}, decoder의 이전 hidden state s_{i-1}와 모든 encoder hidden states H가 decoder의 입력 값으로 들어가 다음에 올 예측 단어값 $\hat{y}_i$와 hidden state s_i를 반환

```
        for t in range(1, trg_len):

            # insert input token embedding, previous hidden state and all encoder hidden states
            # receive output tensor (predictions) and new hidden state
            output, hidden = self.decoder(input, hidden, encoder_outputs)

            # place predictions in a tensor holding predictions for each token
            outputs[t] = output
```

iv. 모든 target 단어에 대해 과정 (iii)을 반복 수행

E. Loss 분석

i. Cross entropy loss 사용

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

- ✓ N : batch size, C : 총 class 개수 = output dim
- ✓ Target field의 총 vocab 수를 class 개수로 하여 classification하는 것과 같음
  1. Prediction 값의 output dim 차원으로 softmax를 취하여 가장 높은 확률 값을 갖는 class를 최종 예측 단어로 선정
- ✓ Prediction shape : [batch size * (입력 문장 내 단어 수 - 1), target field의 총 vocab 수]
- ✓ Label shape : [batch size * (입력 문장 내 단어 수 - 1)]
- ✓ Target sentence의 첫번째 토큰인 <sos> 토큰은 제외하고 처리하여 (입력 문장 내 단어 수 - 1)만큼의 크기를 가짐

```
output = output[1:].view(-1, output_dim)
trg = trg[1:].view(-1)

# trg : [(trg len - 1) * batch size]
# output : [(trg len - 1) * batch size, output dim]

loss = criterion(output, trg)
```

3. 모델 학습 및 평가

A. 모델 학습

```
N_EPOCHS = 10
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()

    # Train
    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)

    # Validation
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
```

```
    if valid_loss < best_valid_loss:
        # Update best loss and save weights
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut3-model.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```

```
Epoch: 01 | Time: 1m 20s
        Train Loss: 5.020 | Train PPL: 151.383
         Val. Loss: 4.814 |  Val. PPL: 123.220
Epoch: 02 | Time: 1m 23s
        Train Loss: 4.128 | Train PPL:  62.026
         Val. Loss: 4.587 |  Val. PPL:  98.216
Epoch: 03 | Time: 1m 24s
        Train Loss: 3.473 | Train PPL:  32.242
         Val. Loss: 3.751 |  Val. PPL:  42.546
Epoch: 04 | Time: 1m 24s
        Train Loss: 2.916 | Train PPL:  18.465
         Val. Loss: 3.437 |  Val. PPL:  31.104
Epoch: 05 | Time: 1m 24s
        Train Loss: 2.521 | Train PPL:  12.438
         Val. Loss: 3.343 |  Val. PPL:  28.305
Epoch: 06 | Time: 1m 25s
        Train Loss: 2.234 | Train PPL:   9.342
         Val. Loss: 3.271 |  Val. PPL:  26.339
Epoch: 07 | Time: 1m 24s
        Train Loss: 1.991 | Train PPL:   7.321
         Val. Loss: 3.134 |  Val. PPL:  22.957
Epoch: 08 | Time: 1m 24s
        Train Loss: 1.773 | Train PPL:   5.889
         Val. Loss: 3.227 |  Val. PPL:  25.195
Epoch: 09 | Time: 1m 24s
        Train Loss: 1.601 | Train PPL:   4.960
         Val. Loss: 3.265 |  Val. PPL:  26.186
Epoch: 10 | Time: 1m 24s
        Train Loss: 1.488 | Train PPL:   4.427
         Val. Loss: 3.309 |  Val. PPL:  27.351
```

B. 모델 평가

```
# weights load
model.load_state_dict(torch.load('tut3-model.pt'))

# Evaluate model - using CrossEntropyLoss
test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
| Test Loss: 3.129 | Test PPL:  22.862 |
```