NLP_coding_assignment_1

전공 : 메타버스 테크놀로지

학번 / 이름 : V2022117 / 황민규

previous model (briefly):

1 - Sequence to Sequence Learning with Neural Networks

여기서는 아주 기본적인 seq2seq모델에 대해 소개한다.

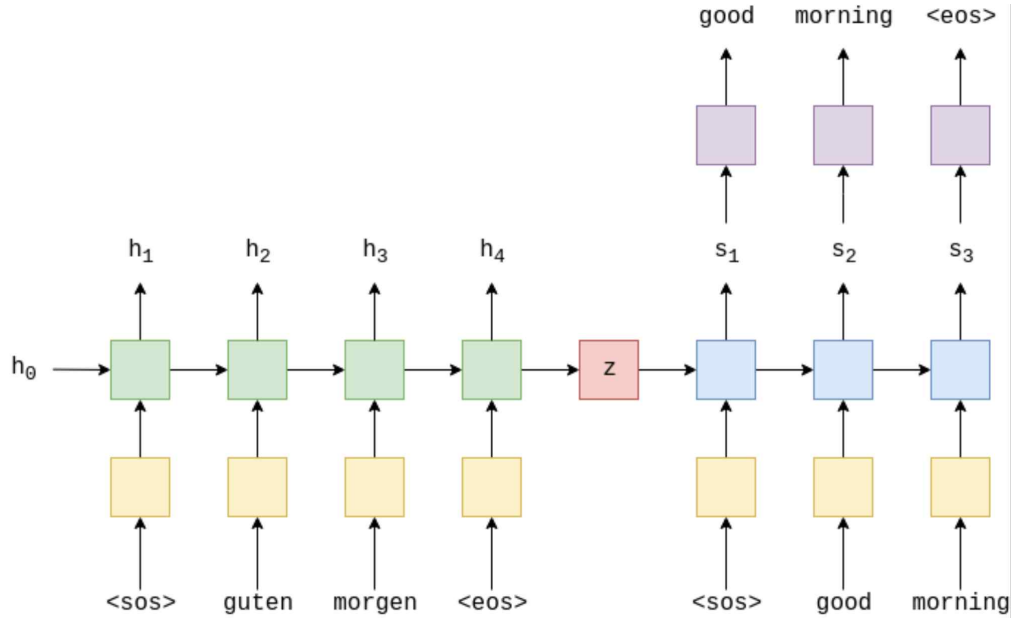


그림 [1] 기본적인 seq2seq의 모델도

그림 [1]에서 확인할 수 있듯, seq2seq모델은 기본적으로 encoder(green), decoder(blue)의 구조를 가지고 있으며, 이 둘은 RNN모델(LSTM, GRU)을 사용한다.

embedding layer :

embedding layer(yellow)에서는 각 Input(one-hot vectors)을 dense vectors로 변환을 해주는 기능을 하며, encoder, decoder에 서로 다른 embedding layer가 각각 하나씩 존재한다.

encoder :

encoder는 통상 RNN모델을 사용한다. encoder에서는 embedding layer를 통과한 Input을 대상으로 RNN모델을 통해 hidden states를 계산하고 마지막 hidden state인 context vector를 구하는데 사용한다.

context vector :

Z(red)는 encoder에서 RNN모델에서 얻어지는 마지막 hidden state며 통상적으로 context vector라 한다. context vector에는 encoder에 Input으로 주어지는 sentence의 모든 정보를 가지고 있으며, 차후 decoder의 initial hidden state로 입력이 된다.

decoder :

decoder에서는 진정한 기계번역이 시작된다. 먼저 Input(이전 time step의 Output)으로는 encoder와 동일하게 embedding layer를 통과한 dense vector를 받으며, initial hidden state로는 encoder에서 얻

은 context vector를 사용한다.

decoder_Input : 1. embedded input word(without <sos> token, output of previous time step) d(yt)

2. encoder context vector z or decoder hidden state st

통상적으로는 학습의 정확도와 속도를 높이기 위해 teching force라는 기법을 사용한다.

문제점 :

1. 입력 시퀀스의 길이에 상관없이, 항상 고정된 크기의 벡터(context vector)에 모든 정보를 압축하기 때문에 이 과정에서 정보 손실이 발생한다.

2. 입력 시퀀스의 길이가 길어지면 RNN에서 발생된 기울기 소실(vanising gradient) 문제가 존재한다.

2 - Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

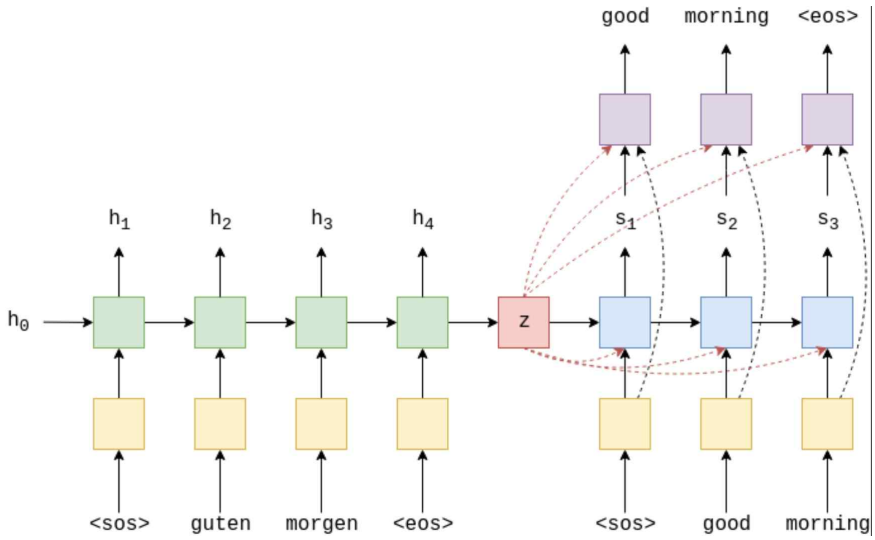


그림 [2]

여기서는 이전 모델에서 조금 더 진화된 모델을 제시한다.

이전 모델과 비슷한 구조를 가지고 있으며, 가장 큰 변화는 decoder부분이다.

이전에는 z(context vector)를 단순히 decoder의 initial hidden state로 입력했다면, 본 모델에서는 매 time step마다 hidden state와 Linear function(purple)에 입력하고 있으며, 추가적으로 embedded word를 Linear function(purple)에 입력하고 있다.

linear function_Input : 1. encoder context vector z

2. embedded input word d(yt)

3. decoder hidden state st

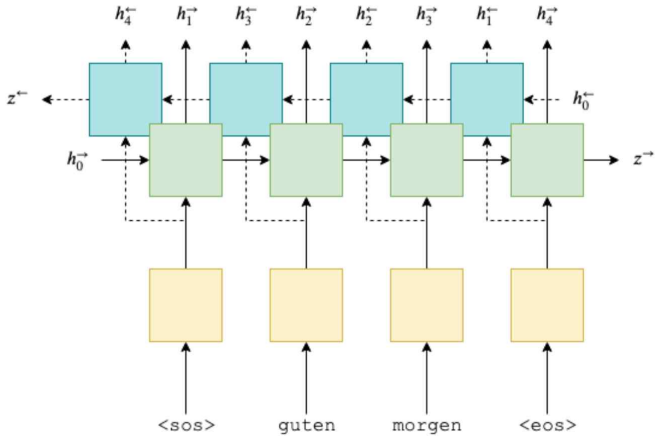
문제점 :

본 방식은 encoder 문장의 정보의 손실을 최소한으로 줄일 수 있다는 장점이 있지만, decoder에서 단어를 예측하는 매 time step마다 endcoder의 전체 문장에 대한 정보(context vector)를 똑같이 계속 본다는 문제점이 있다.

3 - Neural Machine Translation by Jointly Learning to Align and Translate

attention의 기본 아이디어는 decorder에서 output word를 예측하는 매 time step마다, encoder에서의 전체 입력 문장을 다시 한 번 참고하는 것이다. 다만, 이전 모델과 달리 모두 동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야 할 단어와 연관성 있는 입력 단어 부분을 좀 더 주목하게 한다.

Encoder :



본 모델에서는 single layer GRU, bidirectional RNN(쌍방향 RNN)을 사용한다.

bidirectional RNN :

딥러닝 모델은 예측모델입니다.

ex) 나는 _을 뒤집어 쓰고 펑펑 울었다.

라는 문장에서, 나는 _ 라는 시점에서 빈칸의 단어 x를 예측할 수 없다.

위의 예시처럼 시간적으로 앞의 데이터만 가지고는 올바른 예측을 하기 힘든 모델이 있다. 이러한 문제를 해결하기 위하여, 정방향뿐 아닌 역방향의 전파까지 실행하는 기법이다.

A forward RNN(green) : 문장의 왼쪽부터 오른쪽으로(정방향) embedding한다.

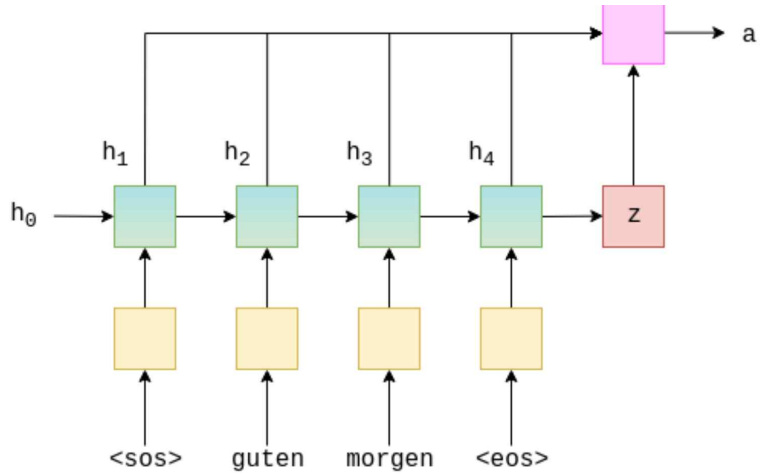
A backward RNN(teal) : 문장의 오른쪽부터 왼쪽으로(역방향) embedding한다.

$$h_t^{\rightarrow} = \text{EncoderGRU}^{\rightarrow}(e(x_t^{\rightarrow}), h_{t-1}^{\rightarrow})$$
$$h_t^{\leftarrow} = \text{EncoderGRU}^{\leftarrow}(e(x_t^{\leftarrow}), h_{t-1}^{\leftarrow})$$

양방향이기 때문에 2개의 context vector가 생긴다. 하지만 decoder는 bidirection이 아니기 때문에 single context vector z를 사용해야 한다.

이를 위해 encoder의 2개의 context vector를 concatenating을 진행하여 single vector로 만든다.

Attention :



attention layer에 encoder의 output인 모든 encoder hidden states H={h1,h2,....hT}를 Input으로 받으며, encoder의 context vector도 함께 Input으로 받는다.

해당 과정에서 첫번째로 energy를 구한다.

$$E_t = \tanh(\text{attn}(s_{t-1}, H))$$

H와 decoder hidden state st-1을 concat한다. 그 후에 linear function attn()을 지나고, 마지막으로 tanh()를 통과한다.

이를 통해 encoder의 hidden state가 decoder hidden state와 얼마나 잘 “match”되는지 알 수 있다.

Attention vector :

energy Et는 [dec hid dim, src len]의 size를 가지고있다.

이를 [src len]으로 변환하기 위해 아래의 수식을 사용한다.

$$\hat{a}_t = vE_t$$

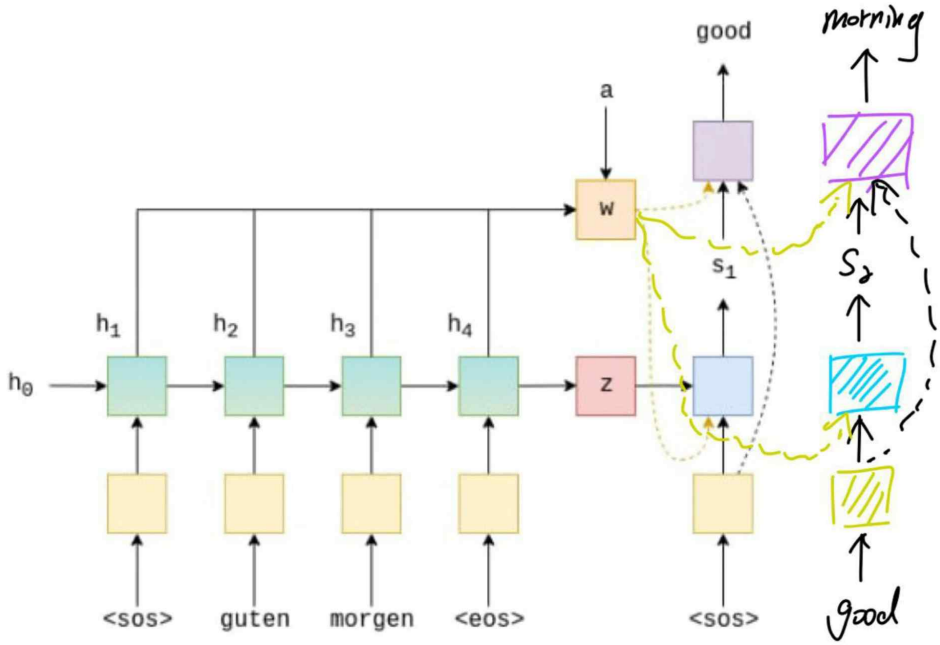
v = [1, dec hid dim]의 size를 가지고있으며, Et의 가중치로 볼 수 있다. 마지막에는 squeeze()함수를 사용해 차원 하나를 삭제함으로써 [src len]의 size를 얻는다.

attention vector는 위의 at_hat에 softmax를 사용하여 0~1까지의 값을 취하고, 합이 1이 된다.

source sentence의 길이만큼의 크기인 attention vector a 를 계산한다.

최종적으로 source sentence 길이 만큼의, 각각의 src token에 얼마만큼 attention 해야하는지 가중치가 담긴 attention vector가 산출된다.

Decoder :



source sentence의 hidden states H와 attention vector ai의 weighted sum을 통해 weighted source vector w를 계산한다.

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

위의 식에서 확인할 수 있듯이 weighted source vector w는 context vector ci와 같다. 이는 즉 w는 source sentence의 정보를 지닌 벡터라는 의미와 같다.

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

위의 식을 통해 현재 시점의 hidden state si를 구한다. ci = w이다.

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

위의 식을 통해 다음에 올 예측단어 yi_hat을 구할 수 있다. f()는 Linear function이다.

Seq2Seq :

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

        self.encoder = encoder
        self.decoder = decoder
        self.device = device # --> cuda

    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)
```

위의 코드는 Seq2Seq의 모델이다. device는 Cuda(Gpu)를 사용하여 진행했다.

```
        #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)

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

먼저 encoder에 sorce sentence를 입력으로 준다. 그리고 decoder의 first input으로 <sos>토큰이 들어간다.

```
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

    #decide if we are going to use teacher forcing or not
    teacher_force = random.random() < teacher_forcing_ratio
```

decoder의 이전 time step의 output yi-1, hidden state si-1와 모든 encoder hidden states H가 decoder의 입력 값으로 들어간다. 그리고 예측 단어 yi_hat과 현재 time step의 hidden state si를 출력한다.

본 모델의 teaching force의 ratio값은 0.5로 설정하고 학습을 진행한다.

Loss :

```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

본 모델에서는 Loss function으로 CrossEntropyLoss를 사용했다.

CrossEntropyLoss

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

criterion = nn.CrossEntropyLoss()

loss = criterion(predict, target) --> predict = output

Train model :

```
N_EPOCHS = 30
CLIP = 1
train_loss_list = []
valid_loss_list = []
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)
    train_loss_list.append(train_loss)
    valid_loss_list.append(valid_loss)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        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}')
```

여기서는 epoch을 30으로 설정하였다.

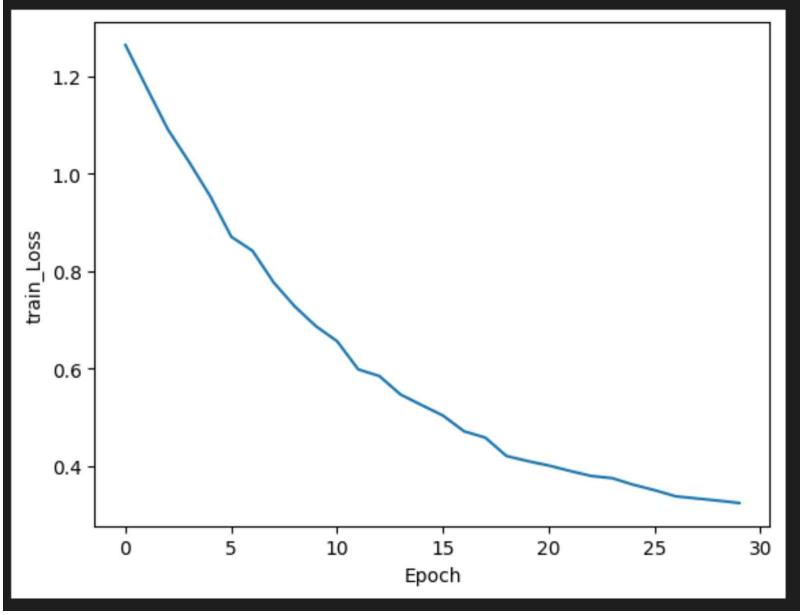
```
Output exceeds the size limit. Open the full output data in a text editor
Epoch: 01 | Time: 0m 43s
        Train Loss: 1.263 | Train PPL:   3.538
         Val. Loss: 3.480 |  Val. PPL:  32.461
Epoch: 02 | Time: 0m 43s
        Train Loss: 1.176 | Train PPL:   3.242
         Val. Loss: 3.534 |  Val. PPL:  34.270
Epoch: 03 | Time: 0m 44s
        Train Loss: 1.091 | Train PPL:   2.977
         Val. Loss: 3.513 |  Val. PPL:  33.539
Epoch: 04 | Time: 0m 44s
        Train Loss: 1.024 | Train PPL:   2.785
         Val. Loss: 3.608 |  Val. PPL:  36.901
Epoch: 05 | Time: 0m 44s
        Train Loss: 0.954 | Train PPL:   2.595
         Val. Loss: 3.641 |  Val. PPL:  38.113
Epoch: 06 | Time: 0m 44s
        Train Loss: 0.870 | Train PPL:   2.387
         Val. Loss: 3.803 |  Val. PPL:  44.834
Epoch: 07 | Time: 0m 44s
        Train Loss: 0.842 | Train PPL:   2.320
         Val. Loss: 3.804 |  Val. PPL:  44.889
Epoch: 08 | Time: 0m 45s
        Train Loss: 0.777 | Train PPL:   2.174
         Val. Loss: 3.924 |  Val. PPL:  50.592
Epoch: 09 | Time: 0m 45s
...
         Val. Loss: 5.066 |  Val. PPL: 158.609
Epoch: 30 | Time: 0m 42s
        Train Loss: 0.324 | Train PPL:   1.383
         Val. Loss: 5.122 |  Val. PPL: 167.691
```

train loss 그래프

```
plt.xlabel('Epoch')
plt.ylabel('train_Loss')
plt.plot(epoch_list, train_loss_list)
```

✓ 0.1s

```
[<matplotlib.lines.Line2D at 0x1ddef4a9190>]
```



valid loss 그래프

```
plt.xlabel('Epoch')
plt.ylabel('valid_loss')
plt.plot(epoch_list, valid_loss_list)
```

✓ 0.1s

```
[<matplotlib.lines.Line2D at 0x1ddef406700>]
```