CSE291 Projec1

Shanchuan You

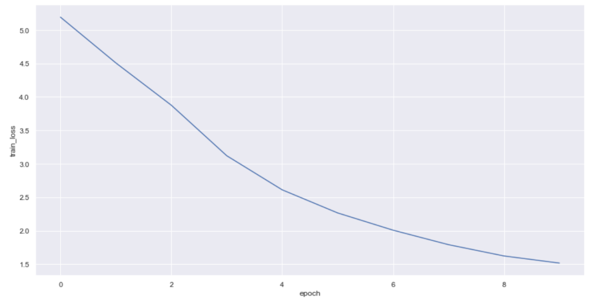
Abstract

This Report contains three parts. The first task is a basic seq2seq model which can transform German language into standard English.

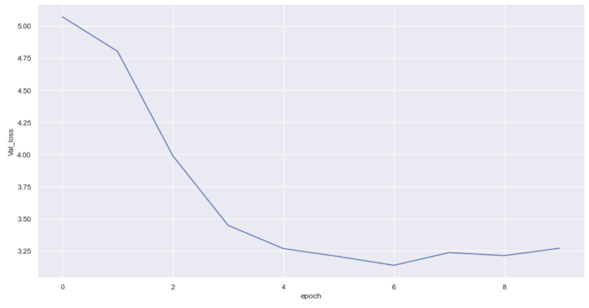
1 Task1

1.1 Training loss and the Validation loss

The first graph is the Training loss of the first 10 epoches.



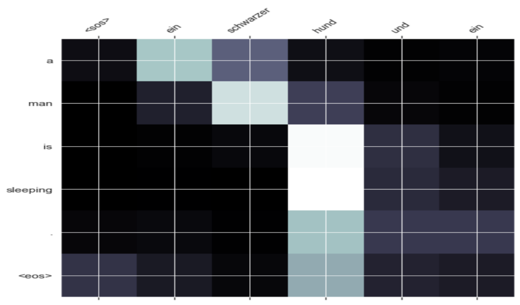
The below graph is the Validation loss of the first 10 epoches.



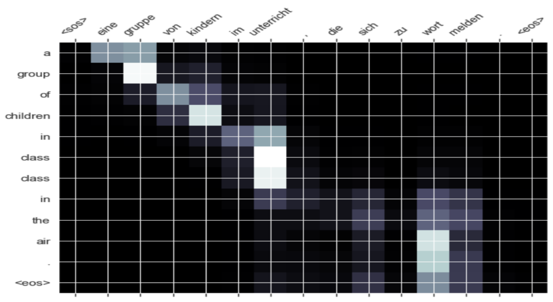
As we can see from the graph, as the epcoh increases, the training loss is getting lower, while the validation loss decreases at the beginning, but it tends to increase again.

1.2 Translation Samples and Attention Matrices

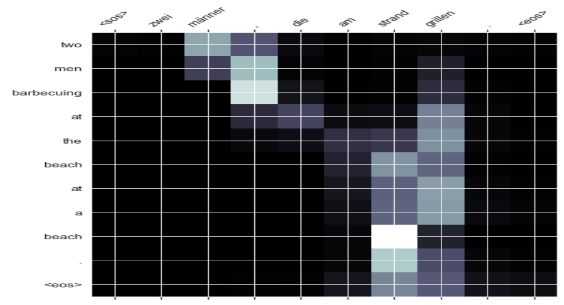
Src: ['Ein', 'Mann', 'schläft', '.']
Trg: ['A', 'man', 'sleeps']
Predicted: ['a', 'man', 'is', 'sleeping', '.', '$< eos >$']



Src: ["eine", "gruppe", "von", "kindern", "im", "unterricht", ",", "die", "sich", "zu", "wort", "melden", "."]
Trg: ["a", "group", "of", "kids", "in", "class", "with", "their", "hands", "in", "the", "air", "."]
Predicted: ['a', 'group', 'of', 'children', 'in', 'class', 'class', 'in', 'the', 'air', '.', '¡eos¿']

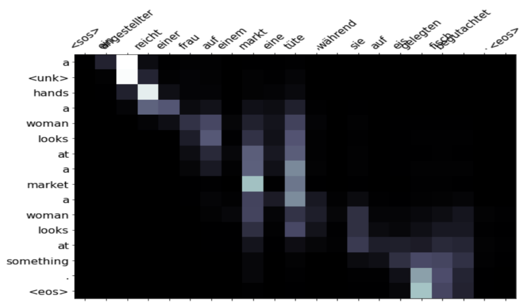


Src: ['zwei', 'männer', ',', 'die', 'am', 'strand', 'grillen', '.']
Trg: ['two', 'men', 'barbecuing', 'at', 'a', 'beach', '.']
Predicted: ['two', 'men', 'barbecuing', 'at', 'the', 'beach', 'at', 'a', 'beach', '.', '¡eos¿']



Src: ['ein', 'angestellter', 'reicht', 'einer', 'frau', 'auf', 'einem', 'markt', 'eine', 'tüte', ',', 'während', 'sie', 'auf', 'eis', 'gelegten', 'fisch', 'begutachtet', '.']
Trg: ['an', 'employee', 'is', 'handing', 'a', 'woman', 'a', 'bag', 'while', 'she', 'is', 'browsing', 'through', 'fish', 'on', 'ice', 'at', 'a', 'street', 'market', '.']

Predicted: ['a', '¡unk¿', 'hands', 'a', 'woman', 'looks', 'at', 'a', 'market', 'a', 'woman', 'looks', 'at', 'something', '.', '¡eos¿']



1.3 BLEU

BLEU : 28

2 Task2 Beam search

2.1 Simple Description of the Beam Search

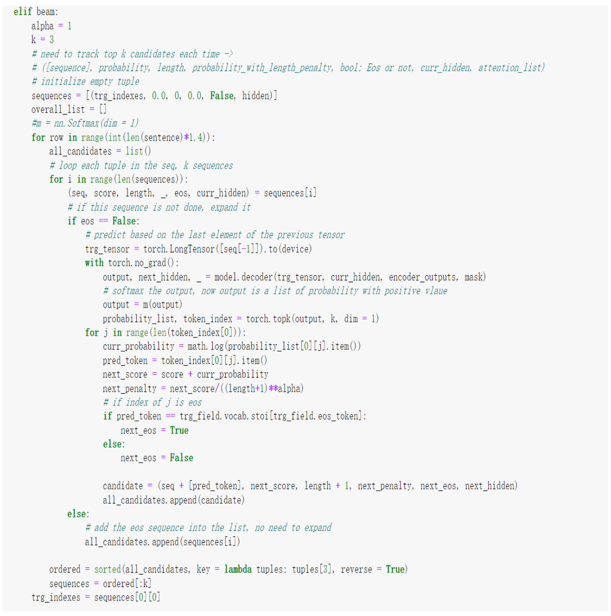
At the beginning, we initialize an empty list. Then for each step t, the algorithm loop through each current candidate, from candidate 1 to candidate k, where k is the beam width. For each candidate, the algorithm would predict the next token, which may have up to 5000 choices of this candidate, given its previous token. Then for each token, we choose the topk choices, as a result yielding k*k results. We perform sorting and choose the first k ranks as our candidates and make it to the next step t+1.

There are some things that are worth noting. The first is the number of step. Usually, the target sentence will be relatively the same length as the source sentence, though with minus +/- differences. So, the number of step will be entirely determined by the input sentence length, which I defined as 1.5*(length of input sentence), and down cast to integer.

The next is the length penalty. We know that the shorter the sentence, the way we calculate its prob the higher. For example a sentence with length 2 has the prob 0.3*0.4, that is because the first token has the prob 0.3, and the second token has prob 0.4. However, a longer sentence may have the prob 0.3*0.5*0.4, as the third token prob will be counted. In order to be fair with all sentences, the beam search algorithm introduces the length penalty log(prob) / (length**alpha), where alpha is a hyper parameter to monitor the penalty. The longer the length is, with log(prob) constant, the higher the result it is, since log(prob) is always negative.

2.2 Code

The code is shown below:



```
elif beam:
    alpha = 1
    k = 3
    # need to track top k candidates each time ->
    # ([sequence], probability, length, probability_with_length_penalty, bool: Eos or not, curr_hidden, attention_list)
    # initialize empty tuple
    sequences = [(trg_indexes, 0.0, 0, 0.0, False, hidden)]
    overall_list = []
    #m = nn.Softmax(dim = 1)
    for row in range(int(len(sentence)*1.4)):
        all_candidates = list()
        # loop each tuple in the seq, k sequences
        for i in range(len(sequences)):
            (seq, score, length, _, eos, curr_hidden) = sequences[i]
            # if this sequence is not done, expand it
            if eos == False:
                # predict based on the last element of the previous tensor
                trg_tensor = torch.LongTensor([seq[-1]]).to(device)
                with torch.no_grad():
                    output, next_hidden, _ = model.decoder(trg_tensor, curr_hidden, encoder_outputs, mask)
                    # softmax the output, now output is a list of probability with positive vlaue
                    output = m(output)
                    probability_list, token_index = torch.topk(output, k, dim = 1)
                for j in range(len(token_index[0])):
                    curr_probability = math.log(probability_list[0][j].item())
                    pred_token = token_index[0][j].item()
                    next_score = score + curr_probability
                    next_penalty = next_score/((length+1)**alpha)
                    # if index of j is eos
                    if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                        next_eos = True
                    else:
                        next_eos = False

                    candidate = (seq + [pred_token], next_score, length + 1, next_penalty, next_eos, next_hidden)
                    all_candidates.append(candidate)
            else:
                # add the eos sequence into the list, no need to expand
                all_candidates.append(sequences[i])

        ordered = sorted(all_candidates, key = lambda tuples: tuples[3], reverse = True)
        sequences = ordered[:k]
    trg_indexes = sequences[0][0]
```

2.3 Examples and Prediction

Src: ['Ein', 'Mann', 'schläft', '.']
Trg: ['A', 'man', 'sleeps']

Predicted: ['a', 'man', 'is', 'asleep', '.']

Src: ["eine", "gruppe", "von", "kindern", "im", "unterricht", ",", "die", "sich", "zu", "wort",

”melden”, ”.”]
Trg: [”a”, ”group”, ”of”, ”kids”, ”in”, ”class”, ”with”, ”their”, ”hands”, ”in”, ”the”, ”air”, ”.”]

Predicted[’a’, ’group’, ’of’, ’children’, ’in’, ’class’, ’in’, ’the’, ’air’, ’.’, ’¡eos¿’]

Src: [’zwei’, ’männer’, ’,’, ’die’, ’am’, ’strand’, ’grillen’, ’.’]
Trg: [’two’, ’men’, ’barbecuing’, ’at’, ’a’, ’beach’, ’.’]

Predicted: [’two’, ’men’, ’barbecuing’, ’at’, ’the’, ’beach’, ’.’, ’¡eos¿’]

Src: [’ein’, ’angestellter’, ’reicht’, ’einer’, ’frau’, ’auf’, ’einem’, ’markt’, ’eine’, ’tüte’, ’,’, ’während’, ’sie’, ’auf’, ’eis’, ’gelegten’, ’fisch’, ’begutachtet’, ’.’]
Trg: [’an’, ’employee’, ’is’, ’handing’, ’a’, ’woman’, ’a’, ’bag’, ’while’, ’she’, ’is’, ’browsing’, ’through’, ’fish’, ’on’, ’ice’, ’at’, ’a’, ’street’, ’market’, ’.’]

Predicted: [’a’, ’¡unk¿’, ’solider’, ’a’, ’woman’, ’looks’, ’at’, ’a’, ’market’, ’while’, ’looking’, ’at’, ’something’, ’.’, ’¡eos¿’]

2.4 Comparison

Basically, beam search and greedy search perform relatively the same at shorter sentence, while a little bit different at longer sentence. The reason might be that beam search has many choices each time, and that the greedy search only has one choice.

2.5 BLEU Score report and Comparison with Greedy

K: 3, alpha = 0.75 BLEU: 30.72 Time = 43.4s
k: 5, alpha = 0.75 BLEU: 31.36 Time = 1min 22s
k: 8, alpha = 0.75 BLEU: 31.43 Time = 2min 36s
k: 10 alpha = 0.75 BLEU: 31.43 Time = 3min 38s

As we can see, when the k increases, the BLEU score also slightly increases, but it seems to converges.

In general, Beam Search is definitely better than the Greedy Search, but it consumes many more computation resources, which is a trade off. Thus, we need to manually modify the hyper parameters in order to find a balance between computation complexity and the BLEU score. Despite of this, from the examples, we can find that Beam Search does perform better than the previous greedy.

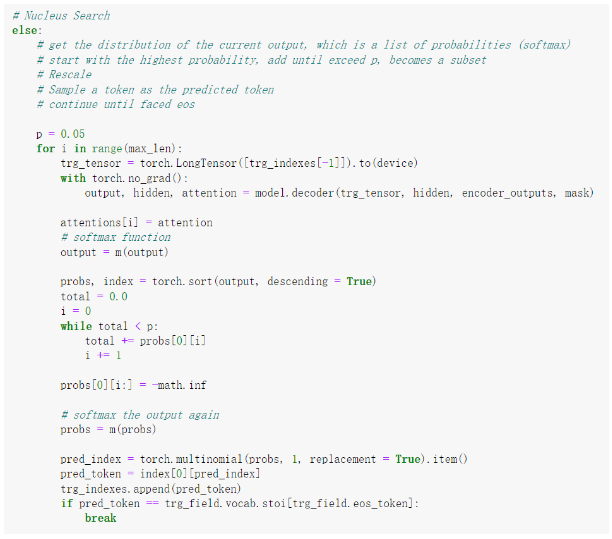
3 Task3 Nucleus Sampling

3.1 Simple Description of the Nucleus Sampling

The logic behind Nucleus sampling is very simple. Unlike greedy search, Nucleus, at each step t, first set up a threshold, says probability p. Then it starts adding each word’s probability from highest to lowest until the cumulative probability exceeds the threshold. Then, we set the other unselected token’s probability as 0. In other words, we always choose the top k token as a subset. Then we softmax this subset, and sample a token from the subset, based on their new assigned prob. Repeat the previous step ubtil Eos predicted or the total number of steps reach the maxlen.

3.2 Code

The code is shown below:



```
# Nucleus Search
else:
    # get the distribution of the current output, which is a list of probabilities (softmax)
    # start with the highest probability, add until exceed p, becomes a subset
    # Rescale
    # Sample a token as the predicted token
    # continue until faced eos

    p = 0.05
    for i in range(max_len):
        trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)
        with torch.no_grad():
            output, hidden, attention = model.decoder(trg_tensor, hidden, encoder_outputs, mask)

        attentions[i] = attention
        # softmax function
        output = m(output)

        probs, index = torch.sort(output, descending = True)
        total = 0.0
        i = 0
        while total < p:
            total += probs[0][i]
            i += 1

        probs[0][i:] = -math.inf

        # softmax the output again
        probs = m(probs)

        pred_index = torch.multinomial(probs, 1, replacement = True).item()
        pred_token = index[0][pred_index]
        trg_indexes.append(pred_token)
        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break
```

3.3 Examples and Prediction

Src: [’Ein’, ’Mann’, ’schläft’, ’.’]
Trg: [’A’, ’man’, ’sleeps’]

Predicted: [’a’, ’man’, ’is’, ’asleep’, ’.’, ’¡eos¿’]

Src: [”eine”, ”gruppe”, ”von”, ”kindern”, ”im”, ”unterricht”, ”,”, ”die”, ”sich”, ”zu”, ”wort”, ”melden”, ”.”]

Trg: [”a”, ”group”, ”of”, ”kids”, ”in”, ”class”, ”with”, ”their”, ”hands”, ”in”, ”the”, ”air”, ”.”]

Predicted[’a’, ’group’, ’of’, ’children’, ’in’, ’class’, ’in’, ’the’, ’air’, ’.’, ’¡eos¿’]

Src: [’zwei’, ’männer’, ’,’, ’die’, ’am’, ’strand’, ’grillen’, ’.’]
Trg: [’two’, ’men’, ’barbecuing’, ’at’, ’a’, ’beach’, ’.’]

Predicted: [’¡sos¿’, ’two’, ’men’, ’barbecuing’, ’at’, ’at’, ’the’, ’beach’, ’.’, ’¡eos¿’]

Src: [’ein’, ’angestellter’, ’reicht’, ’einer’, ’frau’, ’auf’, ’einem’, ’markt’, ’eine’, ’tüte’, ’,’, ’während’, ’sie’, ’auf’, ’eis’, ’gelegten’, ’fisch’, ’begutachtet’, ’.’]
Trg: [’an’, ’employee’, ’is’, ’handing’, ’a’, ’woman’, ’a’, ’bag’, ’while’, ’she’, ’is’, ’browsing’, ’through’, ’fish’, ’on’, ’ice’, ’at’, ’a’, ’street’, ’market’, ’.’]

Predicted: [’a’, ’¡unk¿’, ’hands’, ’a’, ’woman’, ’looks’, ’at’, ’a’, ’market’, ’a’, ’woman’, ’looks’, ’at’, ’something’, ’.’, ’¡eos¿’]

3.4 Comparison

Compared with the previous Beam search and the greedy search, they are basically the same. However, There are a slightly different when the input sentence is pretty long. Greedy Search and Nucleus Sampling are predict relatively the same on 4th sentence, while beam search is different. This is probably beam search has many choices at each time step t, and modifying the length penalty plays an really important role during the step.

3.5 BLEU

Threshold $->$ p
p = 0.3 BLEU = 28.49 Time = 20.3
p = 0.2 BLEU = 29.15 Time = 19.9
p = 0.1 BLEU = 29.47 Time = 19.2
p = 0.05BLEU = 29.62 Time = 19.5

So, when the p decreases, the BLEU increases. This is probably because a larger p means a larger subset, where many more elements might be put into the pool to be considered. As a result, it decreases the accuracy. Unlike Greedy search, which always pick the the highest probability token, Nucleus sampling makes the overall choosing process more fun.

```
elif beam:
    alpha = 1
    k = 3
    # need to track top k candidates each time ->
    # ([sequence], probability, length, probability_with_length_penalty, bool: Eos or not, curr_hidden, attention_list)
    # initialize empty tuple
    sequences = [(trg_indexes, 0.0, 0, 0.0, False, hidden)]
    overall_list = []
    #m = nn.Softmax(dim = 1)
    for row in range(int(len(sentence)*1.4)):
        all_candidates = list()
        # loop each tuple in the seq, k sequences
        for i in range(len(sequences)):
            (seq, score, length, _, eos, curr_hidden) = sequences[i]
            # if this sequence is not done, expand it
            if eos == False:
                # predict based on the last element of the previous tensor
                trg_tensor = torch.LongTensor([seq[-1]]).to(device)
                with torch.no_grad():
                    output, next_hidden, _ = model.decoder(trg_tensor, curr_hidden, encoder_outputs, mask)
                    # softmax the output, now output is a list of probability with positive vlaue
                    output = m(output)
                    probability_list, token_index = torch.topk(output, k, dim = 1)
                for j in range(len(token_index[0])):
                    curr_probability = math.log(probability_list[0][j].item())
                    pred_token = token_index[0][j].item()
                    next_score = score + curr_probability
                    next_penalty = next_score/((length+1)**alpha)
                    # if index of j is eos
                    if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                        next_eos = True
                    else:
                        next_eos = False

                    candidate = (seq + [pred_token], next_score, length + 1, next_penalty, next_eos, next_hidden)
                    all_candidates.append(candidate)
            else:
                # add the eos sequence into the list, no need to expand
                all_candidates.append(sequences[i])

        ordered = sorted(all_candidates, key = lambda tuples: tuples[3], reverse = True)
        sequences = ordered[:k]
    trg_indexes = sequences[0][0]
```

```
# Nucleus Search
else:
    # get the distribution of the current output, which is a list of probabilities (softmax)
    # start with the highest probability, add until exceed p, becomes a subset
    # Rescale
    # Sample a token as the predicted token
    # continue until faced eos

    p = 0.05
    for i in range(max_len):
        trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)
        with torch.no_grad():
            output, hidden, attention = model.decoder(trg_tensor, hidden, encoder_outputs, mask)

        attentions[i] = attention
        # softmax function
        output = m(output)

        probs, index = torch.sort(output, descending = True)
        total = 0.0
        i = 0
        while total < p:
            total += probs[0][i]
            i += 1

        probs[0][i:] = -math.inf

        # softmax the output again
        probs = m(probs)

        pred_index = torch.multinomial(probs, 1, replacement = True).item()
        pred_token = index[0][pred_index]
        trg_indexes.append(pred_token)
        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break
```