Lab10 Report

st122314

I. Greedy Search

1. Load and Process the Data

We are dealing with sequences of words, which cannot be directly mapped to a continuous vector space as we need for LSTMs. We will therefore create a mapping from each unique word in the dataset to an index value.

- Voc class : creates both the forward mapping from words to indices and the reverse mapping from indices back to words, as well as a count of each word and a total word count.
  - (addWord): a method to add a word to the vocabulary
  - (addSentence): a method to add all words in a sentence at once
  - (trim) : a method to trim infrequently seen words

In vocab preprocessing, the following steps were employed

- Converting Unicode strings to ASCII using unicodeToAscii
- normalizeString : convert every letter to lowercase and trim all non-letter characters except for basic punctuation
- filterPairs: filtering out sentences with length greater than the MAX_LENGTH threshold to improve training convergence

After processing chat data file and made into pairs of question-answer sentences, the below two-step process were performed.

- Trim words appearing fewer than MIN_COUNT times with the previously-given Voc.trim method.
- Filter out all sentence pairs containing trimmed words.

Next,dataset were splitted into testing and training pair sets and convereted to tensors by using:

- inputVar: handles the process of converting sentences to tensor by creating a correctly shaped zero-padded tensor
  - returns a tensor of lengths for each of the sequences in the batch which will be passed to our decoder later
- outputVar: performs a similar function to inputVar, but instead of returning a lengths tensor
  - returns a binary mask tensor and a maximum target sentence length
- batch2TrainData: takes a bunch of pairs and returns the input and target tensors using the aforementioned functions

2. Define Model

- Encoder : consist of bidirectional LSTM units which consider the embeddings or features of next word suitable to predict target variable
- Decoder : consist “attention mechanism” that allows the decoder to pay attention to certain parts of the input sequence, rather than using the entire fixed context at every step

3. Training

- maskNLLLoss: calculates our loss based on our decoder’s output tensor, the target tensor, and a binary mask tensor describing the padding of the target tensor
  - calculates the average negative log likelihood of the elements that correspond to a 1 in the mask tensor.

In single training iteration, teacher forcing and gradient clipping were used to aid in conergence.

- teacher forcing : set by teacher_forcing_ratio, the current target word was used as the decoder’s next input rather than using the decoder’s current guess
- gradient clipping : use for countering the “exploding gradient” problem

In the full training procedure, trainIters : run n_iterations of training given the passed models, optimizers, data, etc.

GreedySearchDecoder :

- is greedy decoding operation which use during training when we are NOT using teacher forcing
- choose the word from decoder_output with the highest softmax value for each time step
- is optimal on a single time-step level.

After training the model and evaluating, the following is the running result of the Greedy Search.

The below cell is the result of data load and processing in Greedy Search Decoding

```
st122314@c62b2f4dc41a:~/Lab10$ /bin/python3 /home/st122314/Lab10/greedy-chatbot.py
Start preparing training data ...
Reading lines...
Read 221282 sentence pairs
Trimmed to 64271 sentence pairs
Counting words...
Counted words: 18008

pairs:
['there .', 'where ?']
['you have my word . as a gentleman', 'you re sweet .']
['hi .', 'looks like things worked out tonight huh ?']
['you know chastity ?', 'i believe we share an art instructor']
['have fun tonight ?', 'tons']
['well no . . .', 'then that s all you had to say .']
['then that s all you had to say .', 'but']
['but', 'you always been this selfish ?']
['do you listen to this crap ?', 'what crap ?']
['what good stuff ?', 'the real you .']
```

```
keep_words 7823 / 18005 = 0.4345
Trimmed from 64271 pairs to 53165, 0.8272 of total
[['there .', 'where ?'], ['you have my word . as a gentleman', 'you re sweet .'],
['hi .', 'looks like things worked out tonight huh ?'], ['have fun tonight ?',
'tons'], ['well no . . .', 'then that s all you had to say .']]
[['you have my word . as a gentleman', 'you re sweet .'], ['well no . . .', 'then
that s all you had to say .'], ['have fun tonight ?', 'tons'], ['there .', 'where
?'], ['hi .', 'looks like things worked out tonight huh ?']]
tensor([[  54,   25,   25,  124,  290],
        [1655,  200, 1140,    9,  380],
        [   4,   53,   40,  125,    4],
        [   2,  360,  125,  242,   76],
        [   0,  180,   25,  188,   37],
        [   0,    4,  132,  170,  945],
        [   0,    2,  920,   66,    4],
        [   0,    0, 3117,    2,    2],
        [   0,    0,    4,    0,    0],
        [   0,    0,    2,    0,    0]])
tensor([[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [False,  True,  True,  True,  True],
        [False,  True,  True,  True,  True],
        [False,  True,  True,  True,  True],
        [False, False,  True,  True,  True],
        [False, False,  True, False, False],
        [False, False,  True, False, False]])
```

We can see the training with 12000 iterations in Greedy Search Decoding and loss values were deceased.

```
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 0.8%; Average loss: 5.3060
Iteration: 200; Percent complete: 1.7%; Average loss: 4.6995
Iteration: 300; Percent complete: 2.5%; Average loss: 4.4700
Iteration: 400; Percent complete: 3.3%; Average loss: 4.3123
Iteration: 500; Percent complete: 4.2%; Average loss: 4.1686
Iteration: 600; Percent complete: 5.0%; Average loss: 4.0704
Iteration: 700; Percent complete: 5.8%; Average loss: 3.9952
Iteration: 800; Percent complete: 6.7%; Average loss: 3.9128
Iteration: 900; Percent complete: 7.5%; Average loss: 3.8448
Iteration: 1000; Percent complete: 8.3%; Average loss: 3.7770
Iteration: 1100; Percent complete: 9.2%; Average loss: 3.7236
Iteration: 1200; Percent complete: 10.0%; Average loss: 3.6551
Iteration: 1300; Percent complete: 10.8%; Average loss: 3.6092
Iteration: 1400; Percent complete: 11.7%; Average loss: 3.5425
Iteration: 1500; Percent complete: 12.5%; Average loss: 3.5004
Iteration: 1600; Percent complete: 13.3%; Average loss: 3.4536
Iteration: 1700; Percent complete: 14.2%; Average loss: 3.4007
Iteration: 1800; Percent complete: 15.0%; Average loss: 3.3559
Iteration: 1900; Percent complete: 15.8%; Average loss: 3.3233
Iteration: 2000; Percent complete: 16.7%; Average loss: 3.2634
content/cb_model/Chat/2-4_512
Iteration: 2100; Percent complete: 17.5%; Average loss: 3.2232
Iteration: 2200; Percent complete: 18.3%; Average loss: 3.1959
```

```
Iteration: 2300; Percent complete: 19.2%; Average loss: 3.1307
Iteration: 2400; Percent complete: 20.0%; Average loss: 3.0985
Iteration: 2500; Percent complete: 20.8%; Average loss: 3.0538
Iteration: 2600; Percent complete: 21.7%; Average loss: 3.0083
Iteration: 2700; Percent complete: 22.5%; Average loss: 2.9551
Iteration: 2800; Percent complete: 23.3%; Average loss: 2.9424
Iteration: 2900; Percent complete: 24.2%; Average loss: 2.8887
Iteration: 3000; Percent complete: 25.0%; Average loss: 2.8404
Iteration: 3100; Percent complete: 25.8%; Average loss: 2.8148
Iteration: 3200; Percent complete: 26.7%; Average loss: 2.7623
Iteration: 3300; Percent complete: 27.5%; Average loss: 2.7137
Iteration: 3400; Percent complete: 28.3%; Average loss: 2.6826
Iteration: 3500; Percent complete: 29.2%; Average loss: 2.6348
Iteration: 3600; Percent complete: 30.0%; Average loss: 2.6094
Iteration: 3700; Percent complete: 30.8%; Average loss: 2.5551
Iteration: 3800; Percent complete: 31.7%; Average loss: 2.5328
Iteration: 3900; Percent complete: 32.5%; Average loss: 2.4776
Iteration: 4000; Percent complete: 33.3%; Average loss: 2.4431
content/cb_model/Chat/2-4_512
Iteration: 4100; Percent complete: 34.2%; Average loss: 2.4073
Iteration: 4200; Percent complete: 35.0%; Average loss: 2.3723
Iteration: 4300; Percent complete: 35.8%; Average loss: 2.3369
Iteration: 4400; Percent complete: 36.7%; Average loss: 2.2857
Iteration: 4500; Percent complete: 37.5%; Average loss: 2.2638
Iteration: 4600; Percent complete: 38.3%; Average loss: 2.2127
Iteration: 4700; Percent complete: 39.2%; Average loss: 2.1878
Iteration: 4800; Percent complete: 40.0%; Average loss: 2.1517
Iteration: 4900; Percent complete: 40.8%; Average loss: 2.1029
Iteration: 5000; Percent complete: 41.7%; Average loss: 2.0706
Iteration: 5100; Percent complete: 42.5%; Average loss: 2.0426
Iteration: 5200; Percent complete: 43.3%; Average loss: 1.9931
Iteration: 5300; Percent complete: 44.2%; Average loss: 1.9791
Iteration: 5400; Percent complete: 45.0%; Average loss: 1.9256
Iteration: 5500; Percent complete: 45.8%; Average loss: 1.9048
Iteration: 5600; Percent complete: 46.7%; Average loss: 1.8588
Iteration: 5700; Percent complete: 47.5%; Average loss: 1.8291
Iteration: 5800; Percent complete: 48.3%; Average loss: 1.7999
Iteration: 5900; Percent complete: 49.2%; Average loss: 1.7652
Iteration: 6000; Percent complete: 50.0%; Average loss: 1.7376
content/cb_model/Chat/2-4_512
Iteration: 6100; Percent complete: 50.8%; Average loss: 1.7044
Iteration: 6200; Percent complete: 51.7%; Average loss: 1.6738
Iteration: 6300; Percent complete: 52.5%; Average loss: 1.6465
Iteration: 6400; Percent complete: 53.3%; Average loss: 1.6107
Iteration: 6500; Percent complete: 54.2%; Average loss: 1.5808
Iteration: 6600; Percent complete: 55.0%; Average loss: 1.5432
Iteration: 6700; Percent complete: 55.8%; Average loss: 1.5168
Iteration: 6800; Percent complete: 56.7%; Average loss: 1.4900
Iteration: 6900; Percent complete: 57.5%; Average loss: 1.4641
Iteration: 7000; Percent complete: 58.3%; Average loss: 1.4292
Iteration: 7100; Percent complete: 59.2%; Average loss: 1.4075
Iteration: 7200; Percent complete: 60.0%; Average loss: 1.3770
Iteration: 7300; Percent complete: 60.8%; Average loss: 1.3565
Iteration: 7400; Percent complete: 61.7%; Average loss: 1.3333
Iteration: 7500; Percent complete: 62.5%; Average loss: 1.3031
Iteration: 7600; Percent complete: 63.3%; Average loss: 1.2788
Iteration: 7700; Percent complete: 64.2%; Average loss: 1.2551
Iteration: 7800; Percent complete: 65.0%; Average loss: 1.2299
Iteration: 7900; Percent complete: 65.8%; Average loss: 1.2075
Iteration: 8000; Percent complete: 66.7%; Average loss: 1.1816
content/cb_model/Chat/2-4_512
```

```
Iteration: 8100; Percent complete: 67.5%; Average loss: 1.1534
Iteration: 8200; Percent complete: 68.3%; Average loss: 1.1403
Iteration: 8300; Percent complete: 69.2%; Average loss: 1.1189
Iteration: 8400; Percent complete: 70.0%; Average loss: 1.0921
Iteration: 8500; Percent complete: 70.8%; Average loss: 1.0769
Iteration: 8600; Percent complete: 71.7%; Average loss: 1.0542
Iteration: 8700; Percent complete: 72.5%; Average loss: 1.0361
Iteration: 8800; Percent complete: 73.3%; Average loss: 1.0177
Iteration: 8900; Percent complete: 74.2%; Average loss: 0.9974
Iteration: 9000; Percent complete: 75.0%; Average loss: 0.9761
Iteration: 9100; Percent complete: 75.8%; Average loss: 0.9492
Iteration: 9200; Percent complete: 76.7%; Average loss: 0.9416
Iteration: 9300; Percent complete: 77.5%; Average loss: 0.9166
Iteration: 9400; Percent complete: 78.3%; Average loss: 0.8996
Iteration: 9500; Percent complete: 79.2%; Average loss: 0.8808
Iteration: 9600; Percent complete: 80.0%; Average loss: 0.8655
Iteration: 9700; Percent complete: 80.8%; Average loss: 0.8558
Iteration: 9800; Percent complete: 81.7%; Average loss: 0.8343
Iteration: 9900; Percent complete: 82.5%; Average loss: 0.8248
Iteration: 10000; Percent complete: 83.3%; Average loss: 0.8070
content/cb_model/Chat/2-4_512
Iteration: 10100; Percent complete: 84.2%; Average loss: 0.7996
Iteration: 10200; Percent complete: 85.0%; Average loss: 0.7849
Iteration: 10300; Percent complete: 85.8%; Average loss: 0.7646
Iteration: 10400; Percent complete: 86.7%; Average loss: 0.7534
Iteration: 10500; Percent complete: 87.5%; Average loss: 0.7359
Iteration: 10600; Percent complete: 88.3%; Average loss: 0.7193
Iteration: 10700; Percent complete: 89.2%; Average loss: 0.6993
Iteration: 10800; Percent complete: 90.0%; Average loss: 0.6930
Iteration: 10900; Percent complete: 90.8%; Average loss: 0.6822
Iteration: 11000; Percent complete: 91.7%; Average loss: 0.6747
Iteration: 11100; Percent complete: 92.5%; Average loss: 0.6678
Iteration: 11200; Percent complete: 93.3%; Average loss: 0.6526
Iteration: 11300; Percent complete: 94.2%; Average loss: 0.6381
Iteration: 11400; Percent complete: 95.0%; Average loss: 0.6272
Iteration: 11500; Percent complete: 95.8%; Average loss: 0.6203
Iteration: 11600; Percent complete: 96.7%; Average loss: 0.6108
Iteration: 11700; Percent complete: 97.5%; Average loss: 0.5931
Iteration: 11800; Percent complete: 98.3%; Average loss: 0.5870
Iteration: 11900; Percent complete: 99.2%; Average loss: 0.5797
Iteration: 12000; Percent complete: 100.0%; Average loss: 0.5726
content/cb_model/Chat/2-4_512
0 0.1410802874817691 0.04887164517296948
1000 0.14636273855370768 0.05369469026668399
2000 0.14459147083794235 0.05515170412998706
3000 0.14469663503300492 0.054557500573799676
4000 0.1481420628454823 0.055882947532570594
5000 0.1464385377903077 0.05527206952174228
6000 0.144915610840369 0.054479847706515996
7000 0.15316237002079322 0.06355854896913507
8000 0.15251515144401484 0.06301620760084482
```

In this Greedy Search Decoding, we calculated BLEU score for 1 gram and 2 gram which are 0.15179748338026577 and 0.062497703310824354 respectively.

```
Total Bleu Score for 1 grams on testing pairs:  0.15179748338026577
Total Bleu Score for 2 grams on testing pairs:  0.062497703310824354
```

Chat with the greedy search bot

```
> Hello?
Bot: hello . s welles . .
> Nice to meet you in this terminal.
Bot: you ll go now . .
> How are you?
Bot: all right son . let s go !
> okay.. Thank you
Bot: i ll get the dress . out .
>
```

II. Beam Search

- Beam search is an improved version of greedy search
- It has a hyperparameter named beam size, k
- At time step 1, selecting k tokens with the highest conditional probabilities. Each of them will be the first token of k candidate output sequences, respectively
- At each subsequent time step, based on the k candidate output sequences at the previous time step, k candidate output sequences has been selected with the highest conditional probabilities from k|r| possible choices.

Beam search and Greedy search have the same data loading, processing, modelling, training and evaluation. One differnce is GreedySearchDecoder and Beam decoding class.

The below cell is data load and processing for Beam Search Decoding.

```
st122314@9a1d87bf22c3:~/Lab10$ /bin/python3 /home/st122314/Lab10/beam_search.py
Start preparing training data ...
Reading lines...
Read 221282 sentence pairs
Trimmed to 64271 sentence pairs
Counting words...
Counted words: 18008

pairs:
['there .', 'where ?']
['you have my word . as a gentleman', 'you re sweet .']
['hi .', 'looks like things worked out tonight huh ?']
['you know chastity ?', 'i believe we share an art instructor']
['have fun tonight ?', 'tons']
['well no . . .', 'then that s all you had to say .']
['then that s all you had to say .', 'but']
['but', 'you always been this selfish ?']
['do you listen to this crap ?', 'what crap ?']
['what good stuff ?', 'the real you .']
keep_words 7823 / 18005 = 0.4345
Trimmed from 64271 pairs to 53165, 0.8272 of total
```

We can see the training with 12000 iterations in Beam Search Decoding and loss values were deceased.

```
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 0.8%; Average loss: 5.3070
Iteration: 200; Percent complete: 1.7%; Average loss: 4.6527
Iteration: 300; Percent complete: 2.5%; Average loss: 4.3967
Iteration: 400; Percent complete: 3.3%; Average loss: 4.2879
Iteration: 500; Percent complete: 4.2%; Average loss: 4.1806
Iteration: 600; Percent complete: 5.0%; Average loss: 4.0683
Iteration: 700; Percent complete: 5.8%; Average loss: 4.0168
Iteration: 800; Percent complete: 6.7%; Average loss: 3.9498
Iteration: 900; Percent complete: 7.5%; Average loss: 3.8872
Iteration: 1000; Percent complete: 8.3%; Average loss: 3.8207
Iteration: 1100; Percent complete: 9.2%; Average loss: 3.7593
Iteration: 1200; Percent complete: 10.0%; Average loss: 3.7044
Iteration: 1300; Percent complete: 10.8%; Average loss: 3.6422
Iteration: 1400; Percent complete: 11.7%; Average loss: 3.5868
Iteration: 1500; Percent complete: 12.5%; Average loss: 3.5488
Iteration: 1600; Percent complete: 13.3%; Average loss: 3.4775
Iteration: 1700; Percent complete: 14.2%; Average loss: 3.4311
Iteration: 1800; Percent complete: 15.0%; Average loss: 3.3923
Iteration: 1900; Percent complete: 15.8%; Average loss: 3.3407
Iteration: 2000; Percent complete: 16.7%; Average loss: 3.3037
content/cb_model/Chat/2-4_512
Iteration: 2100; Percent complete: 17.5%; Average loss: 3.2544
Iteration: 2200; Percent complete: 18.3%; Average loss: 3.2259
Iteration: 2300; Percent complete: 19.2%; Average loss: 3.1731
Iteration: 2400; Percent complete: 20.0%; Average loss: 3.1425
Iteration: 2500; Percent complete: 20.8%; Average loss: 3.0807
Iteration: 2600; Percent complete: 21.7%; Average loss: 3.0427
Iteration: 2700; Percent complete: 22.5%; Average loss: 3.0337
Iteration: 2800; Percent complete: 23.3%; Average loss: 2.9592
Iteration: 2900; Percent complete: 24.2%; Average loss: 2.9324
Iteration: 3000; Percent complete: 25.0%; Average loss: 2.8996
Iteration: 3100; Percent complete: 25.8%; Average loss: 2.8528
Iteration: 3200; Percent complete: 26.7%; Average loss: 2.8014
Iteration: 3300; Percent complete: 27.5%; Average loss: 2.7761
Iteration: 3400; Percent complete: 28.3%; Average loss: 2.7320
Iteration: 3500; Percent complete: 29.2%; Average loss: 2.7024
Iteration: 3600; Percent complete: 30.0%; Average loss: 2.6705
Iteration: 3700; Percent complete: 30.8%; Average loss: 2.6362
Iteration: 3800; Percent complete: 31.7%; Average loss: 2.5955
Iteration: 3900; Percent complete: 32.5%; Average loss: 2.5634
Iteration: 4000; Percent complete: 33.3%; Average loss: 2.5134
content/cb_model/Chat/2-4_512
Iteration: 4100; Percent complete: 34.2%; Average loss: 2.4784
Iteration: 4200; Percent complete: 35.0%; Average loss: 2.4509
Iteration: 4300; Percent complete: 35.8%; Average loss: 2.4266
Iteration: 4400; Percent complete: 36.7%; Average loss: 2.3841
Iteration: 4500; Percent complete: 37.5%; Average loss: 2.3469
Iteration: 4600; Percent complete: 38.3%; Average loss: 2.3299
Iteration: 4700; Percent complete: 39.2%; Average loss: 2.2926
Iteration: 4800; Percent complete: 40.0%; Average loss: 2.2482
Iteration: 4900; Percent complete: 40.8%; Average loss: 2.2217
Iteration: 5000; Percent complete: 41.7%; Average loss: 2.1911
Iteration: 5100; Percent complete: 42.5%; Average loss: 2.1572
Iteration: 5200; Percent complete: 43.3%; Average loss: 2.1202
Iteration: 5300; Percent complete: 44.2%; Average loss: 2.0856
Iteration: 5400; Percent complete: 45.0%; Average loss: 2.0642
Iteration: 5500; Percent complete: 45.8%; Average loss: 2.0391
```

```
Iteration: 5600; Percent complete: 46.7%; Average loss: 1.9981
Iteration: 5700; Percent complete: 47.5%; Average loss: 1.9735
Iteration: 5800; Percent complete: 48.3%; Average loss: 1.9402
Iteration: 5900; Percent complete: 49.2%; Average loss: 1.9202
Iteration: 6000; Percent complete: 50.0%; Average loss: 1.8856
content/cb_model/Chat/2-4_512
Iteration: 6100; Percent complete: 50.8%; Average loss: 1.8527
Iteration: 6200; Percent complete: 51.7%; Average loss: 1.8236
Iteration: 6300; Percent complete: 52.5%; Average loss: 1.7926
Iteration: 6400; Percent complete: 53.3%; Average loss: 1.7556
Iteration: 6500; Percent complete: 54.2%; Average loss: 1.7305
Iteration: 6600; Percent complete: 55.0%; Average loss: 1.6976
Iteration: 6700; Percent complete: 55.8%; Average loss: 1.6857
Iteration: 6800; Percent complete: 56.7%; Average loss: 1.6450
Iteration: 6900; Percent complete: 57.5%; Average loss: 1.6235
Iteration: 7000; Percent complete: 58.3%; Average loss: 1.6138
Iteration: 7100; Percent complete: 59.2%; Average loss: 1.5743
Iteration: 7200; Percent complete: 60.0%; Average loss: 1.5397
Iteration: 7300; Percent complete: 60.8%; Average loss: 1.5149
Iteration: 7400; Percent complete: 61.7%; Average loss: 1.5023
Iteration: 7500; Percent complete: 62.5%; Average loss: 1.4609
Iteration: 7600; Percent complete: 63.3%; Average loss: 1.4424
Iteration: 7700; Percent complete: 64.2%; Average loss: 1.4171
Iteration: 7800; Percent complete: 65.0%; Average loss: 1.3851
Iteration: 7900; Percent complete: 65.8%; Average loss: 1.3616
Iteration: 8000; Percent complete: 66.7%; Average loss: 1.3330
content/cb_model/Chat/2-4_512
Iteration: 8100; Percent complete: 67.5%; Average loss: 1.3193
Iteration: 8200; Percent complete: 68.3%; Average loss: 1.2993
Iteration: 8300; Percent complete: 69.2%; Average loss: 1.2645
Iteration: 8400; Percent complete: 70.0%; Average loss: 1.2376
Iteration: 8500; Percent complete: 70.8%; Average loss: 1.2215
Iteration: 8600; Percent complete: 71.7%; Average loss: 1.2134
Iteration: 8700; Percent complete: 72.5%; Average loss: 1.1757
Iteration: 8800; Percent complete: 73.3%; Average loss: 1.1644
Iteration: 8900; Percent complete: 74.2%; Average loss: 1.1472
Iteration: 9000; Percent complete: 75.0%; Average loss: 1.1251
Iteration: 9100; Percent complete: 75.8%; Average loss: 1.1132
Iteration: 9200; Percent complete: 76.7%; Average loss: 1.0802
Iteration: 9300; Percent complete: 77.5%; Average loss: 1.0645
Iteration: 9400; Percent complete: 78.3%; Average loss: 1.0478
Iteration: 9500; Percent complete: 79.2%; Average loss: 1.0277
Iteration: 9600; Percent complete: 80.0%; Average loss: 1.0020
Iteration: 9700; Percent complete: 80.8%; Average loss: 0.9879
Iteration: 9800; Percent complete: 81.7%; Average loss: 0.9736
Iteration: 9900; Percent complete: 82.5%; Average loss: 0.9519
Iteration: 10000; Percent complete: 83.3%; Average loss: 0.9386
content/cb_model/Chat/2-4_512
Iteration: 10100; Percent complete: 84.2%; Average loss: 0.9227
Iteration: 10200; Percent complete: 85.0%; Average loss: 0.9086
Iteration: 10300; Percent complete: 85.8%; Average loss: 0.8902
Iteration: 10400; Percent complete: 86.7%; Average loss: 0.8632
Iteration: 10500; Percent complete: 87.5%; Average loss: 0.8526
Iteration: 10600; Percent complete: 88.3%; Average loss: 0.8487
Iteration: 10700; Percent complete: 89.2%; Average loss: 0.8316
Iteration: 10800; Percent complete: 90.0%; Average loss: 0.8114
Iteration: 10900; Percent complete: 90.8%; Average loss: 0.8068
Iteration: 11000; Percent complete: 91.7%; Average loss: 0.7876
Iteration: 11100; Percent complete: 92.5%; Average loss: 0.7707
Iteration: 11200; Percent complete: 93.3%; Average loss: 0.7672
Iteration: 11300; Percent complete: 94.2%; Average loss: 0.7534
```

```
Iteration: 11400; Percent complete: 95.0%; Average loss: 0.7365
Iteration: 11500; Percent complete: 95.8%; Average loss: 0.7250
Iteration: 11600; Percent complete: 96.7%; Average loss: 0.7108
Iteration: 11700; Percent complete: 97.5%; Average loss: 0.6970
Iteration: 11800; Percent complete: 98.3%; Average loss: 0.6858
Iteration: 11900; Percent complete: 99.2%; Average loss: 0.6898
Iteration: 12000; Percent complete: 100.0%; Average loss: 0.6710
content/cb_model/Chat/2-4_512
0 0.08786571270524224 0.03403024420100487
1000 0.13560245731142448 0.054173508766110434
2000 0.13202144929246235 0.0547724869705568
3000 0.13083288721324685 0.054209945046619204
4000 0.133090986999077 0.05569587503441644
5000 0.13191480116890775 0.055359817765740615
6000 0.1312868774694944 0.05512274845234756
7000 0.14314621766285207 0.06683194587443625
8000 0.14180658707962898 0.06525827987485991
```

In this Beam Search Decoding, we calculated BLEU score for 1 gram and 2 gram which are 0.14142280489931194 and 0.06491881943777443 respectively.

```
Total Bleu Score for 1 grams on testing pairs:  0.14142280489931194
Total Bleu Score for 2 grams on testing pairs:  0.06491881943777443
```

Chat with the beam search bot.

```
> hello
Bot: do you want to see doc ?
> sure
Bot: do you want a perfect for it ?
> of course
Bot: if you trusted me
> yes
Bot: why ?
> because you are a bot
Error: Encountered unknown word.
>
```

Independent work

For the Independent work, there are two part:

- 1. Find another dataset of sentence pairs in a different domain and see if you can preprocess the data and train a chatbot model on it using the same code we developed today. Report your results.
- 2. Replace the LSTM encoder/decoder with a Transformer. Check out the PyTorch Transformer module documentation

Task1

For this work, I downloaded the sentence pairs dataset from [https://www.kaggle.com/code/alincijov/dialog-chatbot-using-bahdanau-attention/data?select=dialogs.txt (https://www.kaggle.com/code/alincijov/dialog-](https://www.kaggle.com/code/alincijov/dialog-chatbot-using-bahdanau-attention/data?select=dialogs.txt)

chatbot-using-bahdanau-attention/data?select=dialogs.txt)

The below cell is the result of data load and processing in Greedy Search Decoding

```
Start preparing training data ...
Reading lines...
Read 3725 sentence pairs
Trimmed to 2103 sentence pairs
Counting words...
Counted words: 1783

pairs:
['hi how are you doing ?', 'i m fine . how about yourself ?']
['i m fine . how about yourself ?', 'i m pretty good . thanks for asking .']
['i m pretty good . thanks for asking .', 'no problem . so how have you been ?']
['no problem . so how have you been ?', 'i ve been great . what about you ?']
['what school do you go to ?', 'i go to pcc .']
['i go to pcc .', 'do you like it there ?']
['good luck with school .', 'thank you very much .']
['how s it going ?', 'i m doing well . how about you ?']
['i m doing well . how about you ?', 'never better thanks .']
['never better thanks .', 'so how have you been lately ?']
keep_words 910 / 1780 = 0.5112
Trimmed from 2103 pairs to 1184, 0.5630 of total
[['i m fine . how about yourself ?', 'i m pretty good . thanks for asking .'], ['i
m pretty good . thanks for asking .', 'no problem . so how have you been ?'], ['no
problem . so how have you been ?', 'i ve been great . what about you ?'], ['what
school do you go to ?', 'i go to pcc .'], ['i go to pcc .', 'do you like it there
?']]
[['i m pretty good . thanks for asking .', 'no problem . so how have you been ?'],
['no problem . so how have you been ?', 'i ve been great . what about you ?'], ['i
m fine . how about yourself ?', 'i m pretty good . thanks for asking .'], ['what
school do you go to ?', 'i go to pcc .'], ['i go to pcc .', 'do you like it there
?']]
tensor([[ 19,  26,   8,  89, 306],
        [595,  39,  94, 615,  11],
        [  5,  98,   5, 120,   2],
        [ 76, 112, 190, 729,   0],
        [614,  32,  43, 106,   0],
        [ 11,   7,  11, 535,   0],
        [  2,   2,   2,  12,   0],
        [  0,   0,   0,  11,   0],
        [  0,   0,   0,   2,   0]])
tensor([[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True, False],
        [False, False, False,  True, False],
        [False, False, False,  True, False]])
```

We can see the training with 12000 iterations with Greedy Search Decoding and loss values were deceased.

```
Starting Training!
Initializing ...
```

```
Training...
Iteration: 100; Percent complete: 0.8%; Average loss: 4.7258
Iteration: 200; Percent complete: 1.7%; Average loss: 3.9227
Iteration: 300; Percent complete: 2.5%; Average loss: 3.2025
Iteration: 400; Percent complete: 3.3%; Average loss: 2.3496
Iteration: 500; Percent complete: 4.2%; Average loss: 1.5253
Iteration: 600; Percent complete: 5.0%; Average loss: 0.8843
Iteration: 700; Percent complete: 5.8%; Average loss: 0.4936
Iteration: 800; Percent complete: 6.7%; Average loss: 0.2895
Iteration: 900; Percent complete: 7.5%; Average loss: 0.1809
Iteration: 1000; Percent complete: 8.3%; Average loss: 0.1249
Iteration: 1100; Percent complete: 9.2%; Average loss: 0.0945
Iteration: 1200; Percent complete: 10.0%; Average loss: 0.0817
Iteration: 1300; Percent complete: 10.8%; Average loss: 0.0650
Iteration: 1400; Percent complete: 11.7%; Average loss: 0.0553
Iteration: 1500; Percent complete: 12.5%; Average loss: 0.0475
Iteration: 1600; Percent complete: 13.3%; Average loss: 0.0459
Iteration: 1700; Percent complete: 14.2%; Average loss: 0.0482
Iteration: 1800; Percent complete: 15.0%; Average loss: 0.0378
Iteration: 1900; Percent complete: 15.8%; Average loss: 0.0341
Iteration: 2000; Percent complete: 16.7%; Average loss: 0.0329
content/cb_model/Chat/2-4_512
Iteration: 2100; Percent complete: 17.5%; Average loss: 0.0304
Iteration: 2200; Percent complete: 18.3%; Average loss: 0.0291
Iteration: 2300; Percent complete: 19.2%; Average loss: 0.0285
Iteration: 2400; Percent complete: 20.0%; Average loss: 0.0306
Iteration: 2500; Percent complete: 20.8%; Average loss: 0.0273
Iteration: 2600; Percent complete: 21.7%; Average loss: 0.0254
Iteration: 2700; Percent complete: 22.5%; Average loss: 0.0238
Iteration: 2800; Percent complete: 23.3%; Average loss: 0.0239
Iteration: 2900; Percent complete: 24.2%; Average loss: 0.0235
Iteration: 3000; Percent complete: 25.0%; Average loss: 0.0262
Iteration: 3100; Percent complete: 25.8%; Average loss: 0.0490
Iteration: 3200; Percent complete: 26.7%; Average loss: 0.0307
Iteration: 3300; Percent complete: 27.5%; Average loss: 0.0236
Iteration: 3400; Percent complete: 28.3%; Average loss: 0.0249
Iteration: 3500; Percent complete: 29.2%; Average loss: 0.0225
Iteration: 3600; Percent complete: 30.0%; Average loss: 0.0221
Iteration: 3700; Percent complete: 30.8%; Average loss: 0.0213
Iteration: 3800; Percent complete: 31.7%; Average loss: 0.0212
Iteration: 3900; Percent complete: 32.5%; Average loss: 0.0212
Iteration: 4000; Percent complete: 33.3%; Average loss: 0.0202
content/cb_model/Chat/2-4_512
Iteration: 4100; Percent complete: 34.2%; Average loss: 0.0214
Iteration: 4200; Percent complete: 35.0%; Average loss: 0.0219
Iteration: 4300; Percent complete: 35.8%; Average loss: 0.0277
Iteration: 4400; Percent complete: 36.7%; Average loss: 0.0306
Iteration: 4500; Percent complete: 37.5%; Average loss: 0.0325
Iteration: 4600; Percent complete: 38.3%; Average loss: 0.0253
Iteration: 4700; Percent complete: 39.2%; Average loss: 0.0222
Iteration: 4800; Percent complete: 40.0%; Average loss: 0.0211
Iteration: 4900; Percent complete: 40.8%; Average loss: 0.0205
Iteration: 5000; Percent complete: 41.7%; Average loss: 0.0201
Iteration: 5100; Percent complete: 42.5%; Average loss: 0.0211
Iteration: 5200; Percent complete: 43.3%; Average loss: 0.0224
Iteration: 5300; Percent complete: 44.2%; Average loss: 0.0207
Iteration: 5400; Percent complete: 45.0%; Average loss: 0.0204
Iteration: 5500; Percent complete: 45.8%; Average loss: 0.0207
Iteration: 5600; Percent complete: 46.7%; Average loss: 0.0205
Iteration: 5700; Percent complete: 47.5%; Average loss: 0.0202
Iteration: 5800; Percent complete: 48.3%; Average loss: 0.0203
```

```
Iteration: 5900; Percent complete: 49.2%; Average loss: 0.0192
Iteration: 6000; Percent complete: 50.0%; Average loss: 0.0197
content/cb_model/Chat/2-4_512
Iteration: 6100; Percent complete: 50.8%; Average loss: 0.0195
Iteration: 6200; Percent complete: 51.7%; Average loss: 0.0205
Iteration: 6300; Percent complete: 52.5%; Average loss: 0.0249
Iteration: 6400; Percent complete: 53.3%; Average loss: 0.0361
Iteration: 6500; Percent complete: 54.2%; Average loss: 0.0254
Iteration: 6600; Percent complete: 55.0%; Average loss: 0.0214
Iteration: 6700; Percent complete: 55.8%; Average loss: 0.0209
Iteration: 6800; Percent complete: 56.7%; Average loss: 0.0222
Iteration: 6900; Percent complete: 57.5%; Average loss: 0.0204
Iteration: 7000; Percent complete: 58.3%; Average loss: 0.0202
Iteration: 7100; Percent complete: 59.2%; Average loss: 0.0205
Iteration: 7200; Percent complete: 60.0%; Average loss: 0.0195
Iteration: 7300; Percent complete: 60.8%; Average loss: 0.0192
Iteration: 7400; Percent complete: 61.7%; Average loss: 0.0191
Iteration: 7500; Percent complete: 62.5%; Average loss: 0.0198
Iteration: 7600; Percent complete: 63.3%; Average loss: 0.0194
Iteration: 7700; Percent complete: 64.2%; Average loss: 0.0184
Iteration: 7800; Percent complete: 65.0%; Average loss: 0.0200
Iteration: 7900; Percent complete: 65.8%; Average loss: 0.0192
Iteration: 8000; Percent complete: 66.7%; Average loss: 0.0210
content/cb_model/Chat/2-4_512
Iteration: 8100; Percent complete: 67.5%; Average loss: 0.0248
Iteration: 8200; Percent complete: 68.3%; Average loss: 0.0226
Iteration: 8300; Percent complete: 69.2%; Average loss: 0.0205
Iteration: 8400; Percent complete: 70.0%; Average loss: 0.0197
Iteration: 8500; Percent complete: 70.8%; Average loss: 0.0191
Iteration: 8600; Percent complete: 71.7%; Average loss: 0.0192
Iteration: 8700; Percent complete: 72.5%; Average loss: 0.0197
Iteration: 8800; Percent complete: 73.3%; Average loss: 0.0184
Iteration: 8900; Percent complete: 74.2%; Average loss: 0.0193
Iteration: 9000; Percent complete: 75.0%; Average loss: 0.0195
Iteration: 9100; Percent complete: 75.8%; Average loss: 0.0292
Iteration: 9200; Percent complete: 76.7%; Average loss: 0.0235
Iteration: 9300; Percent complete: 77.5%; Average loss: 0.0191
Iteration: 9400; Percent complete: 78.3%; Average loss: 0.0196
Iteration: 9500; Percent complete: 79.2%; Average loss: 0.0189
Iteration: 9600; Percent complete: 80.0%; Average loss: 0.0185
Iteration: 9700; Percent complete: 80.8%; Average loss: 0.0192
Iteration: 9800; Percent complete: 81.7%; Average loss: 0.0193
Iteration: 9900; Percent complete: 82.5%; Average loss: 0.0187
Iteration: 10000; Percent complete: 83.3%; Average loss: 0.0192
content/cb_model/Chat/2-4_512
Iteration: 10100; Percent complete: 84.2%; Average loss: 0.0186
Iteration: 10200; Percent complete: 85.0%; Average loss: 0.0185
Iteration: 10300; Percent complete: 85.8%; Average loss: 0.0185
Iteration: 10400; Percent complete: 86.7%; Average loss: 0.0192
Iteration: 10500; Percent complete: 87.5%; Average loss: 0.0187
Iteration: 10600; Percent complete: 88.3%; Average loss: 0.0188
Iteration: 10700; Percent complete: 89.2%; Average loss: 0.0196
Iteration: 10800; Percent complete: 90.0%; Average loss: 0.0200
Iteration: 10900; Percent complete: 90.8%; Average loss: 0.0198
Iteration: 11000; Percent complete: 91.7%; Average loss: 0.0206
Iteration: 11100; Percent complete: 92.5%; Average loss: 0.0266
Iteration: 11200; Percent complete: 93.3%; Average loss: 0.0227
Iteration: 11300; Percent complete: 94.2%; Average loss: 0.0192
Iteration: 11400; Percent complete: 95.0%; Average loss: 0.0185
Iteration: 11500; Percent complete: 95.8%; Average loss: 0.0194
Iteration: 11600; Percent complete: 96.7%; Average loss: 0.0185
```

```
Iteration: 11700; Percent complete: 97.5%; Average loss: 0.0186
Iteration: 11800; Percent complete: 98.3%; Average loss: 0.0193
Iteration: 11900; Percent complete: 99.2%; Average loss: 0.0182
Iteration: 12000; Percent complete: 100.0%; Average loss: 0.0189
content/cb_model/Chat/2-4_512
```

We calculated BLEU score for 1 gram and 2 gram which are 0.15784220489319176 and 0.06314280893119497 respectively.

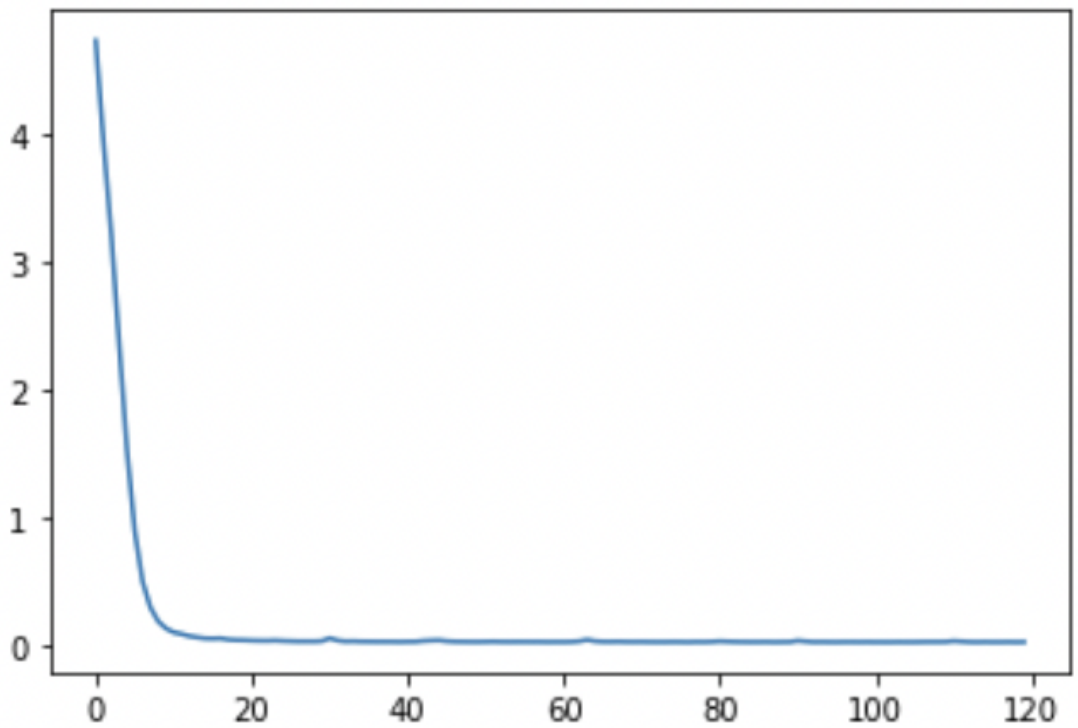
```
Total Bleu Score for 1 grams on testing pairs:  0.15784220489319176
Total Bleu Score for 2 grams on testing pairs:  0.06314280893119497
```

We can see the following plotting

In [59]:

```
from IPython.display import Image
Image("loss.png")
```

Out[59]:



Task2

In this task, seq2seq model was trained on Transformer.

Data Sourcing and Processing

For data sourcing and processing, I used tochtext library to access Multi30k dataset to train a German to English language translation model, insted of using dialuoge seq2seq datasets. Moreover, Multi30k dataset from torchtext library yields a pair of source-target raw sentences.

In [60]:

```
import os
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

In [61]:

```
#!pip install -U spacy
```

In [55]:

```
#!python -m spacy download en_core_web_sm
```

In [54]:

```
#!python -m spacy download de_core_news_sm
```

In [7]:

```
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.datasets import Multi30k
from typing import Iterable, List


SRC_LANGUAGE = 'de'
TGT_LANGUAGE = 'en'

# Place-holders
token_transform = {}
vocab_transform = {}


# Create source and target language tokenizer. Make sure to install the dependencies
# pip install -U spacy
# python -m spacy download en_core_web_sm
# python -m spacy download de_core_news_sm
token_transform[SRC_LANGUAGE] = get_tokenizer('spacy', language='de_core_news_sm')
token_transform[TGT_LANGUAGE] = get_tokenizer('spacy', language='en_core_web_sm')


# helper function to yield list of tokens
def yield_tokens(data_iter: Iterable, language: str) -> List[str]:
    language_index = {SRC_LANGUAGE: 0, TGT_LANGUAGE: 1}

    for data_sample in data_iter:
        yield token_transform[language](data_sample[language_index[language]])

# Define special symbols and indices
UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX = 0, 1, 2, 3
# Make sure the tokens are in order of their indices to properly insert them in voca
special_symbols = ['<unk>', '<pad>', '<bos>', '<eos>']

for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    # Training data Iterator
    train_iter = Multi30k(split='train', language_pair=(SRC_LANGUAGE, TGT_LANGUAGE))
    # Create torchtext's Vocab object
    vocab_transform[ln] = build_vocab_from_iterator(yield_tokens(train_iter, ln),
                                                    min_freq=1,
                                                    specials=special_symbols,
                                                    special_first=True)

# Set UNK_IDX as the default index. This index is returned when the token is not fou
# If not set, it throws RuntimeError when the queried token is not found in the Voca
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    vocab_transform[ln].set_default_index(UNK_IDX)
```

Transformers

The follwoing seq2seq network using transformer consists of three parts.

- 1.Firstly, the embedding layer :
  - This layer converts tensor of input indices into corresponding tensor of input embeddings.
  - These embedding are further augmented with positional encodings to provide position information of input tokens to the model.

- 2. Transformer model.
- 3. Finally, the output of Transformer model is passed through linear layer that give un-normalized probabilities for each token in the target language.

In [12]:

```
from torch import Tensor
import torch
import torch.nn as nn
from torch.nn import Transformer
import math
DEVICE = torch.device('cuda:1' if torch.cuda.is_available() else 'cpu')

# helper Module that adds positional encoding to the token embedding to introduce a
class PositionalEncoding(nn.Module):
    def __init__(self,
                 emb_size: int,
                 dropout: float,
                 maxlen: int = 5000):
        super(PositionalEncoding, self).__init__()
        den = torch.exp(- torch.arange(0, emb_size, 2)* math.log(10000) / emb_size)
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        pos_embedding = torch.zeros((maxlen, emb_size))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding: Tensor):
        return self.dropout(token_embedding + self.pos_embedding[:token_embedding.si

# helper Module to convert tensor of input indices into corresponding tensor of toke
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size

    def forward(self, tokens: Tensor):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)

# Seq2Seq Network
class Seq2SeqTransformer(nn.Module):
    def __init__(self,
                 num_encoder_layers: int,
                 num_decoder_layers: int,
                 emb_size: int,
                 nhead: int,
                 src_vocab_size: int,
                 tgt_vocab_size: int,
                 dim_feedforward: int = 512,
                 dropout: float = 0.1):
        super(Seq2SeqTransformer, self).__init__()
        self.transformer = Transformer(d_model=emb_size,
                                       nhead=nhead,
                                       num_encoder_layers=num_encoder_layers,
                                       num_decoder_layers=num_decoder_layers,
                                       dim_feedforward=dim_feedforward,
                                       dropout=dropout)
        self.generator = nn.Linear(emb_size, tgt_vocab_size)
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(
```

```
            emb_size, dropout=dropout)

    def forward(self,
                src: Tensor,
                trg: Tensor,
                src_mask: Tensor,
                tgt_mask: Tensor,
                src_padding_mask: Tensor,
                tgt_padding_mask: Tensor,
                memory_key_padding_mask: Tensor):
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        outs = self.transformer(src_emb, tgt_emb, src_mask, tgt_mask, None,
                                src_padding_mask, tgt_padding_mask, memory_key_paddi
        return self.generator(outs)

    def encode(self, src: Tensor, src_mask: Tensor):
        return self.transformer.encoder(self.positional_encoding(
                             self.src_tok_emb(src)), src_mask)

    def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):
        return self.transformer.decoder(self.positional_encoding(
                          self.tgt_tok_emb(tgt)), memory,
                          tgt_mask)
```

During training, a subsequent word mask is needed to prevent model to look into the future words when making predictions. Furthermorw, masks are needed to hide source and target padding tokens.

In [13]:

```
def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1,
    return mask


def create_mask(src, tgt):
    src_seq_len = src.shape[0]
    tgt_seq_len = tgt.shape[0]

    tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
    src_mask = torch.zeros((src_seq_len, src_seq_len),device=DEVICE).type(torch.bool

    src_padding_mask = (src == PAD_IDX).transpose(0, 1)
    tgt_padding_mask = (tgt == PAD_IDX).transpose(0, 1)
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

Then, our loss function is the cross-entropy loss and the optmizer is used for training.

In [14]:

```
torch.manual_seed(0)

SRC_VOCAB_SIZE = len(vocab_transform[SRC_LANGUAGE])
TGT_VOCAB_SIZE = len(vocab_transform[TGT_LANGUAGE])
EMB_SIZE = 512
NHEAD = 8
FFN_HID_DIM = 512
BATCH_SIZE = 128
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3

transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS, EMB_SIZE,
                                 NHEAD, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, FFN_HID_DIM)

for p in transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)

transformer = transformer.to(DEVICE)

loss_fn = torch.nn.CrossEntropyLoss(ignore_index=PAD_IDX)

optimizer = torch.optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98),
```

In [15]:

```
from torch.nn.utils.rnn import pad_sequence

# helper function to club together sequential operations
def sequential_transforms(*transforms):
    def func(txt_input):
        for transform in transforms:
            txt_input = transform(txt_input)
        return txt_input
    return func

# function to add BOS/EOS and create tensor for input sequence indices
def tensor_transform(token_ids: List[int]):
    return torch.cat((torch.tensor([BOS_IDX]),
                      torch.tensor(token_ids),
                      torch.tensor([EOS_IDX])))

# src and tgt language text transforms to convert raw strings into tensors indices
text_transform = {}
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    text_transform[ln] = sequential_transforms(token_transform[ln], #Tokenization
                                               vocab_transform[ln], #Numericalizatio
                                               tensor_transform) # Add BOS/EOS and c


# function to collate data samples into batch tesors
def collate_fn(batch):
    src_batch, tgt_batch = [], []
    for src_sample, tgt_sample in batch:
        src_batch.append(text_transform[SRC_LANGUAGE](src_sample.rstrip("\n")))
        tgt_batch.append(text_transform[TGT_LANGUAGE](tgt_sample.rstrip("\n")))

    src_batch = pad_sequence(src_batch, padding_value=PAD_IDX)
    tgt_batch = pad_sequence(tgt_batch, padding_value=PAD_IDX)
    return src_batch, tgt_batch
```

Training and Evaluation

In [17]:

```
from torch.utils.data import DataLoader

def train_epoch(model, optimizer):
    model.train()
    losses = 0
    train_iter = Multi30k(split='train', language_pair=(SRC_LANGUAGE, TGT_LANGUAGE))
    train_dataloader = DataLoader(train_iter, batch_size=BATCH_SIZE, collate_fn=coll

    for src, tgt in train_dataloader:
        src = src.to(DEVICE)
        tgt = tgt.to(DEVICE)

        tgt_input = tgt[:-1, :]

        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tg

        logits = model(src, tgt_input, src_mask, tgt_mask,src_padding_mask, tgt_padd

        optimizer.zero_grad()

        tgt_out = tgt[1:, :]
        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))
        loss.backward()

        optimizer.step()
        losses += loss.item()

    return losses / len(train_dataloader)


def evaluate(model):
    model.eval()
    losses = 0

    val_iter = Multi30k(split='valid', language_pair=(SRC_LANGUAGE, TGT_LANGUAGE))
    val_dataloader = DataLoader(val_iter, batch_size=BATCH_SIZE, collate_fn=collate_

    for src, tgt in val_dataloader:
        src = src.to(DEVICE)
        tgt = tgt.to(DEVICE)

        tgt_input = tgt[:-1, :]

        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tg

        logits = model(src, tgt_input, src_mask, tgt_mask,src_padding_mask, tgt_padd

        tgt_out = tgt[1:, :]
        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))
        losses += loss.item()

    return losses / len(val_dataloader)
```

In [19]:

```
from timeit import default_timer as timer
NUM_EPOCHS = 20
train_losses = []
val_losses = []

for epoch in range(1, NUM_EPOCHS+1):
    start_time = timer()
    train_loss = train_epoch(transformer, optimizer)
    train_losses.append(train_loss)

    end_time = timer()
    val_loss = evaluate(transformer)
    val_losses.append(val_loss)


    print((f"Epoch: {epoch}, Train loss: {train_loss:.3f}, Val loss: {val_loss:.3f},
```

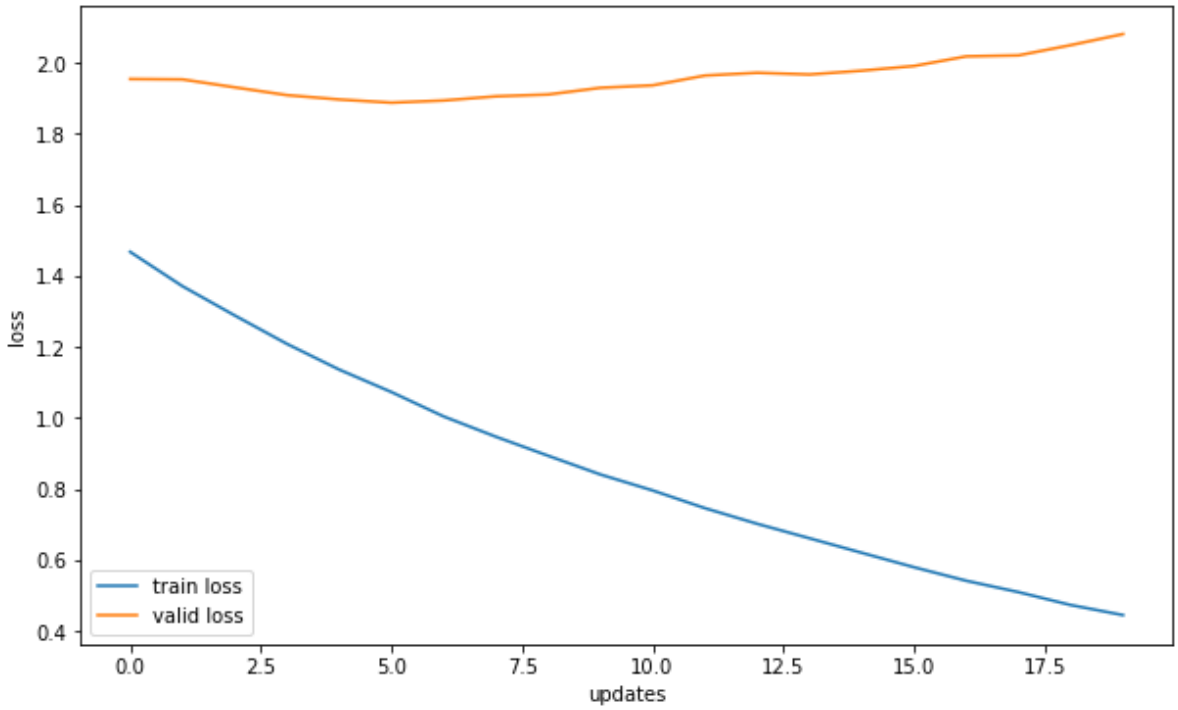
```
Epoch: 1, Train loss: 1.467, Val loss: 1.954, Epoch time = 25.058s
Epoch: 2, Train loss: 1.370, Val loss: 1.953, Epoch time = 24.299s
Epoch: 3, Train loss: 1.288, Val loss: 1.930, Epoch time = 23.925s
Epoch: 4, Train loss: 1.208, Val loss: 1.908, Epoch time = 22.763s
Epoch: 5, Train loss: 1.136, Val loss: 1.896, Epoch time = 23.761s
Epoch: 6, Train loss: 1.072, Val loss: 1.887, Epoch time = 22.663s
Epoch: 7, Train loss: 1.004, Val loss: 1.893, Epoch time = 24.857s
Epoch: 8, Train loss: 0.946, Val loss: 1.905, Epoch time = 24.518s
Epoch: 9, Train loss: 0.893, Val loss: 1.910, Epoch time = 24.933s
Epoch: 10, Train loss: 0.840, Val loss: 1.929, Epoch time = 25.362s
Epoch: 11, Train loss: 0.795, Val loss: 1.936, Epoch time = 25.092s
Epoch: 12, Train loss: 0.745, Val loss: 1.963, Epoch time = 23.590s
Epoch: 13, Train loss: 0.701, Val loss: 1.971, Epoch time = 24.620s
Epoch: 14, Train loss: 0.661, Val loss: 1.966, Epoch time = 24.583s
Epoch: 15, Train loss: 0.620, Val loss: 1.977, Epoch time = 22.752s
Epoch: 16, Train loss: 0.579, Val loss: 1.990, Epoch time = 22.739s
Epoch: 17, Train loss: 0.541, Val loss: 2.017, Epoch time = 22.922s
Epoch: 18, Train loss: 0.509, Val loss: 2.020, Epoch time = 23.407s
Epoch: 19, Train loss: 0.473, Val loss: 2.049, Epoch time = 23.756s
Epoch: 20, Train loss: 0.444, Val loss: 2.080, Epoch time = 24.167s
```

In [21]:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(train_losses, label = 'train loss')
ax.plot(val_losses, label = 'valid loss')
plt.legend()
ax.set_xlabel('updates')
ax.set_ylabel('loss')
```

Out[21]:

```
Text(0, 0.5, 'loss')
```



In [22]:

```
te output sequence using greedy algorithm
del, src, src_mask, max_len, start_symbol):
ICE)
ask.to(DEVICE)

ncode(src, src_mask)
1, 1).fill_(start_symbol).type(torch.long).to(DEVICE)
ax_len-1):
ory.to(DEVICE)
generate_square_subsequent_mask(ys.size(0))
.type(torch.bool)).to(DEVICE)
decode(ys, memory, tgt_mask)
anspose(0, 1)
.generator(out[:, -1])
 = torch.max(prob, dim=1)
next_word.item()

at([ys,
    torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=0)
 == EOS_IDX:



 translate input sentence into target language
 torch.nn.Module, src_sentence: str):

form[SRC_LANGUAGE](src_sentence).view(-1, 1)
.shape[0]
n.zeros(num_tokens, num_tokens)).type(torch.bool)
edy_decode(
 src_mask, max_len=num_tokens + 5, start_symbol=BOS_IDX).flatten()
vocab_transform[TGT_LANGUAGE].lookup_tokens(list(tgt_tokens.cpu().numpy()))).replace(
```

In [23]:

```
print(translate(transformer, "Eine Gruppe von Menschen steht vor einem Iglu ."))
```

```
 A group of people in an igloo
```

In [24]:

```
print(translate(transformer, "Guten Morgen"))
```

```
 morning morning
```

In [25]:

```
print(translate(transformer, "Mein Name ist Mondly"))
```

```
 The professional player is just behind the pin .
```

In [26]:

```
print(translate(transformer, "Ich möchte ein Bier"))
```

```
 I this man is driving a beer .
```

Ref : https://pytorch.org/tutorials/beginner/translation_transformer.html?highlight=transformer (https://pytorch.org/tutorials/beginner/translation_transformer.html?highlight=transformer)

In [ ]:

```
```

In [ ]:

```
```