assignment2_starter_code

March 31, 2023

1 NN-Based Language Model

In this excercise we will run a basic RNN based language model and answer some questions about the code. It is advised to use GPU to run the code. First run the code then answer the questions below that require modifying it.

[ ]:
```
#@title   Imports & Hyperparameter Setup
#@markdown Feel free to experiment with the following hyperparameters at your
#@markdown leasure. For the purpose of this assignment, leave the default values
#@markdown and run the code with these suggested values.
# Some part of the code was referenced from below.
# https://github.com/pytorch/examples/tree/master/word_language_model
# https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/
 ↪02-intermediate/language_model

! git clone https://github.com/yunjey/pytorch-tutorial/
%cd pytorch-tutorial/tutorials/02-intermediate/language_model/

import torch
import torch.nn as nn
import numpy as np
from torch.nn.utils import clip_grad_norm_

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
embed_size = 128 #@param {type:"number"}
hidden_size = 1024 #@param {type:"number"}
num_layers = 1 #@param {type:"number"}
num_epochs = 5 #@param {type:"slider", min:1, max:10, step:1}
batch_size = 20 #@param {type:"number"}
seq_length = 30 #@param {type:"number"}
learning_rate = 0.002 #@param {type:"number"}
#@markdown Number of words to be sampled
num_samples = 50 #@param {type:"number"}

print(f"--> Device selected: {device}")
```

```
c:\Users\x_zhu202\Documents\GitHub\COMP691_LABS\pytorch-
tutorial\tutorials\02-intermediate\language_model
--> Device selected: cuda
```

```
fatal: destination path 'pytorch-tutorial' already exists and is not an empty
directory.
```

[ ]:
```
from data_utils import Dictionary, Corpus

# Load "Penn Treebank" dataset
corpus = Corpus()
ids = corpus.get_data('data/train.txt', batch_size)
vocab_size = len(corpus.dictionary)
num_batches = ids.size(1) // seq_length

print(f"Vcoabulary size: {vocab_size}")
print(f"Number of batches: {num_batches}")
```

```
Vcoabulary size: 10000
Number of batches: 1549
```

1.1 Model Definition

As you can see below, this model stacks `num_layers` many LSTM units vertically to construct our basic RNN-based language model. The diagram below shows a pictorial representation of the model in its simplest form (i.e `num_layers`=1).

[ ]:
```
# RNN based language model
class RNNLM(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
        super(RNNLM, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, h):
        # Embed word ids to vectors
        x = self.embed(x)

        # Forward propagate LSTM
        out, (h, c) = self.lstm(x, h)

        # Reshape output to (batch_size*sequence_length, hidden_size)
        out = out.reshape(out.size(0)*out.size(1), out.size(2))

        # Decode hidden states of all time steps
        out = self.linear(out)
        return out, (h, c)
```

1.2 Training

In this section we will train our model, this should take a couple of minutes! Be patient

[ ]:
```python
model = RNNLM(vocab_size, embed_size, hidden_size, num_layers).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Truncated backpropagation
def detach(states):
    return [state.detach() for state in states]


# Train the model
for epoch in range(num_epochs):
    # Set initial hidden and cell states
    states = (torch.zeros(num_layers, batch_size, hidden_size).to(device),
              torch.zeros(num_layers, batch_size, hidden_size).to(device))

    for i in range(0, ids.size(1) - seq_length, seq_length):
        # Get mini-batch inputs and targets
        inputs = ids[:, i:i+seq_length].to(device)
        targets = ids[:, (i+1):(i+1)+seq_length].to(device)

        # Forward pass
        states = detach(states)
        outputs, states = model(inputs, states)
        loss = criterion(outputs, targets.reshape(-1))

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        clip_grad_norm_(model.parameters(), 0.5)
        optimizer.step()

        step = (i+1) // seq_length
        if step % 100 == 0:
            print ('Epoch [{}/{}], Step[{}/{}], Loss: {:.4f}, Perplexity: {:5.
 ↪2f}'
                   .format(epoch+1, num_epochs, step, num_batches, loss.item(),␣
 ↪np.exp(loss.item())))
```

```
Epoch [1/5], Step[0/1549], Loss: 9.2069, Perplexity: 9965.54
Epoch [1/5], Step[100/1549], Loss: 6.0190, Perplexity: 411.16
Epoch [1/5], Step[200/1549], Loss: 5.9211, Perplexity: 372.81
Epoch [1/5], Step[300/1549], Loss: 5.7155, Perplexity: 303.54
Epoch [1/5], Step[400/1549], Loss: 5.6774, Perplexity: 292.19
Epoch [1/5], Step[500/1549], Loss: 5.1143, Perplexity: 166.39
Epoch [1/5], Step[600/1549], Loss: 5.2022, Perplexity: 181.68
Epoch [1/5], Step[700/1549], Loss: 5.3083, Perplexity: 202.01
Epoch [1/5], Step[800/1549], Loss: 5.2033, Perplexity: 181.86
Epoch [1/5], Step[900/1549], Loss: 5.0703, Perplexity: 159.23
Epoch [1/5], Step[1000/1549], Loss: 5.1053, Perplexity: 164.90
Epoch [1/5], Step[1100/1549], Loss: 5.3635, Perplexity: 213.47
Epoch [1/5], Step[1200/1549], Loss: 5.1588, Perplexity: 173.95
Epoch [1/5], Step[1300/1549], Loss: 5.1094, Perplexity: 165.56
Epoch [1/5], Step[1400/1549], Loss: 4.8076, Perplexity: 122.44
Epoch [1/5], Step[1500/1549], Loss: 5.1488, Perplexity: 172.23
Epoch [2/5], Step[0/1549], Loss: 5.4710, Perplexity: 237.69
Epoch [2/5], Step[100/1549], Loss: 4.5282, Perplexity: 92.60
Epoch [2/5], Step[200/1549], Loss: 4.7244, Perplexity: 112.66
Epoch [2/5], Step[300/1549], Loss: 4.7133, Perplexity: 111.41
Epoch [2/5], Step[400/1549], Loss: 4.5950, Perplexity: 98.98
Epoch [2/5], Step[500/1549], Loss: 4.0911, Perplexity: 59.81
Epoch [2/5], Step[600/1549], Loss: 4.4466, Perplexity: 85.33
Epoch [2/5], Step[700/1549], Loss: 4.3328, Perplexity: 76.16
Epoch [2/5], Step[800/1549], Loss: 4.4553, Perplexity: 86.08
Epoch [2/5], Step[900/1549], Loss: 4.2215, Perplexity: 68.13
Epoch [2/5], Step[1000/1549], Loss: 4.3278, Perplexity: 75.78
Epoch [2/5], Step[1100/1549], Loss: 4.5604, Perplexity: 95.62
Epoch [2/5], Step[1200/1549], Loss: 4.4022, Perplexity: 81.63
Epoch [2/5], Step[1300/1549], Loss: 4.2522, Perplexity: 70.26
Epoch [2/5], Step[1400/1549], Loss: 3.9612, Perplexity: 52.52
Epoch [2/5], Step[1500/1549], Loss: 4.3102, Perplexity: 74.45
Epoch [3/5], Step[0/1549], Loss: 4.4508, Perplexity: 85.69
Epoch [3/5], Step[100/1549], Loss: 3.8243, Perplexity: 45.80
Epoch [3/5], Step[200/1549], Loss: 4.0313, Perplexity: 56.34
Epoch [3/5], Step[300/1549], Loss: 4.0065, Perplexity: 54.95
Epoch [3/5], Step[400/1549], Loss: 3.8685, Perplexity: 47.87
Epoch [3/5], Step[500/1549], Loss: 3.4581, Perplexity: 31.76
Epoch [3/5], Step[600/1549], Loss: 3.7979, Perplexity: 44.61
Epoch [3/5], Step[700/1549], Loss: 3.6312, Perplexity: 37.76
Epoch [3/5], Step[800/1549], Loss: 3.7922, Perplexity: 44.35
Epoch [3/5], Step[900/1549], Loss: 3.5325, Perplexity: 34.21
Epoch [3/5], Step[1000/1549], Loss: 3.6896, Perplexity: 40.03
Epoch [3/5], Step[1100/1549], Loss: 3.7728, Perplexity: 43.50
Epoch [3/5], Step[1200/1549], Loss: 3.7255, Perplexity: 41.49
Epoch [3/5], Step[1300/1549], Loss: 3.4818, Perplexity: 32.52
Epoch [3/5], Step[1400/1549], Loss: 3.2329, Perplexity: 25.35
Epoch [3/5], Step[1500/1549], Loss: 3.6102, Perplexity: 36.97
Epoch [4/5], Step[0/1549], Loss: 3.6704, Perplexity: 39.27
Epoch [4/5], Step[100/1549], Loss: 3.2356, Perplexity: 25.42
Epoch [4/5], Step[200/1549], Loss: 3.4683, Perplexity: 32.08
Epoch [4/5], Step[300/1549], Loss: 3.4890, Perplexity: 32.75
Epoch [4/5], Step[400/1549], Loss: 3.4009, Perplexity: 29.99
Epoch [4/5], Step[500/1549], Loss: 2.9435, Perplexity: 18.98
Epoch [4/5], Step[600/1549], Loss: 3.3262, Perplexity: 27.83
Epoch [4/5], Step[700/1549], Loss: 3.1705, Perplexity: 23.82
Epoch [4/5], Step[800/1549], Loss: 3.3215, Perplexity: 27.70
Epoch [4/5], Step[900/1549], Loss: 3.0867, Perplexity: 21.90
Epoch [4/5], Step[1000/1549], Loss: 3.2723, Perplexity: 26.37
Epoch [4/5], Step[1100/1549], Loss: 3.2085, Perplexity: 24.74
Epoch [4/5], Step[1200/1549], Loss: 3.1873, Perplexity: 24.22
Epoch [4/5], Step[1300/1549], Loss: 3.0351, Perplexity: 20.80
Epoch [4/5], Step[1400/1549], Loss: 2.6961, Perplexity: 14.82
Epoch [4/5], Step[1500/1549], Loss: 3.1527, Perplexity: 23.40
Epoch [5/5], Step[0/1549], Loss: 3.2509, Perplexity: 25.81
Epoch [5/5], Step[100/1549], Loss: 2.8687, Perplexity: 17.61
Epoch [5/5], Step[200/1549], Loss: 3.0505, Perplexity: 21.13
Epoch [5/5], Step[300/1549], Loss: 3.1495, Perplexity: 23.32
Epoch [5/5], Step[400/1549], Loss: 3.0962, Perplexity: 22.11
Epoch [5/5], Step[500/1549], Loss: 2.6367, Perplexity: 13.97
Epoch [5/5], Step[600/1549], Loss: 2.9966, Perplexity: 20.02
Epoch [5/5], Step[700/1549], Loss: 2.8827, Perplexity: 17.86
Epoch [5/5], Step[800/1549], Loss: 2.9748, Perplexity: 19.59
Epoch [5/5], Step[900/1549], Loss: 2.7764, Perplexity: 16.06
Epoch [5/5], Step[1000/1549], Loss: 2.8969, Perplexity: 18.12
Epoch [5/5], Step[1100/1549], Loss: 2.9826, Perplexity: 19.74
Epoch [5/5], Step[1200/1549], Loss: 2.8895, Perplexity: 17.98
Epoch [5/5], Step[1300/1549], Loss: 2.6705, Perplexity: 14.45
Epoch [5/5], Step[1400/1549], Loss: 2.3726, Perplexity: 10.73
Epoch [5/5], Step[1500/1549], Loss: 2.9207, Perplexity: 18.55
```

2 Questions

2.1 1 Q2.1 Detaching or not? (10 points)

The above code implements a version of truncated backpropagation through time. The implementation only requires the `detach()` function (lines 7-9 of the cell) defined above the loop and used once inside the training loop. * Explain the implementation (compared to not using truncated backprop through time). * What does the `detach()` call here achieve? Draw a computational graph. You may choose to answer this question outside the notebook. * When using using line 7-9 we will typically observe less GPU memory being used during training, explain why in your answer.

```
[ ]: print ("Answered in the above PDF. Outside of the notebook.")
```

```
Answered in the above PDF. Outside of the notebook.
```

2.2 Model Prediction

Below we will use our model to generate text sequence!

```
[ ]: # Sample from the model
     with torch.no_grad():
         with open('sample.txt', 'w') as f:
             # Set intial hidden ane cell states
             state = (torch.zeros(num_layers, 1, hidden_size).to(device),
                      torch.zeros(num_layers, 1, hidden_size).to(device))

             # Select one word id randomly
             prob = torch.ones(vocab_size)
             input = torch.multinomial(prob, num_samples=1).unsqueeze(1).to(device)

             for i in range(num_samples):
                 # Forward propagate RNN
                 output, state = model(input, state)

                 # Sample a word id
                 prob = output.exp()
                 word_id = torch.multinomial(prob, num_samples=1).item()

                 # Fill input with sampled word id for the next time step
                 input.fill_(word_id)

                 # File write
                 word = corpus.dictionary.idx2word[word_id]
                 word = '\n' if word == '<eos>' else word + ' '
                 f.write(word)

                 if (i+1) % 100 == 0:
                     print('Sampled [{}/{}] words and save to {}'.format(i+1,␣
       ↪num_samples, 'sample.txt'))
     ! type sample.txt
```

```
had little more than N years
but the report 's current account seems to be this is likely to <unk> because it
tends to enter out office of the cars from the u.s.
cancer patients clearly shortages giving the studio mr. reitman 's
it is n't starting
```

2.3 2 Q2.2 Sampling strategy (7 points)

Consider the sampling procedure above. The current code samples a word:

`word_id = torch.multinomial(prob, num_samples=1).item()`

in order to feed the model at each output step and feeding those to the next timestep. Copy below the above cell and modify this sampling startegy to use a greedy sampling which selects the highest probability word at each time step to feed as the next input.

```
[ ]: # Sample greedily from the model
     # Sample from the model using greedy sampling
     with torch.no_grad():
         with open('sample_greedy.txt', 'w') as f:
             # Set initial hidden and cell states
             state = (torch.zeros(num_layers, 1, hidden_size).to(device),
                      torch.zeros(num_layers, 1, hidden_size).to(device))

             # Select one word id randomly
             prob = torch.ones(vocab_size)
             input = torch.multinomial(prob, num_samples=1).unsqueeze(1).to(device)

             for i in range(num_samples):
                 # Forward propagate RNN
                 output, state = model(input, state)

                 # Sample a word id using greedy approach
                 prob = output.exp()
                 word_id = torch.argmax(prob, dim=-1).item()

                 # Fill input with sampled word id for the next time step
                 input.fill_(word_id)

                 # File write
                 word = corpus.dictionary.idx2word[word_id]
                 word = '\n' if word == '<eos>' else word + ' '
                 f.write(word)

                 if (i+1) % 100 == 0:
                     print('Sampled [{}/{}] words and save to {}'.format(i+1,␣
      ↪num_samples, 'sample_greedy.txt'))
     ! type sample_greedy.txt
```

```
's unexpected strengthening on the economy and the economy
the index which uses the dollar began at N down N
the index was N N
the index registered N in august compared with N in july and N
the index registered N in august with the
```

2.4 3 Q2.3 Embedding Distance (8 points)

Our model has learned a specific set of word embeddings. * Write a function that takes in 2 words and prints the cosine distance between their embeddings using the word embeddings from the above models. * Use it to print the cosine distance of the word "army" and the word "taxpayer".

Refer to the sampling code for how to output the words corresponding to each index. To get the index you can use the function `corpus.dictionary.word2idx`*.*

[ ]:
```python
# Embedding distance
import torch.nn.functional as F

def get_cosine_distance(a, b,model):
    idx1 = corpus.dictionary.word2idx[a]
    idx2 = corpus.dictionary.word2idx[b]

    embed1 = model.embed.weight[idx1]
    embed2 = model.embed.weight[idx2]

    similarity = F.cosine_similarity(embed1.unsqueeze(0), embed2.unsqueeze(0))

    distance = 1 - similarity.item()

    return distance

word1 = 'army'
word2 = 'taxpayer'
distance = get_cosine_distance(word1, word2, model)
print(f"The cosine distance between '{word1}' and '{word2}' is {distance:.4f}")
```

```
The cosine distance between 'army' and 'taxpayer' is 1.1022
```

2.5 4 Q2.4 Teacher Forcing (Extra Credit 2 points)

What is teacher forcing? > Teacher forcing works by using the actual or expected output from the training dataset at the current time step $y(t)$ as input in the next time step $X(t+1)$, rather than the output generated by the network.

In the `Training` code this is achieved, implicitly, when we pass the entire input sequence (`inputs = ids[:, i:i+seq_length].to(device)`) to the model at once.

Copy below the `Training` code and modify it to disable teacher forcing training. Compare the performance of this model, to original model, what can you conclude? (compare perplexity and convergence rate)

[ ]:
```python
# Training code without Teacher Forcing

model = RNNLM(vocab_size, embed_size, hidden_size, num_layers).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    # Set initial hidden and cell states
    states = (torch.zeros(num_layers, batch_size, hidden_size).to(device),
              torch.zeros(num_layers, batch_size, hidden_size).to(device))

    for i in range(0, ids.size(1) - seq_length, seq_length):
        # Get mini-batch inputs and targets
        inputs = ids[:, i:i+seq_length].to(device)
        targets = ids[:, (i+1):(i+1)+seq_length].to(device)

        loss = 0

        # Forward pass
        for time_step in range(seq_length):
            states = detach(states)
            outputs, states = model(inputs, states)
            reshaped_outputs = outputs.view(batch_size, -1, vocab_size)[:, -1, :
↪]
            current_loss = criterion(reshaped_outputs, targets[:, time_step].
↪view(-1))
            loss += current_loss

            inputs = torch.argmax(reshaped_outputs, dim=-1).detach().
↪unsqueeze(1)

        loss /= seq_length

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        clip_grad_norm_(model.parameters(), 0.5)
        optimizer.step()

        step = (i+1) // seq_length
        if step % 100 == 0:
            print ('Epoch [{}/{}], Step[{}/{}], Loss: {:.4f}, Perplexity: {:5.
↪2f}'
                   .format(epoch+1, num_epochs, step, num_batches, loss.item(),␣
↪np.exp(loss.item())))
```

```
Epoch [1/5], Step[0/1549], Loss: 9.2097, Perplexity: 9993.40
Epoch [1/5], Step[100/1549], Loss: 6.5312, Perplexity: 686.22
Epoch [1/5], Step[200/1549], Loss: 6.6389, Perplexity: 764.24
Epoch [1/5], Step[300/1549], Loss: 6.6714, Perplexity: 789.52
Epoch [1/5], Step[400/1549], Loss: 6.5736, Perplexity: 715.95
Epoch [1/5], Step[500/1549], Loss: 6.5178, Perplexity: 677.10
Epoch [1/5], Step[600/1549], Loss: 6.4526, Perplexity: 634.37
Epoch [1/5], Step[700/1549], Loss: 6.6863, Perplexity: 801.36
Epoch [1/5], Step[800/1549], Loss: 6.4238, Perplexity: 616.33
Epoch [1/5], Step[900/1549], Loss: 6.5778, Perplexity: 718.94
Epoch [1/5], Step[1000/1549], Loss: 6.5673, Perplexity: 711.42
Epoch [1/5], Step[1100/1549], Loss: 6.7199, Perplexity: 828.71
Epoch [1/5], Step[1200/1549], Loss: 6.5152, Perplexity: 675.32
Epoch [1/5], Step[1300/1549], Loss: 6.7478, Perplexity: 852.22
Epoch [1/5], Step[1400/1549], Loss: 6.5587, Perplexity: 705.35
Epoch [1/5], Step[1500/1549], Loss: 6.6037, Perplexity: 737.79
Epoch [2/5], Step[0/1549], Loss: 8.1542, Perplexity: 3478.01
Epoch [2/5], Step[100/1549], Loss: 6.3141, Perplexity: 552.29
Epoch [2/5], Step[200/1549], Loss: 6.3861, Perplexity: 593.51
Epoch [2/5], Step[300/1549], Loss: 6.5283, Perplexity: 684.22
Epoch [2/5], Step[400/1549], Loss: 6.4733, Perplexity: 647.60
Epoch [2/5], Step[500/1549], Loss: 6.2380, Perplexity: 511.85
Epoch [2/5], Step[600/1549], Loss: 6.2412, Perplexity: 513.46
Epoch [2/5], Step[700/1549], Loss: 6.4999, Perplexity: 665.05
Epoch [2/5], Step[800/1549], Loss: 6.3046, Perplexity: 547.09
Epoch [2/5], Step[900/1549], Loss: 6.4421, Perplexity: 627.74
Epoch [2/5], Step[1000/1549], Loss: 6.4385, Perplexity: 625.46
Epoch [2/5], Step[1100/1549], Loss: 6.5548, Perplexity: 702.58
Epoch [2/5], Step[1200/1549], Loss: 6.4065, Perplexity: 605.79
Epoch [2/5], Step[1300/1549], Loss: 6.5075, Perplexity: 670.12
Epoch [2/5], Step[1400/1549], Loss: 6.4204, Perplexity: 614.22
Epoch [2/5], Step[1500/1549], Loss: 6.4570, Perplexity: 637.18
Epoch [3/5], Step[0/1549], Loss: 6.7246, Perplexity: 832.63
Epoch [3/5], Step[100/1549], Loss: 6.2055, Perplexity: 495.49
Epoch [3/5], Step[200/1549], Loss: 6.2376, Perplexity: 511.61
Epoch [3/5], Step[300/1549], Loss: 6.4465, Perplexity: 630.50
Epoch [3/5], Step[400/1549], Loss: 6.4238, Perplexity: 616.37
Epoch [3/5], Step[500/1549], Loss: 6.1189, Perplexity: 454.37
Epoch [3/5], Step[600/1549], Loss: 6.2208, Perplexity: 503.12
Epoch [3/5], Step[700/1549], Loss: 6.4263, Perplexity: 617.90
Epoch [3/5], Step[800/1549], Loss: 6.2033, Perplexity: 494.39
Epoch [3/5], Step[900/1549], Loss: 6.3113, Perplexity: 550.74
Epoch [3/5], Step[1000/1549], Loss: 6.3096, Perplexity: 549.85
Epoch [3/5], Step[1100/1549], Loss: 6.3997, Perplexity: 601.69
Epoch [3/5], Step[1200/1549], Loss: 6.2745, Perplexity: 530.87
Epoch [3/5], Step[1300/1549], Loss: 6.3819, Perplexity: 591.05
Epoch [3/5], Step[1400/1549], Loss: 6.3023, Perplexity: 545.81
Epoch [3/5], Step[1500/1549], Loss: 6.2675, Perplexity: 527.17
Epoch [4/5], Step[0/1549], Loss: 6.5475, Perplexity: 697.48
Epoch [4/5], Step[100/1549], Loss: 6.1811, Perplexity: 483.54
Epoch [4/5], Step[200/1549], Loss: 6.2515, Perplexity: 518.80
Epoch [4/5], Step[300/1549], Loss: 6.3563, Perplexity: 576.14
Epoch [4/5], Step[400/1549], Loss: 6.2971, Perplexity: 543.01
Epoch [4/5], Step[500/1549], Loss: 6.0442, Perplexity: 421.66
Epoch [4/5], Step[600/1549], Loss: 6.1266, Perplexity: 457.86
Epoch [4/5], Step[700/1549], Loss: 6.3525, Perplexity: 573.91
Epoch [4/5], Step[800/1549], Loss: 6.1367, Perplexity: 462.51
Epoch [4/5], Step[900/1549], Loss: 6.2070, Perplexity: 496.20
Epoch [4/5], Step[1000/1549], Loss: 6.2381, Perplexity: 511.89
Epoch [4/5], Step[1100/1549], Loss: 6.3913, Perplexity: 596.66
Epoch [4/5], Step[1200/1549], Loss: 6.2039, Perplexity: 494.66
Epoch [4/5], Step[1300/1549], Loss: 6.3092, Perplexity: 549.63
Epoch [4/5], Step[1400/1549], Loss: 6.1635, Perplexity: 475.06
Epoch [4/5], Step[1500/1549], Loss: 6.1648, Perplexity: 475.70
Epoch [5/5], Step[0/1549], Loss: 6.3928, Perplexity: 597.53
Epoch [5/5], Step[100/1549], Loss: 6.1331, Perplexity: 460.84
Epoch [5/5], Step[200/1549], Loss: 6.1405, Perplexity: 464.29
Epoch [5/5], Step[300/1549], Loss: 6.3862, Perplexity: 593.62
Epoch [5/5], Step[400/1549], Loss: 6.2252, Perplexity: 505.33
Epoch [5/5], Step[500/1549], Loss: 5.8803, Perplexity: 357.91
Epoch [5/5], Step[600/1549], Loss: 6.0875, Perplexity: 440.34
Epoch [5/5], Step[700/1549], Loss: 6.2520, Perplexity: 519.07
Epoch [5/5], Step[800/1549], Loss: 6.0414, Perplexity: 420.50
Epoch [5/5], Step[900/1549], Loss: 6.3427, Perplexity: 568.33
Epoch [5/5], Step[1000/1549], Loss: 6.1992, Perplexity: 492.36
Epoch [5/5], Step[1100/1549], Loss: 6.3112, Perplexity: 550.68
Epoch [5/5], Step[1200/1549], Loss: 6.1178, Perplexity: 453.89
Epoch [5/5], Step[1300/1549], Loss: 6.2769, Perplexity: 532.12
Epoch [5/5], Step[1400/1549], Loss: 6.1248, Perplexity: 457.07
Epoch [5/5], Step[1500/1549], Loss: 6.0773, Perplexity: 435.85
```

2.6 5 Q2.5 Distance Comparison (+1 point)

Repeat the work you did for `3 Q2.3 Embedding Distance` for the model in `4 Q2.4 Teacher Forcing` and compare the distances produced by these two models (i.e. with and without the teacher forcing), what can you conclude?

[ ]:
```
distance_without_teacher_forcing = get_cosine_distance(word1, word2, model)
print(f"The cosine distance between '{word1}' and '{word2}' is␣
↪{distance_without_teacher_forcing:.4f}")
```

```
The cosine distance between 'army' and 'taxpayer' is 0.9217
```