HW 10 - Seq2Seq MT with Attention-1

April 9, 2023

1 Yichen Huang

2 yh3550

[ ]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

device = torch.device("mps" if torch.cuda.is_available() else "cpu")
```

[ ]:
```
```

Below are a few shorter problems. Please fill in the cells with your answer.

[ ]:
```
FILL_IN = "FILL_IN"
```

2.0.1 Problem 1

Given the example below, we have a bidirectional GRU. What is the connection between output and hidden? Explain in detail where exactly in output you can find hidden, and why.

[ ]:
```
gru = nn.GRU(1, 1, bidirectional=True, batch_first=True)

# Use this same data for Problems 1 - 4
x = torch.rand(4, 5, 1)

# What is true about hidden and output? Where in output are the values in
 ↪hidden? Be careful!
output, hidden = gru(x)
```

[ ]:
```
print("output shape: ", output.shape)
print("hidden shape: ", hidden.shape)
```

```
output shape:  torch.Size([4, 5, 2])
hidden shape:  torch.Size([2, 4, 1])
```

[ ]:
```
# The output of last sequence in forward of the GRU is the same as the hidden␣
↪state in forward direction
print(torch.equal(output[:, -1, 0], hidden[0, :, :].squeeze()))
# The output of first sequence in backward of the GRU is the same as the hidden␣
↪state in backward direction
print(torch.equal(output[:, 0, -1], hidden[-1, :, :].squeeze()))
```

```
True
True
```

2.0.2 Answer

The input size and hidden size are both 1. The input is 4X5X1, which is the batch size, sequence length, and input size. The output is 4X5X2, which is the batch size, sequence length, and hidden size times 2 because of bidirectional. The hidden is 2X4X1, which is the number of layers times the number of directions, the batch size, and the hidden size. From the structure of GRU, The output of last sequence in forward of the GRU is the same as the hidden state in forward direction; and the output of first sequence in backward of the GRU is the same as the hidden state in backward direction.

[ ]:
```

```

2.0.3 Problem 2

Consider the case when you have num_layers = 2 in a GRU as below. Describe what the conection now is between the hidden layer and the output layer. Specifically, what part of the

[ ]:
```
gru = nn.GRU(1, 1, num_layers=2, batch_first=True)

# What is true about hidden and output? Where in output are the values in␣
↪hidden? Be careful!
output, hidden = gru(x)
```

[ ]:
```
print("output shape: ", output.shape)
print("hidden shape: ", hidden.shape)
```

```
output shape:  torch.Size([4, 5, 1])
hidden shape:  torch.Size([2, 4, 1])
```

[ ]:
```
print(torch.equal(output[:, -1, 0], hidden[-1, :, :].squeeze()))
```

```
True
```

2.0.4 Answer

The input size and hidden size are both 1. The input is 4X5X1, which is the batch size, sequence length, and input size. The output is 4X5X2, which is the batch size, sequence length, and hidden size times 1. The hidden is 2X4X1, which is the number of layers times the number of directions, the batch size, and the hidden size. The output is equal to the hidden state of the last layer.

[ ]:

```

```

2.0.5 Problem 3

Given Problem 2, write code to get the representation across all time steps T of the first layer. I.e., write code below to get $(\vec{h}_1^1, \ldots, \vec{h}_T^1)$. Do this for a GRU with two layers. Note that "output" does not have what you want - you need to be a little clever to get this.

Hint: See the bottom of this notebook if you are totally stuck.

[ ]:

```
torch.manual_seed(42)
gru = nn.GRU(1, 1, num_layers=2, batch_first=True)

# What is true about hidden and output? Where in output are the values in␣
↪hidden? Be careful!
output, hidden = gru(x)

torch.manual_seed(42)
gru1 = nn.GRU(1, 1, num_layers=1, batch_first=True)
gru2 = nn.GRU(1, 1, num_layers=1, batch_first=True)
output1, hidden1 = gru1(x)
output2, hidden2 = gru2(output1)
```

[ ]:

```
# If you follow the hint, you need 2 GRU models each of 1 layer, gru1 and gru2
# hidden1, output1 is the output of gru1 if you push x through
# hidden 2, output2 is the output and hidden state of gru2 if you push output1␣
↪through
# These asserts below should pass
# You need to transfer the gru model's appropriate parameters to the right␣
↪model, gru1 or gru2, then manuall pass data through
assert(torch.all(torch.eq(output, output2)))
assert(torch.all(torch.eq(hidden, torch.vstack((hidden1, hidden2)))))
```

2.0.6 Problem 4

In this problem we want to deal with sequences that are not the same length. Suppose we have 3 sequences of data a, b, c, where the length of a, b and c are $2, 3$ and 4 respectively. Assume you want to do a batch operation where the batch consists of a, b and c and you want to run these through the model. At the end, you'd like to get the final hidden state for each sentence. One way to do this is to pad all the sequences so they are length 4 and feed the 3 by 4 vector into the GRU. - What is the problem with doing this? What is ineffcient about it? What is inefficint about output_padded and how it was computed?

Investigate how to do this better using the 4 imports below. You may not need all of these functions. I.e. create a batch of size 3 containing the 3 tensors.

- What is output_padded1 vs output_padded? Compare the shape and the values inside. What is better about the way output_padded2 was computed?

[ ]:
```
from torch.nn.utils.rnn import pack_sequence, pad_sequence,␣
 ↪pack_padded_sequence, pad_packed_sequence

# Each tensor is in (length, values) format
a = torch.randn(2, 1)
b = torch.randn(3, 1)
c = torch.randn(4, 1)

la, lb, lc = 2, 3, 4

rnn = nn.GRU(1, 1, num_layers=1, batch_first=True)
```

[ ]:
```
# Answer:
# One easy way to do this is to do this is manually. Just have two GRUs and hae␣
 ↪one's output feed into the other
# Then, low through the named parameters of the gru and insert them into one or␣
 ↪the other of the two grus above

seq = [a, b, c]

# Use pad_sequence to pass the create a batch of size 3 and pad it so each␣
 ↪sequence has length 4
# Use batch_first=True
padded = pad_sequence(seq, batch_first=True)

output_padded, hidden_padded = gru(padded)

print(padded.shape)

# Use pack_padded_sequence to pack a, b and c
# Use batch_first=True
packed1 = pack_sequence([a,b,c], enforce_sorted=False)
# pack_padded_sequence is older, the below is a newer command
packed2 = pack_padded_sequence(padded, [la, lb, lc], batch_first=True,␣
 ↪enforce_sorted=False)

output_packed1, hidden_packed1 = gru(packed1)
output_packed2, hidden_packed2 = gru(packed2)

# Use pad_packed_sequence to unpack the results above; you now get padded␣
 ↪results, similar to the ouput_padded and hidden_padded above
```

```
# What is different and the same about output_padded1 and output_padded?
# Why is it more efficient to use this method as opposed to just pad all␣
↪elements in a batch and pass them through?
output_padded1, output_lengths1 = pad_packed_sequence(output_packed1,␣
↪batch_first=True)
output_padded2, output_lengths12= pad_packed_sequence(output_packed2,␣
↪batch_first=True)

print(output_padded2.shape)

assert(torch.all(torch.eq(output_padded1, output_padded2)))
```

```
torch.Size([3, 4, 1])
torch.Size([3, 4, 1])
```

[ ]:

```

```

[ ]:

```

```

[ ]:

```

```

This example is like the previous one in HW 9, but now we want a more complicated model with attention.

[ ]:

```

```

[ ]:

```
SOS_token = 0
EOS_token = 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
```

```
        else:
            self.word2count[word] += 1
```

[ ]:
```
# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

[ ]:
```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

[ ]:
```
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
```

```
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)


# Only use pairs where the english data (pair[1]) has the prefix above
# Also, only consider data where pair[0] and pair[1] have length less than␣
 ↪MAX_LENGTH
# "length" here means the number of tokens, you need to split pair[0] and␣
 ↪pair[1] on ' ' then get the length
def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) <␣
 ↪MAX_LENGTH and p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

[ ]:
```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs


input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
```

```
Reading lines…
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words…
Counted words:
fra 4345
eng 2803
['mes pensees ne sont pas claires .', 'i m not thinking clearly .']
```

[ ]:
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        # Make the encoder a GRU and also make it bidirectional.
        # Let it have 1 layers in the vertical direction.
        self.gru = nn.GRU(hidden_size, hidden_size, num_layers=1,␣
↪bidirectional=True, batch_first=True)

    def forward(self, input, hidden):
        # Get the embeddings and reshape to be (1, 1, -1)
        # Why? remember we use batch size = 1 in this HW for simplicity
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(2, 1, self.hidden_size, device=device)
```

[ ]:
```
class AttentionDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1,␣
↪max_length=MAX_LENGTH):
        super(AttentionDecoderRNN, self).__init__()

        # H
        self.hidden_size = hidden_size

        # vocab_size
        self.output_size = output_size

        self.dropout_p = dropout_p
        self.max_length = max_length

        # Intialize the embedding going from vocab_size to H
        self.embedding = nn.Embedding(self.output_size, self.hidden_size)

        # Initialize the attention projection as a Linear layer from 2*H to␣
↪self.max_length
        self.attention_projection = nn.Linear(2*self.hidden_size, self.
↪max_length)

        # Initialize the output projection as a Linear layer from 2*H -> H␣
↪(this is before we project to the vocab_size)
        self.output_projection = nn.Linear(2*self.hidden_size, self.hidden_size)
```

```
        # Intialize a Dropout layer with self.dropout_p probability
        self.dropout = nn.Dropout(self.dropout_p)

        # Make the GRU be unidirectional and also with 1 hidden layer
        # Input and hidden data each have a dimension of H
        self.gru = nn.GRU(self.hidden_size, self.hidden_size, num_layers=1,␣
↪bidirectional=False, batch_first=True)

        # Intialize a Linear layer going from H to vocab_size
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        # (1, 1, H)
        embedded = self.embedding(input).view(1, 1, -1)

        # Pass embedding through the dropout layer
        embedded = self.dropout(embedded)

        # (1, 2*H)
        # Concatenate yt and kt to get a vector (y_t, k_{t-1})
        embedded_hidden = torch.cat((embedded[0], hidden[0]), 1)

        # (1, MAX_LENGTH)
        # Project the above vector to get a vector mixing the elements of the␣
↪above
        # This vector will be used to get attention scores with all the encoder␣
↪embeddings
        # Here, the scores are scores = W_a[y_{t}, k_{t-1}] + b_a where W_a an␣
↪b_a are in self.attention_projection
        # You can have other formats here, but the one above is enough for this␣
↪problem
        attention_scores = self.attention_projection(embedded_hidden)

        # (1, MAX_LENGTH)
        # Get the attention weights from the scores
        # I.e. get probabilistic from the above scores
        attention_weights = F.softmax(attention_scores, dim=1)

        # (1, 1, H)
        # Multiply the weights by the hidden states (h_1, h_2, .., h_{T_x}) of␣
↪the encoder
        # This should be a vector of the above dimensions, so you'll need␣
↪unsqueeze
        # One way to do this is using torch.bmm on these unsqueezed vectors
```

```
        # This will be the at vector that mixed the encoder's hidden␣
↪representations; "c_{t}"" in lecture
        attention_context = torch.bmm(attention_weights.unsqueeze(0),␣
↪encoder_outputs.unsqueeze(0))

        # (1, 2*H)
        # Concatenate (yt, at) to get a vector that we will use to predict the␣
↪output
        output = torch.cat((embedded[0], attention_context[0]), 1)

        # (1, 1, H)
        # Project the above vector into a new vector we'll use to predict with
        # unsqueeze(0) the result to get the right dimensions
        output = self.output_projection(output).unsqueeze(0)

        # (1, H)
        # Pass through ReLU
        output = F.relu(output)

        # (1, H) and (1, H)
        # Pass the output and hidden through the GRU. Note that we apply␣
↪attention before we pass into the GRU
        # The input ("output" vector) has attentional information in it
        output, hidden = self.gru(output, hidden)

        # (1, vocab_size)
        # Either apply log_softmax to output or leave it alone
        # This will have you use the NLLoss or the CrossEntropyLoss
        output = self.out(output)
        return output, hidden, attention_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

[ ]:
```
# Split a sentence by ' ' and return a list of the tokens (int ids) for each␣
↪word
# Use word2index
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

# Call the above on a sentence
# After calling, add the EOS_token (int id) to the gotten list
# Return a tensor, but reshape it so it's dimensions (-1, 1)
def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, device=device).view(-1, 1)
```

```
# For a source, target pair, call the above. Return a tuple of 2 tensors, one␣
↪input_tensor and another an output_tensor
def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

[ ]:
```
teacher_forcing_ratio = 0.5

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,␣
↪decoder_optimizer, criterion, max_length=MAX_LENGTH):
    # Initialize the hidden states
    encoder_hidden = encoder.initHidden()

    # Reset the optimizer gradients to 0
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize the encoder outputs - these are used to store the vector's␣
↪we'll use to get attention scores
    # This should be (max_length, H) and all zeros to start
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
↪device=device)

    loss = 0

    # Pass the data through the encoder
    for ei in range(input_length):

        encoder_output, encoder_hidden = encoder(input_tensor[ei],␣
↪encoder_hidden)
        # Save the encoder output into "encoder_outputs"

        encoder_outputs[ei] = encoder_output[0,0,:256]

    # Initialize the decoder input to the SOS_token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Initialize the hidden states of the decoder with the hidden states of the␣
↪encoder
    decoder_hidden = encoder_hidden[0].unsqueeze(0)
```

```
    # For this pair, use teacher forcing with 50% probability, else don't
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else␣
↪False

    target_length_used = 0

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input

        target_length_used = target_length

        for di in range(target_length):
            # Push decoder_input, decoder_hidden, and decoder_cell through the␣
↪decoder
            decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output.squeeze(0), target_tensor[di].
↪view(-1))
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            # Push decoder_input, decoder_hidden, and decoder_cell through the␣
↪decoder
            decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
            # Get greedy top probability prediction
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.detach().to(device)  # detach from history as␣
↪input

            # Get the loss
            loss += criterion(decoder_output.squeeze(0), target_tensor[di].
↪view(-1))

            # Update the target_length_used
            target_length_used += 1

            # If the EOS_token was generated, exit
            if topi.item() == EOS_token:
                break

    # Collect gradients
    loss.backward()
```

```
    # Do a step; do this both for the encoder and the decoder
    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length_used
```

[ ]:
```
import time
import math


def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)


def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))

import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # This locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

[ ]:
```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100,␣
 ↪learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    # Initialize the encoder and decoder optimizers with the above learning rate
    encoder_optimizer = torch.optim.SGD(encoder.parameters(), lr=learning_rate)
```

```
    decoder_optimizer = torch.optim.SGD(decoder.parameters(), lr=learning_rate)

    # Get n_iters training pairs
    # In this example, we are effectively doing SGD with batch size 1
    training_pairs = random.choices(pairs, k=n_iters)

    # The loss; either NLLLoss if you use log sigmoids or CrossEntropyLoss if␣
↪you use logits
    criterion = nn.CrossEntropyLoss()

    for it in range(1, n_iters + 1):
        training_pair = tensorsFromPair(training_pairs[it - 1])
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        # Train on the input, target pair
        loss = train(
            input_tensor=input_tensor,
            target_tensor=target_tensor,
            encoder=encoder,
            decoder=decoder,
            encoder_optimizer=encoder_optimizer,
            decoder_optimizer=decoder_optimizer,
            criterion=criterion
        )

        # Update the total loss and the plot loss
        # We can plot and print at different granularities
        print_loss_total += loss
        plot_loss_total += loss

        if it % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, it / n_iters),
                                         it, it / n_iters * 100,␣
↪print_loss_avg))

        if it % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

            showPlot(plot_losses)
```

[ ]:
```
hidden_size = 256
encoder = EncoderRNN(input_lang.n_words, hidden_size).to(device)
```

```
decoder = AttentionDecoderRNN(hidden_size, output_lang.n_words).to(device)

trainIters(encoder, decoder, 75000, print_every=5000)
```

```
/var/folders/f8/mb2zprsj5wj1n9ygh0fcr3nw0000gn/T/ipykernel_36119/1954892691.py:2
5: RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained until
explicitly closed and may consume too much memory. (To control this warning, see
the rcParam `figure.max_open_warning`). Consider using
`matplotlib.pyplot.close()`.
  plt.figure()
```

```
2m 1s (- 28m 25s) (5000 6%) 3.0714
4m 3s (- 26m 23s) (10000 13%) 2.3926
6m 5s (- 24m 20s) (15000 20%) 2.0595
8m 7s (- 22m 21s) (20000 26%) 1.8053
10m 10s (- 20m 21s) (25000 33%) 1.5721
12m 14s (- 18m 21s) (30000 40%) 1.3952
14m 18s (- 16m 20s) (35000 46%) 1.2557
16m 22s (- 14m 19s) (40000 53%) 1.1192
18m 24s (- 12m 16s) (45000 60%) 0.9862
20m 26s (- 10m 13s) (50000 66%) 0.8900
22m 28s (- 8m 10s) (55000 73%) 0.7996
24m 29s (- 6m 7s) (60000 80%) 0.7419
26m 30s (- 4m 4s) (65000 86%) 0.7199
28m 31s (- 2m 2s) (70000 93%) 0.6420
30m 33s (- 0m 0s) (75000 100%) 0.5532
```

[ ]:

[ ]:

[ ]:

[ ]:
```
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        # Transform the input sentence into a tensor
        input_tensor = tensorFromSentence(input_lang, sentence)

        input_length = input_tensor.size()[0]

        # Initilize the hidden and cell states of the LSTM
        encoder_hidden = encoder.initHidden()

        # Initialize the encoder outputs as in train
        encoder_outputs = torch.zeros(max_length, encoder.hidden_size,
 ↪device=device)
```

```
        # Run the data through the LSTM word by word manually
        # At each step, feed in the input, the hidden state, and the cell state␣
↪and calture the new hidden / cell states
        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],␣
↪encoder_hidden)
            encoder_outputs[ei] = encoder_output[0, 0, :256]

        # Initialize the decoder input with a SOS_token
        decoder_input = torch.tensor([[SOS_token]], device=device)
        # SOS

        # Initialize the decoder hidden state with the encoder's hidden state
        decoder_hidden = encoder_hidden[0].unsqueeze(0)

        # Initialize the decoded words and a matrix of T by T length which will␣
↪store the attention weights
        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            # Pass the data through the decoder
            decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
            # Save the attention matrix above - you might want to look at this␣
↪later to debug
            decoder_attentions[di] = decoder_attention.data
            # Get the top (1) decoder output as use this as the next input
            topv, topi = 0, torch.argmax(decoder_output, dim=-1)

            # Add the word for the topi token to the decoded_words
            decoded_words.append(output_lang.index2word[topi.item()])

            # If EOS was decoded, break
            if topi.item() == EOS_token:
                break

            # Save the token above as the next input
            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

[ ]:
```

```

[ ]:
```python
from nltk.translate.bleu_score import sentence_bleu

def evaluateRandomly(encoder, decoder, n=7500, debug=False):
    bleu_scores = []
    for i in range(n):
        pair = random.choice(pairs)
        if debug:
            print('French Original: ', pair[0])
            print('English Reference: ', pair[1])
        # Leave out the EOS symbol
        output_words, _ = evaluate(encoder, decoder, pair[0])

        # If EOS is at the end, remove it from output_words
        if output_words[-1] == 'EOS':
            output_words = output_words[:-1]

        output_sentence = ' '.join(output_words)
        # Use pair[1] as te refernce and get the BLEU score based on just 2␣
↪grams with 50% weight each
        score = sentence_bleu([pair[1]], output_sentence, weights=(0.5, 0.5))

        # Append the BLEU score to the list of BLEU scores
        bleu_scores.append(score)
        if debug:
            print('Candidate Translation: ', output_sentence)
            print('BLEU: ', score)
            print('')
    print('The mean BLEU score is: ', np.mean(bleu_scores))
```

[ ]:
```python
evaluateRandomly(encoder, decoder)
```

```
The mean BLEU score is:  0.8543203314626014
```

[ ]:
```python
# You should get something > 60 % here
evaluateRandomly(encoder, decoder, debug=True,n=8)
```

```
French Original:  je ne demens pas cela .
English Reference:  i m not denying that .
Candidate Translation:  i m not that .
BLEU:  0.5647181220077593

French Original:  nous sommes bourres .
English Reference:  we re smashed .
Candidate Translation:  we re sloshed .
BLEU:  0.8251983888449983

French Original:  je suis curieuse .
```

```
English Reference:  i m curious .
Candidate Translation:  i m curious .
BLEU:  1.0

French Original:  tu n es pas fatiguee si ?
English Reference:  you re not tired are you ?
Candidate Translation:  you re not tired are you ?
BLEU:  1.0

French Original:  vous etes juste .
English Reference:  you re fair .
Candidate Translation:  you re fair .
BLEU:  1.0

French Original:  je ne suis pas toujours libre le dimanche .
English Reference:  i am not always free on sundays .
Candidate Translation:  i m not always free on sundays .
BLEU:  0.9534722941050486

French Original:  je suis tien et tu es mien .
English Reference:  i am yours and you are mine .
Candidate Translation:  i am yours and you are mine .
BLEU:  1.0

French Original:  je suis a la maison .
English Reference:  i m at home .
Candidate Translation:  i am at home .
BLEU:  0.8864052604279183

The mean BLEU score is:  0.9037242581732156
```

[ ]:

```

```

[ ]:

```

```

Hint for Problem 3: create two layer=1 GRU models and transfer the 2 layer's model's parameters to the appropriate GRU. Then, manually push data through.

[ ]:

```
# If you follow the hint, you need 2 GRU models each of 1 layer, gru1 and gru2
# hidden1, output1 is the output of gru1 if you push x through
# hidden 2, output2 is the output and hidden state of gru2 if you push output1␣
↪through
# These asserts below should pass
# You need to transfer the gru model's appropriate parameters to the right␣
↪model, gru1 or gru2, then manuall pass data through
assert(torch.all(torch.eq(output, output2)))
assert(torch.all(torch.eq(hidden, torch.vstack((hidden1, hidden2)))))
```