Machine Translation with Seq2Seq and Transformers

In this exercise you will implement a Sequence to Sequence(Seq2Seq) and a Transformer model and use them to perform machine translation.

A quick note: if you receive the following TypeError "super(type, obj): obj must be an instance or subtype of type", try re-importing that part or restarting your kernel and re-running all cells. Once you have finished making changes to the model constuctor, you can avoid this issue by commenting out all of the model instantiations after the first (e.g. lines starting with "model = TransformerTranslator(*args, **kwargs)").

In [4]:

```
import numpy as np
import csv
import torch

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1: Introduction

Multi30K: Multilingual English-German Image Descriptions

Multi30K is a dataset for machine translation tasks. It is a multilingual corpus containing English sentences and their German translation. In total it contains 31014 sentences(29000 for training, 1014 for validation, and 1000 for testing). As one example:

En: `Two young, White males are outside near many bushes.`

De: `Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.`

You can read more info about the dataset here. The following parts of this assignment will be based on this dataset.

TorchText: A PyTorch Toolkit for Text Dataset and NLP Tasks

TorchText is a PyTorch package that consists of data processing utilities and popular datasets for natural language. The key idea of TorchText is that datasets can be organized in *Field*, *TralsationDataset*, and *BucketIterator* classes. They serve to help with data splitting and loading, token encoding, sequence padding, etc. You don't need to know about how TorchText works in detail, but you might want to know about why those classes are needed and what operations are necessary for machine translation. These knowledge can be migrated to all sequential data modeling. In the following parts, we will provide you with some code to help you understand.

You can refer to torchtext's documentation(v0.6.0) here.

Spacy

Spacy is package designed for tokenization in many languages. Tokenization is a process of splitting raw text data into lists of tokens that can be further processed. Since TorchText only provides tokenizer for English, we will be using Spacy for our assignment.

Notice: For the following assignment, we strongly recommend you to work in a virtual python environment. We recommend Anaconda, a powerful environment control tool. You can download it here.

1.1: Prerequisites

Before you start this assignment, you need to have all required packages installed either on the terminal you are using, or in the virtual environment. Please make sure you have the following package installed:

`PyTorch, TorchText, Spacy, Tqdm, Numpy`

You can first check using either `pip freeze` in terminal or `conda list` in conda environment. Then run the following code block to make sure they can be imported.

In [5]:

```
# Just run this block. Please do not modify the following code.
import math
import time

# Pytorch package
import torch
import torch.nn as nn
import torch.optim as optim

# Torchtest package
```

```
from torchtext.legacy.datasets  import Multi30k
from torchtext.legacy.data import Field, BucketIterator

# Tqdm progress bar
from tqdm import tqdm_notebook, tqdm

# Code provide to you for training and evaluation
from utils import train, evaluate, set_seed_nb, unit_test_values
```

Once you properly import the above packages, you can proceed to download Spacy English and German tokenizers by running the following command in your **terminal**. They will take some time.

```
python -m spacy download en
```

```
python -m spacy download de
```

Now lets check your GPU availability and load some sanity checkers. By default you should be using your gpu for this assignment if you have one available.

In [6]:
```
# Check device availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("You are using device: %s" % device)
```

```
You are using device: cuda
```

In [7]:
```
# load checkers
d1 = torch.load('./data/d1.pt')
d2 = torch.load('./data/d2.pt')
d3 = torch.load('./data/d3.pt')
d4 = torch.load('./data/d4.pt')
```

1.2: Preprocess Data

With TorchText and Spacy tokenizers ready, you can now prepare the data using *TorchText* objects. Just run the following code blocks. Read the comment and try to understand what they are for.

In [9]:
```
# You don't need to modify any code in this block

# Define the maximum length of the sentence. Shorter sentences will be padded to that length and longer sentenc
```

```
MAX_LEN = 20

# Define the source and target language
SRC = Field(tokenize = "spacy",
            tokenizer_language="de_core_news_sm",
            init_token = '<sos>',
            eos_token = '<eos>',
            fix_length = MAX_LEN,
            lower = True)

TRG = Field(tokenize = "spacy",
            tokenizer_language="en_core_web_sm",
            init_token = '<sos>',
            eos_token = '<eos>',
            fix_length = MAX_LEN,
            lower = True)

# Download and split the data. It should take some time
train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
                                                    fields = (SRC, TRG))
```

In [10]:

```
# Define Batchsize
BATCH_SIZE = 128

# Build the vocabulary associated with each language
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)

# Get the padding index to be ignored later in loss calculation
PAD_IDX = TRG.vocab.stoi['<pad>']

# Get data-loaders using BucketIterator
train_loader, valid_loader, test_loader = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE, device = device)

# Get the input and the output sizes for model
input_size = len(SRC.vocab)
output_size = len(TRG.vocab)
```

2: Implement Vanilla RNN and LSTM

In this section, you will need to implement a Vanilla RNN and an LSTM unit using PyTorch Linear layers and nn.Parameter. This is designed to help you to understand how they work behind the scene. The code you will be working with is in *LSTM.py* and *RNN.py* under *naive* folder. Please refer to instructions among this notebook and those files.

2.1: Implement an RNN Unit

In this section you will be using PyTorch Linear layers and activations to implement a vanilla RNN unit. Please refer to the following structure and complete the code in RNN.py:

Please implement the following update rules:

Run the following block to check your implementation

In [11]:

```
from models.naive.RNN import VanillaRNN

set_seed_nb()
x1,x2 = (1,4), (-1,2)
h1,h2 = (-1,2,0,4), (0,1,3,-1)
batch = 5
x = torch.FloatTensor(np.linspace(x1,x2,batch))
h = torch.FloatTensor(np.linspace(h1,h2,batch))

rnn = VanillaRNN(x.shape[-1], h.shape[-1], 3)
out, hidden = rnn.forward(x,h)

expected_out, expected_hidden = unit_test_values('rnn')

if out is not None:
    print('Close to out: ', expected_out.allclose(out, atol=1e-4))
    print('Close to hidden: ', expected_hidden.allclose(hidden, atol=1e-4))
else:
    print("NOT IMPLEMENTED")
```

```
Close to out:  True
Close to hidden:  True
```

2.2: Implement an LSTM Unit

In this section you will be using PyTorch nn.Parameter and activations to implement an LSTM unit. You can simply translate the following equations using nn.Parameter and PyTorch activation functions to build an LSTM from scratch:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Here's a great visualization of the above equation from Colah's blog to help you understand LSTM unit. You can also read more about it from that blog.

If you want to see nn.Parameter in example, check out this tutorial from PyTorch. Run the following block to check your implementation.

Note that in this case we are implementing a full loop with LSTM, iterating over each time step. The test cases reflect this as there are multiple sequences.

In [12]:

```
from models.naive.LSTM import LSTM

set_seed_nb()
x1,x2 = np.mgrid[-1:3:3j, -1:4:2j]
h1,h2 = np.mgrid[-2:2:3j, 1:3:4j]
batch = 4
x = torch.FloatTensor(np.linspace(x1,x2,batch))
h = torch.FloatTensor(np.linspace(h1,h2,batch))
lstm = LSTM(x.shape[-1], h.shape[-1])
h_t, c_t = lstm.forward(x)

expected_ht, expected_ct = unit_test_values('lstm')

if h_t is not None:
    print('Close to h_t: ', expected_ht.allclose(h_t, atol=1e-4))
    print('Close to c_t; ', expected_ct.allclose(c_t, atol=1e-4))
else:
    print("NOT IMPLEMENTED")
```

```
Close to h_t:  True
Close to c_t;  True
```

3: Train a Seq2Seq Model

In this section, you will be working on implementing a simple Seq2Seq model. You will first implement an Encoder and a Decoder, and then join them together with a Seq2Seq architecture. You will need to complete the code in *Decoder.py*, *Encoder.py*, and *Seq2Seq.py* under *seq2seq* folder. Please refer to the instructions in those files.

3.1: Implement the Encoder

In this section you will be implementing an RNN/LSTM based encoder to model Enligsh texts. Please refer to the instructions in *seq2seq/Encoder.py*. Run the following block to check your implementation.

In [13]:

```
from models.seq2seq.Encoder import Encoder

set_seed_nb()
i, n, h = 10, 4, 2

encoder = Encoder(i, n, h, h)
x_array = np.random.rand(5,1) * 10
```

```
x = torch.LongTensor(x_array)
out, hidden = encoder.forward(x)
expected_out, expected_hidden = unit_test_values('encoder')

print('Close to out: ', expected_out.allclose(out, atol=1e-4))
print('Close to hidden: ', expected_hidden.allclose(hidden, atol=1e-4))
```

```
Close to out:  True
Close to hidden:  True
```

3.2: Implement the Decoder

In this section you will be implementing an RNN/LSTM based decoder to model German texts. Please refer to the instructions in *seq2seq/Decoder.py*. Run the following block to check your implementation.

In [14]:

```
from models.seq2seq.Decoder import Decoder

set_seed_nb()
i, n, h =  10, 2, 2
decoder = Decoder(h, n, n, i)
x_array = np.random.rand(5, 1) * 10
x = torch.LongTensor(x_array)
_, enc_hidden = unit_test_values('encoder')
out, hidden = decoder.forward(x,enc_hidden)

expected_out, expected_hidden = unit_test_values('decoder')
# print(expected_hidden.shape)
# print(out.shape)


# print(out)
# print(expected_out)

print('Close to out: ', expected_out.allclose(out, atol=1e-4))
print('Close to hidden: ', expected_hidden.allclose(hidden, atol=1e-4))
```

```
Close to out:  True
Close to hidden:  True
```

3.3: Implement the Seq2Seq

In this section you will be implementing the Seq2Seq model that utilizes the Encoder and Decoder you implemented. Please refer to the instructions in *seq2seq/Seq2Seq.py*. Run the following block to check your implementation.

In [15]:
```
from models.seq2seq.Seq2Seq import Seq2Seq

set_seed_nb()
embedding_size = 32
hidden_size = 32
input_size = 8
output_size = 8
batch, seq = 1, 2

encoder = Encoder(input_size, embedding_size, hidden_size, hidden_size)
decoder = Decoder(embedding_size, hidden_size, hidden_size, output_size)

seq2seq = Seq2Seq(encoder, decoder, 'cpu')
x_array = np.random.rand(batch, seq) * 10
x = torch.LongTensor(x_array)
out = seq2seq.forward(x)

expected_out = unit_test_values('seq2seq')

# print(expected_out)
# print(out)

print('Close to out: ', expected_out.allclose(out, atol=1e-4))
```

```
Close to out:  True
```

3.4: Train your Seq2Seq model

Now its time to combine what we have and train a Seq2Seq translator. We provided you with some training code and you can simply run them to see how your translator works. If you implemented everything correctly, you should see some meaningful translation in the output. You can modify the hyperparameters to improve the results. You can also tune the BATCH_SIZE in section 1.2.

In [16]:
```
# Hyperparameters. You are welcome to modify these
encoder_emb_size = 32
encoder_hidden_size = 64
encoder_dropout = 0.2

decoder_emb_size = 32
```

```
decoder_hidden_size = 64
decoder_dropout = 0.2

learning_rate = 1e-3
model_type = "LSTM"

EPOCHS = 10

#input size and output size
input_size = len(SRC.vocab)
output_size = len(TRG.vocab)
```

In [17]:

```
# Declare models, optimizer, and loss function
encoder = Encoder(input_size, encoder_emb_size, encoder_hidden_size, decoder_hidden_size, dropout = encoder_drc
decoder = Decoder(decoder_emb_size, encoder_hidden_size, encoder_hidden_size, output_size, dropout = decoder_dr
seq2seq_model = Seq2Seq(encoder, decoder, device)

optimizer = optim.Adam(seq2seq_model.parameters(), lr = learning_rate)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
```

In [18]:

```
# If training freezes it is due to memory error
for epoch_idx in range(EPOCHS):
    print("-----------------------------------")
    print("Epoch %d" % (epoch_idx+1))
    print("-----------------------------------")

    train_loss, avg_train_loss = train(seq2seq_model, train_loader, optimizer, criterion)
    scheduler.step(train_loss)

    val_loss, avg_val_loss = evaluate(seq2seq_model, valid_loader, criterion)

    avg_train_loss = avg_train_loss.item()
    avg_val_loss = avg_val_loss.item()
    print("Training Loss: %.4f. Validation Loss: %.4f. " % (avg_train_loss, avg_val_loss))
    print("Training Perplexity: %.4f. Validation Perplexity: %.4f. " % (np.exp(avg_train_loss), np.exp(avg_val_
```

```
-----------------------------------
Epoch 1
-----------------------------------

Training Loss: 5.9040. Validation Loss: 5.1771.
```

```
Training Perplexity: 366.4853. Validation Perplexity: 177.1735.
---------------------------------
Epoch 2
---------------------------------

Training Loss: 5.2063. Validation Loss: 5.0790.
Training Perplexity: 182.4198. Validation Perplexity: 160.6160.
---------------------------------
Epoch 3
---------------------------------

Training Loss: 5.1436. Validation Loss: 5.0404.
Training Perplexity: 171.3303. Validation Perplexity: 154.5384.
---------------------------------
Epoch 4
---------------------------------

Training Loss: 5.1025. Validation Loss: 4.9827.
Training Perplexity: 164.4311. Validation Perplexity: 145.8716.
---------------------------------
Epoch 5
---------------------------------

Training Loss: 5.0488. Validation Loss: 4.9261.
Training Perplexity: 155.8345. Validation Perplexity: 137.8421.
---------------------------------
Epoch 6
---------------------------------

Training Loss: 4.9910. Validation Loss: 4.8602.
Training Perplexity: 147.0861. Validation Perplexity: 129.0513.
---------------------------------
Epoch 7
---------------------------------

Training Loss: 4.9290. Validation Loss: 4.7981.
Training Perplexity: 138.2387. Validation Perplexity: 121.2836.
---------------------------------
Epoch 8
---------------------------------

Training Loss: 4.8754. Validation Loss: 4.7466.
Training Perplexity: 131.0233. Validation Perplexity: 115.1949.
---------------------------------
```

```
Epoch 9
---------------------------------


Training Loss: 4.8220. Validation Loss: 4.6911.
Training Perplexity: 124.2152. Validation Perplexity: 108.9680.
---------------------------------
Epoch 10
---------------------------------


Training Loss: 4.7745. Validation Loss: 4.6387.
Training Perplexity: 118.4486. Validation Perplexity: 103.4128.
```

4: Train a Transformer

We will be implementing a one-layer Transformer **encoder** which, similar to an RNN, can encode a sequence of inputs and produce a final output of possibility of tokens in target language. This is the architecture:

You can refer to the original paper for more details. Before implementing a translator, we'll make a transformer binary classifier on the CoLA dataset below.

The Corpus of Linguistic Acceptability (CoLA)

The Corpus of Linguistic Acceptability (CoLA) in its full form consists of 10657 sentences from 23 linguistics publications, expertly annotated for acceptability (grammaticality) by their original authors. Native English speakers consistently report a sharp contrast in acceptability between pairs of sentences. Some examples include:

`What did Betsy paint a picture of?` (Correct)

`What was a picture of painted by Betsy?` (Incorrect)

You can read more info about the dataset here. This is a binary classification task (predict 1 for correct grammar and 0 otherwise).

We will be using this dataset as a sanity checker for the forward pass of the Transformer architecture discussed in class. The general intuitive notion is that we will *encode* the sequence of tokens in the sentence, and then predict a binary output based on the final state that is the output of the model.

Load the preprocessed data

We've appended a "CLS" token to the beginning of each sequence, which can be used to make predictions. The benefit of appending this token to the beginning of the sequence (rather than the end) is that we can extract it quite easily (we don't need to remove paddings and figure out the length of each individual sequence in the batch). We'll come back to this.

We've additionally already constructed a vocabulary and converted all of the strings of tokens into integers which can be used for vocabulary lookup for you. Feel free to explore the data here.

In [19]:

```
train_inxs = np.load('./data/train_inxs.npy')
val_inxs = np.load('./data/val_inxs.npy')
train_labels = np.load('./data/train_labels.npy')
val_labels = np.load('./data/val_labels.npy')

# load dictionary
word_to_ix = {}
with open("./data/word_to_ix.csv", "r") as f:
    reader = csv.reader(f)
    for line in reader:
        word_to_ix[line[0]] = line[1]
print("Vocabulary Size:", len(word_to_ix))

print(train_inxs.shape) # 7000 training instances, of (maximum/padded) length 43 words.
print(val_inxs.shape) # 1551 validation instances, of (maximum/padded) length 43 words.
print(train_labels.shape)
print(val_labels.shape)

d1 = torch.load('./data/d1.pt').to(device)
d2 = torch.load('./data/d2.pt').to(device)
d3 = torch.load('./data/d3.pt').to(device)
d4 = torch.load('./data/d4.pt').to(device)
```

```
Vocabulary Size: 1542
(7000, 43)
(1551, 43)
(7000,)
(1551,)
```

Instead of using numpy for this model, we will be using Pytorch to implement the forward pass. You will not need to implement the backward pass for the various layers in this assigment.

The file `models/Transformer.py` contains the model class and methods for each layer. This is where you will write your implementations.

4.1: Embeddings

We will format our input embeddings similarly to how they are constructed in BERT (source of figure). Recall from lecture that unlike a RNN, a Transformer does not include any positional information about the order in which the words in the sentence occur. Because of this, we need to append a positional encoding token at each position. (We will ignore the segment embeddings and [SEP] token here, since we are only encoding one sentence at a time). We have already appended the [CLS] token for you in the previous step.

Your first task is to implement the embedding lookup, including the addition of positional encodings. Open the file `Transformer.py` and complete all code parts for `Deliverable 1`.

In [20]:
```
from models.Transformer import TransformerTranslator
inputs = train_inxs[0:2]
inputs = torch.LongTensor(inputs).to(device)

model = TransformerTranslator(input_size=len(word_to_ix), output_size=2, device=device, hidden_dim=128, num_hea
model.to(device)

embeds = model.embed(inputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(embeds, d1)).item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017998493276536465
```

4.2: Multi-head Self-Attention

Attention can be computed in matrix-form using the following formula:

We want to have multiple self-attention operations, computed in parallel. Each of these is called a *head*. We concatenate the heads and multiply them with the matrix `attention_head_projection` to produce the output of this layer.

After every multi-head self-attention and feedforward layer, there is a residual connection + layer normalization. Make sure to implement this, using the following formula:

Open the file `models/transformer.py` and implement the `multihead_attention` function. We have already initialized all of the layers you will need in the constructor.

In [21]:
```
hidden_states = model.multi_head_attention(embeds)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(hidden_states, d2)).item()) # should be very small (
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017090855399146676
```

4.3: Element-Wise Feed-forward Layer

Open the file `models/transformer.py` and complete code for `Deliverable 3` : Include layer norm and addition as per transformer diagram

In [22]:
```
outputs = model.feedforward_layer(hidden_states)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(outputs, d3)).item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017097736708819866
```

4.4: Final Layer

Open the file `models/transformer.py` and complete code for `Deliverable 4` , to produce logits scores for all tokens in target language.

In [23]:
```
scores = model.final_layer(outputs)

print("Difference:", torch.sum(torch.pairwise_distance(scores, d4)).item()) # should be very small (<1e-5)
```

```
# try:
#     print("Difference:", torch.sum(torch.pairwise_distance(scores, d4)).item()) # should be very small (<1e-5
# except:
#     print("NOT IMPLEMENTED")
```

```
Difference: 2.5990862923208624e-05
```

4.5: Putting it all together

Open the file `models/transformer.py` and complete the method `forward`, by putting together all of the methods you have developed in the right order to perform a full forward pass.

In [24]:

```
inputs = train_inxs[0:2]
inputs = torch.LongTensor(inputs).to(device)

outputs = model.forward(inputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(outputs, scores)).item()) # should be very small (<1
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 2.6229752620565705e-05
```

Great! We've just implemented a Transformer forward pass for translation. One of the big perks of using PyTorch is that with a simple training loop, we can rely on automatic differentation (autograd) to do the work of the backward pass for us. This is not required for this assignment, but you can explore this on your own.

4.6: Train the Transformer

Now, we will extend your Transformer classifier to perform translation.

For simplicity, we have reduced the seq2seq translation problem to a token-for-token classification problem. The encoder will predict tokens of the target language. Hence, that this is *not* a real translation model like the seq2seq model from part 1. But as you will see, the Transformer can still learn to produce fluent text (with low perplexity). Also note that the BLEU score is a more typical metric for evaluating translations. Here, we use perplexity which is a language modeling metric for fluency.

Now you can start training the Transformer translator. We provided you with some training code and you can simply run them to see how your translator works. If you implemented everything correctly, you should see some meaningful translation in the output. Compare the results from the Seq2Seq model, which one is better? You can modify the hyperparameters to improve the results. You can also tune the BATCH_SIZE in section 1.2.

In [25]:
```
# Hyperparameters
learning_rate = 1e-2
EPOCHS = 10

# Model
trans_model = TransformerTranslator(input_size, output_size, device, max_length = MAX_LEN).to(device)

# optimizer = optim.Adam(model.parameters(), lr = learning_rate)
optimizer = torch.optim.Adam(trans_model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
```

In [26]:
```
for epoch_idx in range(EPOCHS):
    print("-----------------------------------")
    print("Epoch %d" % (epoch_idx+1))
    print("-----------------------------------")

    train_loss, avg_train_loss = train(trans_model, train_loader, optimizer, criterion)
    scheduler.step(train_loss)

    val_loss, avg_val_loss = evaluate(trans_model, valid_loader, criterion)

    avg_train_loss = avg_train_loss.item()
    avg_val_loss = avg_val_loss.item()
    print("Training Loss: %.4f. Validation Loss: %.4f. " % (avg_train_loss, avg_val_loss))
    print("Training Perplexity: %.4f. Validation Perplexity: %.4f. " % (np.exp(avg_train_loss), np.exp(avg_val_
```

```
-----------------------------------
Epoch 1
-----------------------------------


Training Loss: 4.1708. Validation Loss: 3.6846.
Training Perplexity: 64.7657. Validation Perplexity: 39.8274.
-----------------------------------
Epoch 2
-----------------------------------
```

```
Training Loss: 3.7621. Validation Loss: 3.5390.
Training Perplexity: 43.0378. Validation Perplexity: 34.4321.
-----------------------------------
Epoch 3
-----------------------------------

Training Loss: 3.6036. Validation Loss: 3.4940.
Training Perplexity: 36.7315. Validation Perplexity: 32.9177.
-----------------------------------
Epoch 4
-----------------------------------

Training Loss: 3.4887. Validation Loss: 3.4550.
Training Perplexity: 32.7434. Validation Perplexity: 31.6591.
-----------------------------------
Epoch 5
-----------------------------------

Training Loss: 3.4041. Validation Loss: 3.4568.
Training Perplexity: 30.0877. Validation Perplexity: 31.7154.
-----------------------------------
Epoch 6
-----------------------------------

Training Loss: 3.3348. Validation Loss: 3.4568.
Training Perplexity: 28.0739. Validation Perplexity: 31.7167.
-----------------------------------
Epoch 7
-----------------------------------

Training Loss: 3.2594. Validation Loss: 3.4471.
Training Perplexity: 26.0336. Validation Perplexity: 31.4103.
-----------------------------------
Epoch 8
-----------------------------------

Training Loss: 3.2025. Validation Loss: 3.4359.
Training Perplexity: 24.5929. Validation Perplexity: 31.0609.
-----------------------------------
Epoch 9
-----------------------------------

Training Loss: 3.1571. Validation Loss: 3.4256.
Training Perplexity: 23.5033. Validation Perplexity: 30.7409.
```

```
-----------------------------------
Epoch 10
-----------------------------------


Training Loss: 3.1219. Validation Loss: 3.4401.
Training Perplexity: 22.6895. Validation Perplexity: 31.1914.
```

Translations

Run the code below to see some of your translations. Modify to your liking.

In [27]:
```python
def translate(model, dataloader):
    model.eval()
    with torch.no_grad():
        # Get the progress bar
        #progress_bar = tqdm(dataloader, asci = True)
        for batch_idx, data in enumerate(dataloader):
            source = data.src.transpose(1,0)
            target = data.trg.transpose(1,0)

            translation = model(source)
            return target, translation
```

In [28]:
```python
# Select Transformer or Seq2Seq model
# model = trans_model
model = seq2seq_model
```

In [29]:
```python
#Set model equal to trans_model or seq2seq_model
target, translation = translate(model, valid_loader)
```

In [30]:
```python
raw = np.array([list(map(lambda x: TRG.vocab.itos[x], target[i])) for i in range(target.shape[0])])
print(raw)
```

```
[['<sos>' 'a' 'man' ... '<pad>' '<pad>' '<pad>']
 ['<sos>' 'a' 'man' ... '<pad>' '<pad>' '<pad>']
 ['<sos>' 'a' 'man' ... '<pad>' '<pad>' '<pad>']
 ...
 ['<sos>' 'boy' 'doing' ... '<pad>' '<pad>' '<pad>']
 ['<sos>' 'kids' 'are' ... '<pad>' '<pad>' '<pad>']
 ['<sos>' 'a' 'man' ... '<pad>' '<pad>' '<pad>']]
```

In [31]:
```
token_trans = np.argmax(translation.cpu().numpy(), axis = 2)
translated = np.array([list(map(lambda x: TRG.vocab.itos[x], token_trans[i])) for i in range(token_trans.shape[
print(translated)
```

```
[['<unk>' 'a' 'man' ... '<eos>' '<eos>' '<eos>']
 ['<unk>' 'a' 'man' ... '<eos>' '<eos>' '<eos>']
 ['<unk>' 'a' 'man' ... '<eos>' '<eos>' '<eos>']
 ...
 ['<unk>' 'a' 'in' ... '<eos>' '<eos>' '<eos>']
 ['<unk>' 'people' 'are' ... '<eos>' '<eos>' '<eos>']
 ['<unk>' 'a' 'man' ... '<eos>' '<eos>' '<eos>']]
```