CS291 Structured Prediction in NLP — Project1 Machine Translation

Sourabh Raja Murali
UC San Diego
srajamurali@ucsd.edu

May 1, 2022

1 Problem Description

In this paper we discuss about the machine translation problem by training a seq2seq architecture with an attention mechanism. The dataset we will be using for the experimentation is Multi30k machine translation dataset.
We will look at a bunch of decoding strategies like greedy decoding, beam search decoding and nucleus sampling and compare and contrast these methods to look at how they perform in comparison.

2 Task 1 Seq2Seq model

The task of machine translation in our case from German to English is achieved through seq2seq model and the model and its implementation is defined in the source code that I have submitted. We train the model on the Multi30k dataset as mentioned earlier. We run the model for 10 epochs and we get the loss curve for the training and validation as seen below in the figure1.

We can infer from the graph that the training loss slowly decreases with the increasing epochs. However the validation loss starts reducing before flattening. I have used the trained model to translate sample sentences from the validation set into English. Two such sentences where the source sentence is in German, target sentence is in English and the translated sentence which is the model prediction are shown here.
Example1
src: ”ein schwarzer hund und ein gefleckter hund kämpfen .”
trg: ”a black dog and a spotted dog are fighting ”
model prediction: ”a black dog and. a spotted dog fighting .”

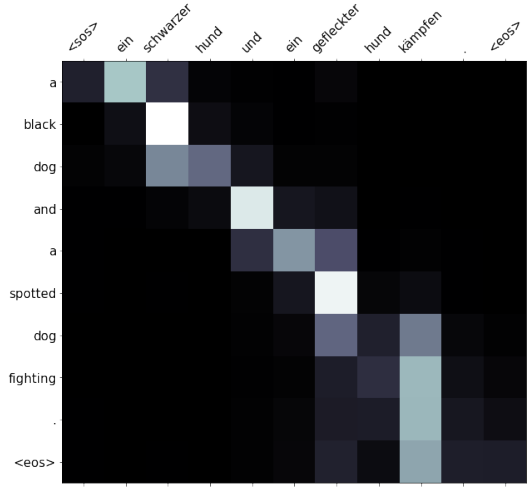


Figure 2: Attention visualization for each trg token

Example2
src: ”eine frau spielt ein lied auf ihrer geige .”
trg: ”a female playing a song on her violin .”
model prediction: ”a woman is a song on her violin .”

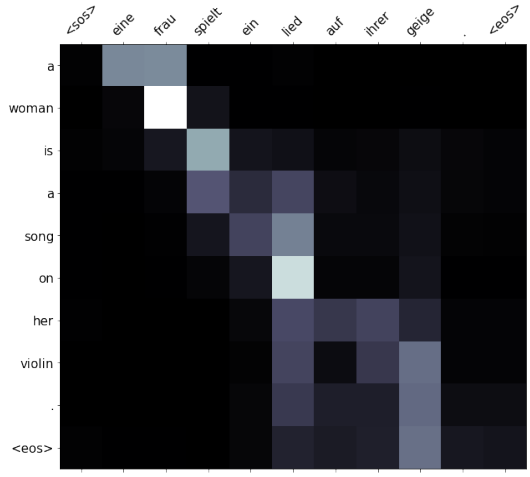


Figure 3: Attention visualization for each trg token

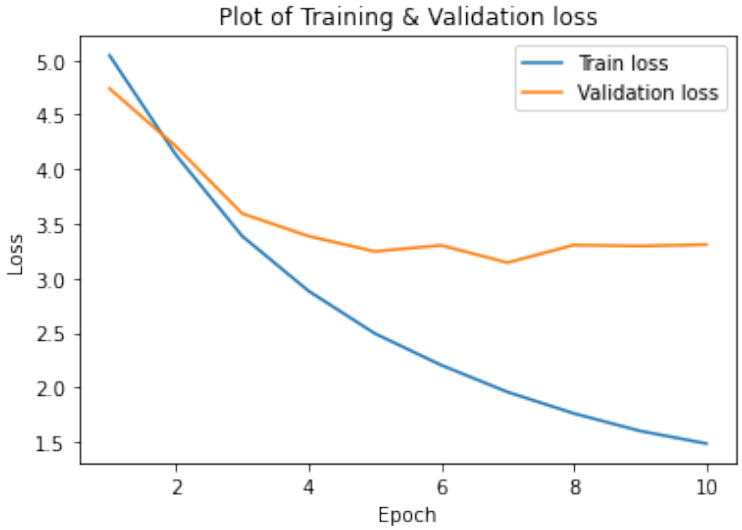


Figure 1: Training & Validation Loss vs Epochs

When we look at the heat map of attention of source sentence to each token in the target sentence we can evidently make out when words related are translated to target tokens. Like for example black dog is influencing schwarzer hund and so on. If I knew the German language I could have made much more sense out of the heat map however my knowledge is nil. When we will look at these model predictions we can make out that the model is doing a pretty good job translating sentences from German to English. However there would be cases where the translation might not make sense or is slightly off the track. The test loss for the model that we trained is **3.158** and the test perplexity turned out to be **23.52**. The bleu score using the greedy sampling method was **29.49**.

2.1 Task 3: Beam Search

As we have seen the seq2seq model leverages an encoder and decoder framework with LSTM or GRU as the basic building blocks. Encoder map a source sequence, encodes the source information and passes it to the decoder. The decoder takes the encoded data from the encoder as input and along with the ¡sos¿ token as the initial input and produces an output sequence. We do not want any random translation but the best and most likely words to be chosen for the translation into the target language. In order to select the most likely words like we saw the most easiest approach is the greedy approach where we pick the maximum probability of the word that is spit out by the decoder at each time step. Unlike greedy, beam search algorithm selects multiple alternatives from the input sequence at each time step based on the conditional probability. The count of this multiple alternatives depends on the beam width (k) that is passed as a parameter to the beam search function. At each time step, the algorithm selects k number of alternatives with the highest probabilities as the most likely probable choices for the time step. We also prune the branches of the beam search if we encounter end of sentence token as the most probable one at that time step. Therefore beam search considers multiple best options based on the beam width using conditional probability, which is better than the sub-optimal Greedy search. I have implemented this greedy search algorithm in PyTorch which can be seen in the notebook attached. I am maintaining a data structure which captures the log probability, hidden state output from the decoder and the indices list of target token till that time step. With the for loop we loop through each of this list data structure and take the top k probable sequences and branch from there. Finally we would end up in k sequences which have the highest probability values among all the possible sequences. It runs fairly quickly as I have used mostly PyTorch functions for faster computations and avoiding loops as much as possible. The samples generated by this method with beam size set to 5 is shown below and is compared to the ones obtained from greedy method. In my observation for some samples beam search gave good variations in the translations but

for few cases the other candidates from he beam search contain repetition in words. Following are 3 different examples picked from the test dataset.

Example1:
src: "die person im gestreiften shirt klettert auf einen berg ."
trg: "the person in the striped shirt is mountain climbing. "
greedy strategy translation: "the person in the striped shirt is climbing a mountain."
beam search translations:
1. the person in the striped shirt is climbing a mountain.
2. the person in the striped shirt is climbing on a mountain.
3. the person in the striped shirt is rock climbing.
4. the person in the striped shirt is rock climbing a mountain.
5. the person in the striped shirt is climbing up a mountain.

Example2:
src: "ein mann schneidet äste von bäumen ."
trg: "a man cutting branches of trees "
greedy strategy translation: "a man is chopping some trees."
beam search translations:
1. a man is chopping some trees.
2. a man is chopping some trees trees.
3. a man is chopping some some trees.
4. a man is chopping bubbles surrounded by trees.
5. a man is chopping some debris.

Example3:
src: "vier weiße hunde mit maulkörben springen über eine rote wand."
trg: "four white dogs with muzzles jump over a red wall. "
greedy strategy translation: "four white dogs with muzzles jumping over a red wall."
beam search translations:
1. four white dogs with muzzles jumping over a red wall.
2. four white dogs dogs jumping over a red wall.
3. four white dogs wearing muzzles jumping over a red wall.
4. four white dogs with muzzles over a red wall.
5. four white dogs with muzzles over over a red wall.

When we look at these sample translations we can see a diversity in most of the cases in translation from source to target. In some cases we can also see that the candidate output from greedy is also one among the beam search output translations. But that does not have to be the case always. In some cases we see repeated words whereas in some cases we see alternate meanings of the same words which is a good thing. There are also cases where there is a totally different meaning being conveyed in the beam search output sentence. This is the qualitative analysis of beam search compared to the greedy search.

Now lets look at the bleu scores on the test data. As mentioned earlier the bleu score using greedy strategy on the test data is 29.49. With the beam search strategy when the beam size is 1 I get the same bleu score of 29.49 which indicates that greedy search is a special case of beam search with beam size 1. We can see the variation of bleu score for different beam sizes from the below table and graph. I have just multiplied the bleu score by 100 to depict on the graph.

beam size	Bleu scores on the test data
1. 1	29.49
2. 3	30.81
3. 5	30.62
4. 7	30.60
5. 10	30.40

Table 1: Blue score for different beam sizes

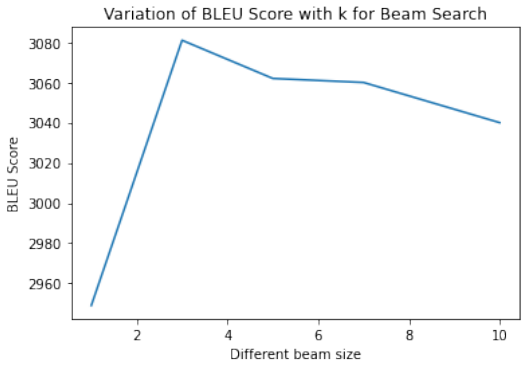


Figure 4: bleu score vs beam size

The reason for decrease in the bleu score for higher beam sizes could be that with higher beam sizes the chances of a diverse word getting picked which qualifies the top probability is more and due to this diverse set of translation words, there could be mismatches in the ngram count which the bleu

score uses and hence we might be getting a dip. As we go to higher beam sizes the bleu score might decrease and it might give better diverse translations however it will have higher computational cost.

3 Nucleus Sampling

In this method of decoding, instead of looking at just the maximum probability token, nucleus sampling focuses on the smallest possible sets of top-V words so that the sum of their probabilities are greater than or equal to p. The tokens that do not fall in this set are dropped by masking their indices and the rest are re-scaled just to ensure that they sum up to 1. Thereby the size of the set of words can dynamically increase or decrease as per the next word's probability distribution. The best part here is that with this method we not only get diverse word choices through the stochasticity of sampling, but also preserves the quality by focusing only on the top-p mass. The samples generated by nucleus sampling with p set to 0.9 are shown here and compared with the greedy search result. I referred to [1] for implementing the nucleus sampling method.

Example1
src: "eine frau spielt ein lied auf ihrer geige ."
trg: "a female playing a song on her violin "
Greedy search result: "a woman is a song on her violin ."
Nucleus Sampling result: "a woman fiddles is a song on her violin on her concert. "

Example2
src: "drei kleine kinder stehen um ein blau-weißes fass herum ."
trg: "three young children stand around a blue and white barrel "
Greedy search result: "three small children standing around a a blue elephant."
Nucleus Sampling result: "hree small children standing around a a yellow hallway. "

By observing the translations from the nucleus sampling method what we can observe is that even though sometimes we do not get nearest translation as that of greedy search we get a more diverse words and sentences which might not be the exact translation of the source sentence. Now let us look at the bleu score variation for different values of top p values. This is summarized in the below table and depicted on the below graph.

top p value	Bleu scores on the test data
1. 0.4	27.85
2. 0.5	26.84
3. 0.6	25.10
4. 0.7	23.65
5. 0.8	20.75
6. 0.9	18.64

Table 2: Blue score for different top p values

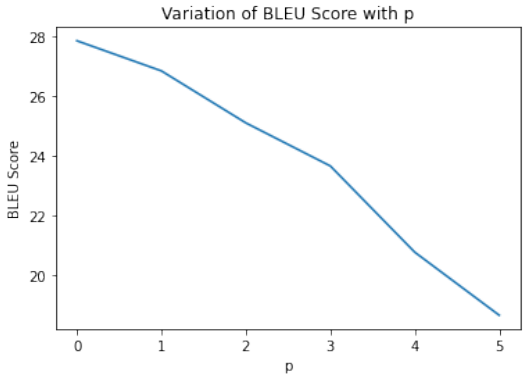


Figure 5: bleu score vs top p value

We can observe the variation of bleu score as we increase the top p value which is a parameter for the nucleus sampling function, the bleu score decreases and it gives a bleu score of 18.64 for the top p value of 0.9. The reason for this could be that bleu score involves evaluating how well the ngrams are matching between the two. But like I mentioned in nucleus sampling we are looking at diverse sentences hence there might not be an exact match in the ngrams. However for the lower values of top p, it mostly behaves like closer to greedy search as it has lesser number of high probability words to pick from. Hence the chances of picking the actual translation word is higher, therefore chances of ngrams match is higher and thereby might be getting a higher bleu score at lower values of top p.

4 Optional Tasks

I have attempted the task of training the seq2seq model on another machine translation dataset for the language pair Spanish, English. The dataset has been downloaded from the following URL "http://storage.googleapis.com/download.tensorflow.org/data". I

have used the SpaCy's Spanish tokenizer to tokenize the sentences and split the data into train, test and validation and following a similar pipeline using PyTorch's TabularDatasets. I have changed the dropout value from 0.5 to 0.6 and all other model parameters have been retained the same. The training loss and validation loss can be seen from the figure 6

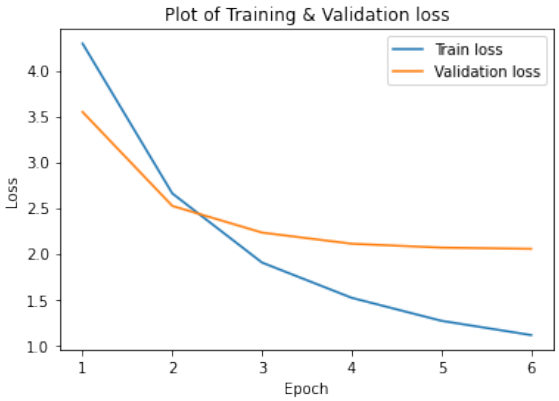


Figure 6: Training & Validation Loss vs Epochs for Spanish to Eng

The model has been trained for 6 epochs. The test loss that I obtained was 2.03 and test perplexity was 7.61.

I have tried to decode from Spanish to English using both greedy search and beam search method. Below are some sample translations. The bleu score from the greedy strategy on test data was 43.03.

Example1:
src: "¿ por qué tom querría lastimar a maría ?"
trg: "why would tom want to hurt mary ? "
greedy strategy translation: "why would tom want to hurt mary ? "
beam search translations:
1. why would tom want to hurt mary ?
2. why would tom want to kill mary ?
3. why would tom want to hurt ? ?
4. why would tom want to wait?
5. why would tom want to drive mary ?

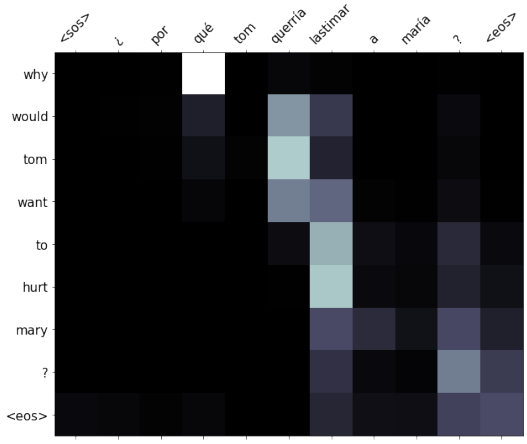


Figure 7: Attention visualization for Spanish translation

Example2:
src: "este sombrero es tuyo ."
trg: "this hat is yours . "
greedy strategy translation: "this hat is yours . "
beam search translations:
1. this hat is yours .
2. this hat is mine .
3. this hat 's yours .
4. this hat is yours . .
5. that hat is yours .

I obtained a bleu score of 44.82 for a beam size of 5 on the test data.

References

[1] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Introduction

In this notebook we will be adding a few improvements - packed padded sequences and masking - to the model from the previous notebook. Packed padded sequences are used to tell our RNN to skip over padding tokens in our encoder. Masking explicitly forces the model to ignore certain values, such as attention over padded elements. Both of these techniques are commonly used in NLP.

We will also look at how to use our model for inference, by giving it a sentence, seeing what it translates it as and seeing where exactly it pays attention to when translating each word.

Finally, we'll use the BLEU metric to measure the quality of our translations.

Preparing Data

First, we'll import all the modules as before, with the addition of the `matplotlib` modules used for viewing the attention.

```
%cd /content/drive/MyDrive/UCSD_courses/CS291A/assignment1/
```

```
/content/drive/MyDrive/UCSD_courses/CS291A/assignment1
```

```
!pip install torch==1.8.0+cu111 torchvision==0.9.0+cu111 torchaudio==0.8.0 -f https://download.pytorch.org/whl/torch_stable.html
!pip install torchtext==0.9.0
```

```
#Used to run this before
# !pip install -U torch==1.8.0 torchtext==0.9.0
# !pip install -U torch==1.8.0+cu111 torchtext==0.9.0+cu111 -f https://download.pytorch.org/whl/torch_stable.html
# Reload environment
# exit()
```

```
!python -m spacy download en_core_web_sm
!python -m spacy download de_core_news_sm
exit()
```

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

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

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import spacy
import numpy as np

import random
import math
import time
import json
import pickle
import os
```

```
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
```

Next, we'll set the random seed for reproducability.

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

As before, we'll import spaCy and define the German and English tokenizers.

```
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

```
def tokenize_de(text):
    """
    Tokenizes German text from a string into a list of strings
    """
    return [tok.text for tok in spacy_de.tokenizer(text)]

def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

When using packed padded sequences, we need to tell PyTorch how long the actual (non-padded) sequences are. Luckily for us, TorchText's `Field` objects allow us to use the `include_lengths` argument, this will cause our `batch.src` to be a tuple. The first element of the tuple is the same as before, a batch of numericalized source sentence as a tensor, and the second element is the non-padded lengths of each source sentence within the batch.

```
SRC = Field(
#   tokenize = tokenize_de,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True,
    include_lengths = True
)

TRG = Field(
    #tokenize = tokenize_en,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True
)
```

We then load the data.

```
# NOTE: this line takes a long time to run on Colab so
# instead we'll load the already tokenized json dataset
# from Github
#
# train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
#                                                     fields = (SRC, TRG))

# fetch from Github repo
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/train.json
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/valid.json
!wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/test.json

# and load to same variables
fields = {'src': ('src', SRC), 'trg': ('trg', TRG)}
train_data, valid_data, test_data = TabularDataset.splits(
    path = '.',
    train = 'train.json',
    validation = 'valid.json',
    test = 'test.json',
    format = 'json',
    fields = fields
)
```

And build the vocabulary.

```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

Next, we handle the iterators.

One quirk about packed padded sequences is that all elements in the batch need to be sorted by their non-padded lengths in descending order, i.e. the first sentence in the batch needs to be the longest. We use two arguments of the iterator to handle this, `sort_within_batch` which tells the iterator that the contents of the batch need to be sorted, and `sort_key` a function which tells the iterator how to sort the elements in the batch. Here, we sort by the length of the `src` sentence.

```
BATCH_SIZE = 128

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

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
```

```
    (train_data, valid_data, test_data),
     batch_size = BATCH_SIZE,
     sort_within_batch = True,
     sort_key = lambda x : len(x.src),
     device = device)
```

Building the Model

Encoder

Next up, we define the encoder.

The changes here all within the `forward` method. It now accepts the lengths of the source sentences as well as the sentences themselves.

After the source sentence (padded automatically within the iterator) has been embedded, we can then use `pack_padded_sequence` on it with the lengths of the sentences. Note that the tensor containing the lengths of the sequences must be a CPU tensor as of the latest version of PyTorch, which we explicitly do so with `to('cpu')`. `packed_embedded` will then be our packed padded sequence. This can be then fed to our RNN as normal which will return `packed_outputs`, a packed tensor containing all of the hidden states from the sequence, and `hidden` which is simply the final hidden state from our sequence. `hidden` is a standard tensor and not packed in any way, the only difference is that as the input was a packed sequence, this tensor is from the final **non-padded element** in the sequence.

We then unpack our `packed_outputs` using `pad_packed_sequence` which returns the `outputs` and the lengths of each, which we don't need.

The first dimension of `outputs` is the padded sequence lengths however due to using a packed padded sequence the values of tensors when a padding token was the input will be all zeros.

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_len):

        #src = [src len, batch size]
        #src_len = [batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        #need to explicitly put lengths on cpu!
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, src_len.to('cpu'))

        packed_outputs, hidden = self.rnn(packed_embedded)

        #packed_outputs is a packed sequence containing all hidden states
        #hidden is now from the final non-padded element in the batch

        outputs, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs)

        #outputs is now a non-packed sequence, all hidden states obtained
        #  when the input is a pad token are all zeros

        #outputs = [src len, batch size, hid dim * num directions]
        #hidden = [n layers * num directions, batch size, hid dim]

        #hidden is stacked [forward_1, backward_1, forward_2, backward_2, ...]
        #outputs are always from the last layer

        #hidden [-2, :, : ] is the last of the forwards RNN
        #hidden [-1, :, : ] is the last of the backwards RNN

        #initial decoder hidden is final hidden state of the forwards and backwards
        #  encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)))

        #outputs = [src len, batch size, enc hid dim * 2]
        #hidden = [batch size, dec hid dim]

        return outputs, hidden
```

Attention

The attention module is where we calculate the attention values over the source sentence.

Previously, we allowed this module to "pay attention" to padding tokens within the source sentence. However, using *masking*, we can force the attention to only be over non-padding elements.

The `forward` method now takes a `mask` input. This is a **[batch size, source sentence length]** tensor that is 1 when the source sentence token is not a padding token, and 0 when it is a padding token. For example, if the source sentence is: ["hello", "how", "are", "you", "?", `<pad>`, `<pad>`], then the mask would be [1, 1, 1, 1, 1, 0, 0].

We apply the mask after the attention has been calculated, but before it has been normalized by the `softmax` function. It is applied using `masked_fill`. This fills the tensor at each element where the first argument (`mask == 0`) is true, with the value given by the second argument (`-1e10`). In other words, it will take the un-normalized attention values, and change the attention values over padded elements to be `-1e10`. As these numbers will be miniscule compared to the other values they will become zero when passed through the `softmax` layer, ensuring no attention is payed to padding tokens in the source sentence.

[ ] ↳ *1 cell hidden*

Decoder

The decoder only needs a few small changes. It needs to accept a mask over the source sentence and pass this to the attention module. As we want to view the values of attention during inference, we also return the attention tensor.

[ ] ↳ *1 cell hidden*

Seq2Seq

The overarching seq2seq model also needs a few changes for packed padded sequences, masking and inference.

We need to tell it what the indexes are for the pad token and also pass the source sentence lengths as input to the `forward` method.

We use the pad token index to create the masks, by creating a mask tensor that is 1 wherever the source sentence is not equal to the pad token. This is all done within the `create_mask` function.

The sequence lengths as needed to pass to the encoder to use packed padded sequences.

The attention at each time-step is stored in the `attentions`

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, src_pad_idx, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.device = device

    def create_mask(self, src):
        mask = (src != self.src_pad_idx).permute(1, 0)
        return mask

    def forward(self, src, src_len, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #src_len = [batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use teacher forcing 75% of the time

        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #encoder_outputs is all hidden states of the input sequence, back and forwards
        #hidden is the final forward and backward hidden states, passed through a linear layer
        encoder_outputs, hidden = self.encoder(src, src_len)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        mask = self.create_mask(src)

```

```
        #mask = [batch size, src len]

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden state, all encoder hidden states
            #  and mask
            #receive output tensor (predictions) and new hidden state
            output, hidden, _ = self.decoder(input, hidden, encoder_outputs, mask)

            #place predictions in a tensor holding predictions for each token
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        return outputs
```

Training the Seq2Seq Model

Next up, initializing the model and placing it on the GPU.

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 512
DEC_HID_DIM = 512
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5
SRC_PAD_IDX = SRC.vocab.stoi[SRC.pad_token]

attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT, attn)

model = Seq2Seq(enc, dec, SRC_PAD_IDX, device).to(device)
```

Then, we initialize the model parameters.

```
def init_weights(m):
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.normal_(param.data, mean=0, std=0.01)
        else:
            nn.init.constant_(param.data, 0)

model.apply(init_weights)
```

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(7853, 256)
    (rnn): GRU(256, 512, bidirectional=True)
    (fc): Linear(in_features=1024, out_features=512, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (attention): Attention(
      (attn): Linear(in_features=1536, out_features=512, bias=True)
      (v): Linear(in_features=512, out_features=1, bias=False)
    )
    (embedding): Embedding(5893, 256)
    (rnn): GRU(1280, 512)
    (fc_out): Linear(in_features=1792, out_features=5893, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

We'll print out the number of trainable parameters in the model, noticing that it has the exact same amount of parameters as the model without these improvements.

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 20,518,405 trainable parameters
```

Then we define our optimizer and criterion.

The `ignore_index` for the criterion needs to be the index of the pad token for the target language, not the source language.

```
optimizer = optim.Adam(model.parameters())
```

```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

Next, we'll define our training and evaluation loops.

As we are using `include_lengths = True` for our source field, `batch.src` is now a tuple with the first element being the numericalized tensor representing the sentence and the second element being the lengths of each sentence within the batch.

Our model also returns the attention vectors over the batch of source source sentences for each decoding time-step. We won't use these during the training/evaluation, but we will later for inference.

```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src, src_len = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, src_len, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src, src_len = batch.src
            trg = batch.trg

            output = model(src, src_len, trg, 0) #turn off teacher forcing
```

```
            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Then, we'll define a useful function for timing how long epochs take.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

The penultimate step is to train our model. Notice how it takes almost half the time as our model without the improvements added in this notebook.

```
# N_EPOCHS = 10
# CLIP = 1

# best_valid_loss = float('inf')

# loss_di = {"epoch_num":[],"train_loss":[],"valid_loss":[],"train_perp":[],"valid_perp":[]}

# for epoch in range(N_EPOCHS):

#     start_time = time.time()

#     train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
#     valid_loss = evaluate(model, valid_iterator, criterion)

#     end_time = time.time()

#     epoch_mins, epoch_secs = epoch_time(start_time, end_time)

#     if valid_loss < best_valid_loss:
#         best_valid_loss = valid_loss
#         torch.save(model.state_dict(), 'tut4-model.pt')

#     print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
#     loss_di["epoch_num"].append(epoch+1)
#     loss_di["train_loss"].append(train_loss)
#     loss_di["train_perp"].append(math.exp(train_loss))
#     loss_di["valid_loss"].append(valid_loss)
#     loss_di["valid_perp"].append(math.exp(valid_loss))
#     print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
#     print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')

# with open("train_loss.pkl","wb") as f:
#     pickle.dump(loss_di,f)
```

Finally, we load the parameters from our best validation loss and get our results on the test set.

We get the improved test perplexity whilst almost being twice as fast!

```
model.load_state_dict(torch.load('tut4-model.pt'))

test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
| Test Loss: 3.158 | Test PPL:  23.528 |
```

Inference

Now we can use our trained model to generate translations.

Note: these translations will be poor compared to examples shown in paper as they use hidden dimension sizes of 1000 and train for 4 days! They have been cherry picked in order to show off what attention should look like on a sufficiently sized model.

Our `translate_sentence` will do the following:

- ensure our model is in evaluation mode, which it should always be for inference
- tokenize the source sentence if it has not been tokenized (is a string)
- numericalize the source sentence
- convert it to a tensor and add a batch dimension
- get the length of the source sentence and convert to a tensor
- feed the source sentence into the encoder
- create the mask for the source sentence
- create a list to hold the output sentence, initialized with an `<sos>` token
- create a tensor to hold the attention values
- while we have not hit a maximum length
    - get the input tensor, which should be either `<sos>` or the last predicted token
    - feed the input, all encoder outputs, hidden state and mask into the decoder
    - store attention values
    - get the predicted next token
    - add prediction to current output sentence prediction
    - break if the prediction was an `<eos>` token
- convert the output sentence from indexes to tokens
- return the output sentence (with the `<sos>` token removed) and the attention values over the sequence

```
def translate_sentence(sentence, src_field, trg_field, model, device, max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]


    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)


    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)

    for i in range(max_len):

        trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)

        with torch.no_grad():
            output, hidden, attention = model.decoder(trg_tensor, hidden, encoder_outputs, mask)

        attentions[i] = attention
        pred_token = output.argmax(1).item()

        trg_indexes.append(pred_token)

        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]
```

```
    # print("Log_prob:",np.log(output.max().item()))

    return trg_tokens[1:], attentions[:len(trg_tokens)-1]
```

Next, we'll make a function that displays the model's attention over the source sentence for each target token generated.

```
def display_attention(sentence, translation, attention):

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)

    attention = attention.squeeze(1).cpu().detach().numpy()

    cax = ax.matshow(attention, cmap='bone')

    ax.tick_params(labelsize=15)

    x_ticks = [''] + ['<sos>'] + [t.lower() for t in sentence] + ['<eos>']
    y_ticks = [''] + translation

    ax.set_xticklabels(x_ticks, rotation=45)
    ax.set_yticklabels(y_ticks)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
    plt.close()
```

Now, we'll grab some translations from our dataset and see how well our model did. Note, we're going to cherry pick examples here so it gives us something interesting to look at, but feel free to change the `example_idx` value to look at different examples.

First, we'll get a source and target from our dataset.

```
example_idx = 12

src = vars(train_data.examples[example_idx])['src']
trg = vars(train_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['ein', 'schwarzer', 'hund', 'und', 'ein', 'gefleckter', 'hund', 'kämpfen', '.']
trg = ['a', 'black', 'dog', 'and', 'a', 'spotted', 'dog', 'are', 'fighting']
```

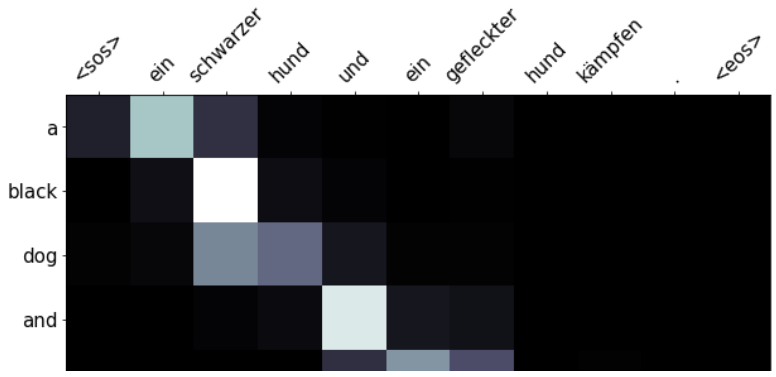
Then we'll use our `translate_sentence` function to get our predicted translation and attention. We show this graphically by having the source sentence on the x-axis and the predicted translation on the y-axis. The lighter the square at the intersection between two words, the more attention the model gave to that source word when translating that target word.

Below is an example the model attempted to translate, it gets the translation correct except changes *are fighting* to just *fighting*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
Log_prob: 2.5583669014150225
predicted trg = ['a', 'black', 'dog', 'and', 'a', 'spotted', 'dog', 'fighting', '.', '<eos>']
```

```
display_attention(src, translation, attention)
```



Translations from the training set could simply be memorized by the model. So it's only fair we look at translations from the validation and testing set too.

Starting with the validation set, let's get an example.

```
example_idx = 14

src = vars(valid_data.examples[example_idx])['src']
trg = vars(valid_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

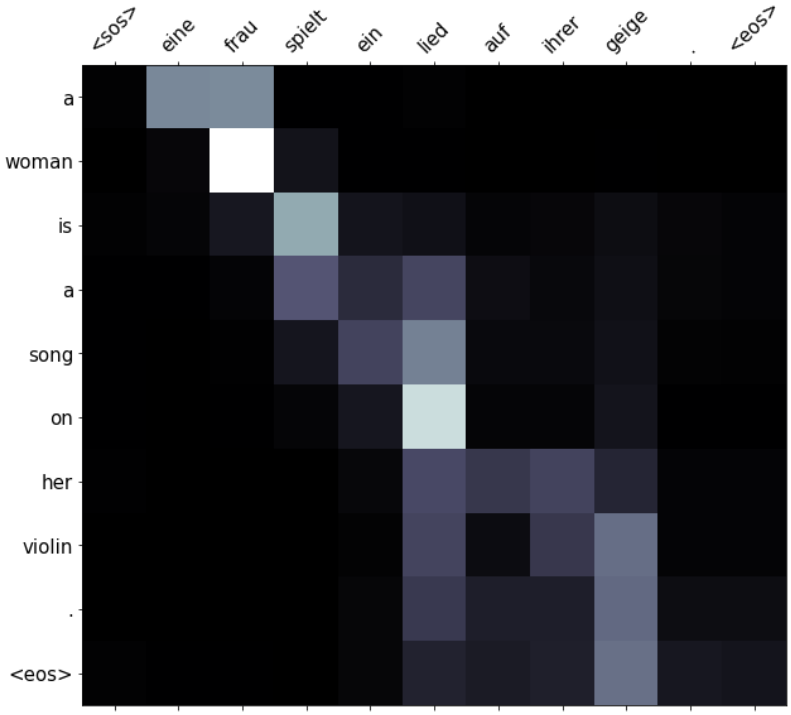
```
src = ['eine', 'frau', 'spielt', 'ein', 'lied', 'auf', 'ihrer', 'geige', '.']
trg = ['a', 'female', 'playing', 'a', 'song', 'on', 'her', 'violin', '.']
```

Then let's generate our translation and view the attention.

Here, we can see the translation is the same except for swapping *female* with *woman*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
predicted trg = ['a', 'woman', 'is', 'a', 'song', 'on', 'her', 'violin', '.', '<eos>']
```



Finally, let's get an example from the test set.

```
example_idx = 18
```

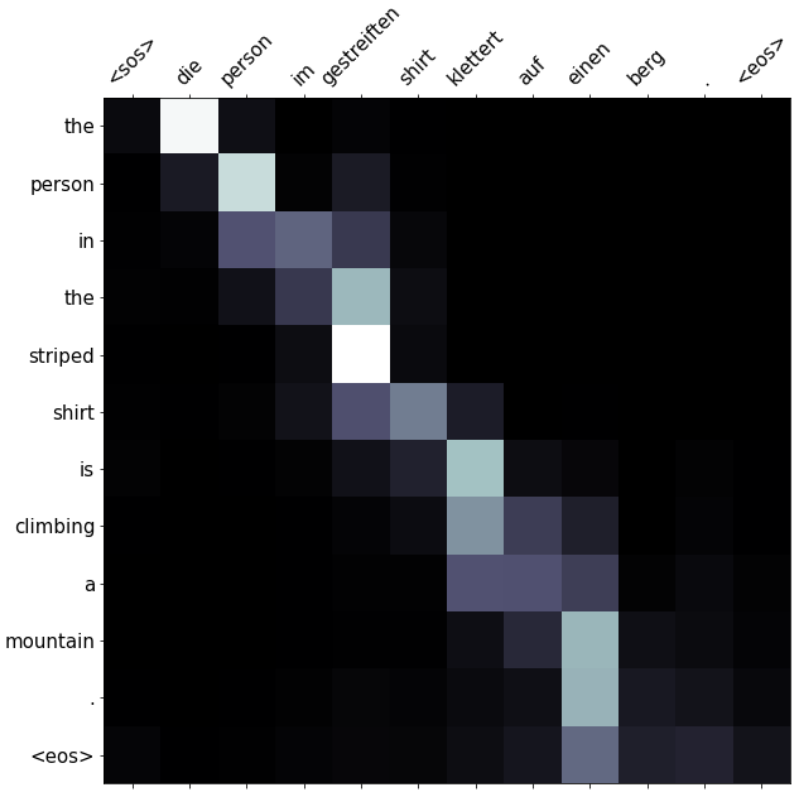
```
src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['die', 'person', 'im', 'gestreiften', 'shirt', 'klettert', 'auf', 'einen', 'berg', '.']
trg = ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'mountain', 'climbing', '.']
```

Again, it produces a slightly different translation than target, a more literal version of the source sentence. It swaps *mountain climbing* for *climbing a mountain*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
Log_prob: 2.7135808763030007
predicted trg = ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'a', 'mountain', '.', '<eos>']
```



BLEU

Previously we have only cared about the loss/perplexity of the model. However there metrics that are specifically designed for measuring the quality of a translation - the most popular is *BLEU*. Without going into too much detail, BLEU looks at the overlap in the predicted and actual target sequences in terms of their n-grams. It will give us a number between 0 and 1 for each sequence, where 1 means there is perfect overlap, i.e. a perfect translation, although is usually shown between 0 and 100. BLEU was designed for multiple candidate translations per source sequence, however in this dataset we only have one candidate per source.

We define a `calculate_bleu` function which calculates the BLEU score over a provided TorchText dataset. This function creates a corpus of the actual and predicted translation for each source sentence and then calculates the BLEU score.

```
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

```

```
        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = translate_sentence(src, src_field, trg_field, model, device, max_len)

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)

def calculate_bleu_beam(data, src_field, trg_field, model, device,beam_size = 5, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = beam_search_decoder(src, src_field, trg_field, model, device,beam_size, max_len)

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)
```

We get a BLEU of around 28. If we compare it to the paper that the attention model is attempting to replicate, they achieve a BLEU score of 26.75. This is similar to our score, however they are using a completely different dataset and their model size is much larger - 1000 hidden dimensions which takes 4 days to train! - so we cannot really compare against that either.

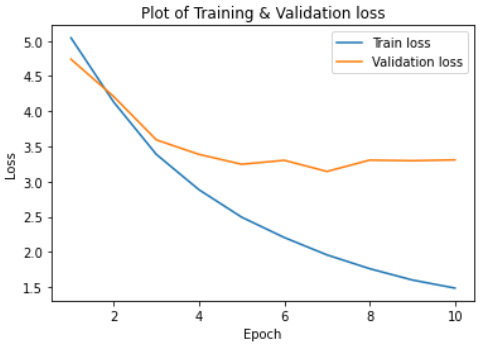
This number isn't really interpretable, we can't really say much about it. The most useful part of a BLEU score is that it can be used to compare different models on the same dataset, where the one with the **higher** BLEU score is "better".

```
bleu_score_greedy = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score_greedy*100:.2f}')
```

```
BLEU score = 29.49
```

In the next tutorials we will be moving away from using recurrent neural networks and start looking at other ways to construct sequence-to-sequence models. Specifically, in the next tutorial we will be using convolutional neural networks.

```
# with open("train_loss.pkl","rb") as f:
#     loss_di = pickle.load(f)

# import matplotlib.pyplot as plt

# plt.plot(loss_di['epoch_num'],loss_di['train_loss'],label = "Train loss")
# plt.plot(loss_di['epoch_num'],loss_di['valid_loss'],label = "Validation loss")
# plt.xlabel("Epoch")
# plt.ylabel("Loss")
# plt.title("Plot of Training & Validation loss")
# plt.legend()
# plt.show()
```



```
#Beam search implementation function

def beam_search_decoder(sentence, src_field, trg_field, model, device, beam_size = 5,max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]

    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    #obtaining encoder output once (outside of beam_search and re-used later everytime)
    with torch.no_grad():
        encoder_outputs, hidden_inp_dec = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)

    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)


    #Sequences hold the log probs, decoders hidden state and current decoded sequence
    sequences = [[0.0,hidden_inp_dec,trg_indexes]] #0.0, hidden, [4]

    for i in range(max_len):
      all_candidates = [] #to capture the new beams

      for beam in range(len(sequences)):
        current_beam = sequences[beam]
        beam_index = current_beam[2]
        beam_hidden = current_beam[1]

        trg_tensor = torch.LongTensor([beam_index[-1]]).to(device)
        if trg_field.vocab.stoi[trg_field.eos_token] in beam_index: #if end of sentence token is encountered prune that branch
          all_candidates.append(current_beam)
          continue

        with torch.no_grad():
          output, hidden, attention = model.decoder(trg_tensor, beam_hidden, encoder_outputs, mask)


        attentions[i] = attention
        # output_new = (output - output.min()) / (output.max() - output.min())
        # output_new = output_new + 0.00000001
        # log_prob_vocab = torch.log(output_new)
        log_prob_vocab = F.log_softmax(output)
        log_prob_vocab = current_beam[0] + log_prob_vocab
        # print("After:",log_prob_vocab[0][0])
        log_probs_k, indices_k = torch.topk(log_prob_vocab, beam_size)

        for i in range(len(log_probs_k[0])):
          temp = [log_probs_k[0][i].item(),hidden,beam_index + [indices_k[0][i].item()]]
          # print(temp)
          all_candidates.append(temp)

      sequences = sorted(all_candidates,key = lambda x: x[0],reverse = True)[:beam_size]

    trg_tokens_list = []
    for i in range(len(sequences)):
      trg_tokens = [trg_field.vocab.itos[i] for i in sequences[i][2]]
      trg_tokens_list.append(trg_tokens[1:])
      print("Prob_value:",sequences[i][0],trg_tokens[1:])

    # return trg_tokens_list,attentions[:len(trg_tokens)-1]
    trg_tokens_highest = [trg_field.vocab.itos[i] for i in sequences[0][2]]
    # print("Target_toks:",trg_tokens[1:])
    return trg_tokens_highest[1:], attentions[:len(trg_tokens_highest)-1]


```

```
#Eg 1

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
# for i in translation:
#   print(f'predicted trg = {i}')
# beam_search_decoder(src, SRC, TRG, model, device,3,50)
```

```
Prob_value: -3.3884968757629395 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'a', 'mountain', '.', '<eos>']
Prob_value: -3.6170332431793213 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'on', 'a', 'mountain', '.', '<e
Prob_value: -4.650404453277588 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'rock', 'climbing', '.', '<eos>']
Prob_value: -4.941495418548584 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'rock', 'climbing', 'a', 'mountain', '.', '<
Prob_value: -5.094176292419434 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'up', 'a', 'mountain', '.', '<eo
predicted trg = ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'a', 'mountain', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Eg 2

example_idx = 18

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

```
src = ['die', 'person', 'im', 'gestreiften', 'shirt', 'klettert', 'auf', 'einen', 'berg', '.']
trg = ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'mountain', 'climbing', '.']
Prob_value: -3.3884968757629395 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'a', 'mountain', '.', '<eos>']
Prob_value: -3.6170332431793213 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'on', 'a', 'mountain', '.', '<e
Prob_value: -4.650404453277588 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'rock', 'climbing', '.', '<eos>']
Prob_value: -4.941495418548584 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'rock', 'climbing', 'a', 'mountain', '.', '<
Prob_value: -5.094176292419434 ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'up', 'a', 'mountain', '.', '<eo
predicted trg = ['the', 'person', 'in', 'the', 'striped', 'shirt', 'is', 'climbing', 'a', 'mountain', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Eg 3

example_idx = 26

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)
print(f'predicted trg = {translation}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

```
src = ['ein', 'mann', 'schneidet', 'äste', 'von', 'bäumen', '.']
trg = ['a', 'man', 'cutting', 'branches', 'of', 'trees', '.']
predicted trg = ['a', 'man', 'is', 'chopping', 'some', 'trees', '.', '<eos>']
Prob_value: -5.643975734710693 ['a', 'man', 'is', 'chopping', 'some', 'trees', '.', '<eos>']
Prob_value: -6.476919174194336 ['a', 'man', 'is', 'chopping', 'some', 'trees', 'trees', '.', '<eos>']
Prob_value: -6.548498153686523 ['a', 'man', 'is', 'chopping', 'some', 'some', 'trees', '.', '<eos>']
Prob_value: -6.951023101806641 ['a', 'man', 'is', 'chopping', 'bubbles', 'surrounded', 'by', 'trees', '.', '<eos>']
Prob_value: -6.9535746574401855 ['a', 'man', 'is', 'chopping', 'some', 'debris', '.', '<eos>']
predicted trg = ['a', 'man', 'is', 'chopping', 'some', 'trees', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Example 4
example_idx = 125

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)
print(f'predicted trg = {translation}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

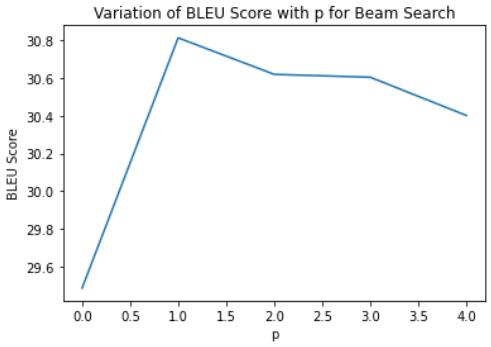
```
src = ['vier', 'weiße', 'hunde', 'mit', 'maulkörben', 'springen', 'über', 'eine', 'rote', 'wand', '.']
```

```
trg = ['four', 'white', 'dogs', 'with', 'muzzles', 'jump', 'over', 'a', 'red', 'wall', '.']
predicted trg = ['four', 'white', 'dogs', 'with', 'muzzles', 'jumping', 'over', 'a', 'red', 'wall', '.', '<eos>']
Prob_value: -3.3962433338165283 ['four', 'white', 'dogs', 'with', 'muzzles', 'jumping', 'over', 'a', 'red', 'wall', '.', '<eos>']
Prob_value: -4.820192337036133 ['four', 'white', 'dogs', 'dogs', 'jumping', 'over', 'a', 'red', 'wall', '.', '<eos>']
Prob_value: -5.057977676391602 ['four', 'white', 'dogs', 'wearing', 'muzzles', 'jumping', 'over', 'a', 'red', 'wall', '.', '<eos>']
Prob_value: -5.07942533493042 ['four', 'white', 'dogs', 'with', 'muzzles', 'over', 'a', 'red', 'wall', '.', '<eos>']
Prob_value: -5.214841842651367 ['four', 'white', 'dogs', 'with', 'muzzles', 'over', 'over', 'a', 'red', 'wall', '.', '<eos>']
predicted trg = ['four', 'white', 'dogs', 'with', 'muzzles', 'jumping', 'over', 'a', 'red', 'wall', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
bleu_score_b = calculate_bleu_beam(test_data, SRC, TRG, model, device,4)

print(f'BLEU score = {bleu_score*100:.2f}')
```

```
beam_sizes = [1, 3, 5, 7, 10]
# beam_sizes = [1, 3]
bleus_beam = []
for i in beam_sizes:
  score = calculate_bleu_beam(test_data, SRC, TRG, model, device, i)
  bleus_beam.append(score)
  print(f'BLEU score = {score*100:.2f}')


import matplotlib.pyplot as plt
bleu_scores_beam = [100*i for i in bleus_beam]
plt.plot(beam_sizes,bleu_scores_beam)
plt.title('Variation of BLEU Score with p for Beam Search')

plt.xlabel('p')
plt.ylabel('BLEU Score')
# plt.savefig('Variation of bleu with p.png')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
BLEU score = 29.49
BLEU score = 30.81
BLEU score = 30.62
BLEU score = 30.60
BLEU score = 30.40
```




```
import matplotlib.pyplot as plt
beam_sizes = [1, 3, 5, 7, 10]
bleus_beam = [29.49,30.81,30.62,30.60,30.40]
bleu_scores_beam = [100*i for i in bleus_beam]
plt.plot(beam_sizes,bleu_scores_beam)
plt.title('Variation of BLEU Score with k for Beam Search')
plt.xlabel('Different beam size')
plt.ylabel('BLEU Score')
# plt.savefig('Variation of bleu with p.png')
plt.show()
```

Variation of BLEU Score with k for Beam Search

```
def nucleus_sampling_decoder(sentence, src_field, trg_field, model, device,top_p = 0.9, temperature = 1,max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]

    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    #obtaining encoder output once (outside of beam_search and re-used later everytime)
    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)

    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)

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

      scaled_output = output/temperature
      log_prob = F.softmax(scaled_output, dim = -1)
      # print("Log_prob:",log_prob)
      log_prob_sorted, sorted_indices = torch.sort(log_prob, descending = True)
      cum_prob = torch.cumsum(log_prob_sorted, dim = -1)
      # print("Cum_prob:",cum_prob)
      sorted_indices_to_remove = cum_prob > top_p
      # print("Indices_to_remove:",sorted_indices_to_remove)

      # Shift the indices to the right to keep also the first token above the threshold
      sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
      # print("Indices_to_remove after:",sorted_indices_to_remove)
      sorted_indices_to_remove[..., 0] = 0
      # print("sorted_indices_to_remove after_0:",sorted_indices_to_remove)
      indices_to_remove = sorted_indices_to_remove.scatter(1, sorted_indices, sorted_indices_to_remove)
      # print("Indices_to_remove:",indices_to_remove)
      log_prob = log_prob.masked_fill(indices_to_remove, 0)
      log_prob = F.normalize(log_prob)
      predicted_index = torch.multinomial(log_prob,1)
      trg_indexes.append(predicted_index)
      if predicted_index == trg_field.vocab.stoi[trg_field.eos_token]:
        break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]
    return trg_tokens[1:], attentions[:len(trg_tokens)-1]

```

```
"""Example 1 """
example_idx = 14

src = vars(valid_data.examples[example_idx])['src']
src_s = ' '.join(src)
trg = vars(valid_data.examples[example_idx])['trg']
trg_s = ' '.join(trg)

print(f'src = {src_s}')
print(f'trg = {trg_s}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'Greedy predicted trg = {translation}')

translation, attention = nucleus_sampling_decoder(src, SRC, TRG, model, device)

print(f'Nucleus sampling predicted trg = {translation}')
```

```
src = eine frau spielt ein lied auf ihrer geige .
trg = a female playing a song on her violin .
Greedy predicted trg = ['a', 'woman', 'is', 'a', 'song', 'on', 'her', 'violin', '.', '<eos>']
Nucleus sampling predicted trg = ['a', 'woman', 'fiddles', 'is', 'a', 'song', 'on', 'her', 'violin', 'on', 'her', 'concert', '.', '<
```

```
"""Example 2 """
example_idx = 12

src = vars(valid_data.examples[example_idx])['src']
src_s = ' '.join(src)
trg = vars(valid_data.examples[example_idx])['trg']
trg_s = ' '.join(trg)

print(f'src = {src_s}')
print(f'trg = {trg_s}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'Greedy predicted trg = {translation}')

translation, attention = nucleus_sampling_decoder(src, SRC, TRG, model, device)

print(f'Nucleus sampling predicted trg = {translation}')
```

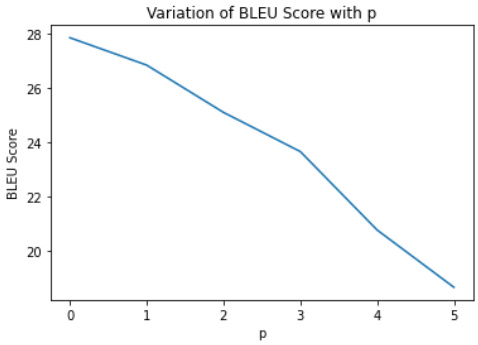
```
src = drei kleine kinder stehen um ein blau-weißes fass herum .
trg = three young children stand around a blue and white barrel .
Greedy predicted trg = ['three', 'small', 'children', 'standing', 'around', 'a', 'a', 'blue', 'elephant', '.', '<eos>']
Nucleus sampling predicted trg = ['three', 'small', 'children', 'standing', 'around', 'a', 'a', 'yellow', 'hallway', '.', '<eos>']
```

```
def calculate_bleu_nucleus(data, src_field, trg_field, model, device,top_p, temperature, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = nucleus_sampling_decoder(src, src_field, trg_field, model, device,top_p, temperature, max_len)

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)
```

```
l = [0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
bleus = []
for i in l:
  score = calculate_bleu_nucleus(test_data, SRC, TRG, model, device, i, 1.0)
  bleus.append(score)
  print(f'BLEU score = {score*100:.2f}')
```

```
BLEU score = 27.85
BLEU score = 26.84
BLEU score = 25.10
BLEU score = 23.65
BLEU score = 20.75
BLEU score = 18.64
```

```
import matplotlib.pyplot as plt
bleu_scores_nucleus = [100*i for i in bleus]
plt.plot(bleu_scores_nucleus)
plt.title('Variation of BLEU Score with p')

plt.xlabel('p')
plt.ylabel('BLEU Score')
# plt.savefig('Variation of bleu with p.png')
plt.show()
```


Variation of BLEU Score with p
BLEU Score
p
28
26
24
22
20
0
1
2
3
4
5

Introduction

In this notebook we will be adding a few improvements - packed padded sequences and masking - to the model from the previous notebook. Packed padded sequences are used to tell our RNN to skip over padding tokens in our encoder. Masking explicitly forces the model to ignore certain values, such as attention over padded elements. Both of these techniques are commonly used in NLP.

We will also look at how to use our model for inference, by giving it a sentence, seeing what it translates it as and seeing where exactly it pays attention to when translating each word.

Finally, we'll use the BLEU metric to measure the quality of our translations.

Preparing Data

First, we'll import all the modules as before, with the addition of the `matplotlib` modules used for viewing the attention.

```
# from torchtext import data
# class TabularDataset_From_List(data.Dataset):

#     def __init__(self, input_list, format, fields, skip_header=False,
#         make_example = {
#             'json': Example.fromJSON, 'dict': Example.fromdict,
#             'tsv': Example.fromTSV, 'csv': Example.fromCSV}[format.low

#         examples = [make_example(item, fields) for item in input_list]

#         if make_example in (Example.fromdict, Example.fromJSON):
#             fields, field_dict = [], fields
#             for field in field_dict.values():
#                 if isinstance(field, list):
#                     fields.extend(field)
#                 else:
#                     fields.append(field)

#         super(TabularDataset_From_List, self).__init__(examples, field

#     @classmethod
#     def splits(cls, path=None, root='.data', train=None, validation=No
#                test=None, **kwargs):
#         if path is None:
#             path = cls.download(root)
#         train_data = None if train is None else cls(
#             train, **kwargs)
#         val_data = None if validation is None else cls(
#             validation, **kwargs)
#         test_data = None if test is None else cls(
#             test, **kwargs)
#         return tuple(d for d in (train_data, val_data, test_data)
#                      if d is not None)
```

"@classmethod" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
%cd /content/drive/MyDrive/UCSD_courses/CS291A/assignment1/span/
```

```
/content/drive/MyDrive/UCSD_courses/CS291A/assignment1/span
```

```
!pip install torch==1.8.0+cu111 torchvision==0.9.0+cu111 torchaudio==0.8.0 -f https://download.pytorch.org/whl/torch_stable.html
!pip install torchtext==0.9.0
```

```
#Used to run this before
# !pip install -U torch==1.8.0 torchtext==0.9.0
# !pip install -U torch==1.8.0+cu111 torchtext==0.9.0+cu111 -f https://download.pytorch.org/whl/torch_stable.html
# Reload environment
# exit()
```

```
!python -m spacy download en_core_web_sm
!python -m spacy download es_core_news_sm
exit()
```

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

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

```

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import spacy
import numpy as np

import random
import math
import time
import json
import pickle
import os

import tensorflow as tf
from tensorflow import keras
import pathlib
```

```
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
```

Next, we'll set the random seed for reproducability.

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

As before, we'll import spaCy and define the German and English tokenizers.

```
spacy_de = spacy.load('es_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

```
def tokenize_de(text):
    """
    Tokenizes German text from a string into a list of strings
    """
    return [tok.text for tok in spacy_de.tokenizer(text)]

def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

When using packed padded sequences, we need to tell PyTorch how long the actual (non-padded) sequences are. Luckily for us, TorchText's `Field` objects allow us to use the `include_lengths` argument, this will cause our `batch.src` to be a tuple. The first element of the tuple is the same as before, a batch of numericalized source sentence as a tensor, and the second element is the non-padded lengths of each source sentence within the batch.

```
SRC = Field(
  # tokenize = tokenize_de,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True,
    include_lengths = True
)

TRG = Field(
    # tokenize = tokenize_en,
    init_token = '<sos>',
    eos_token = '<eos>',
    lower = True
)
```

```
text_file = keras.utils.get_file(
    fname="spa-eng.zip",
    origin="http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip",
    extract=True,
)
text_file = pathlib.Path(text_file).parent / "spa-eng" / "spa.txt"

```

```
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    eng, spa = line.split("\t")
    eng = tokenize_en(eng)
    spa = tokenize_de(spa)
    # spa = "[start] " + spa + " [end]"
    text_pairs.append((eng, spa))

for _ in range(5):
    print(random.choice(text_pairs))

random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples : num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples :]
```

```
(['This', 'is', 'boring', '.'], ['Es', 'aburrido', '.'])
(['Please', 'try', 'to', 'be', 'as', 'brief', 'as', 'possible', '.'], ['Por', 'favor', 'trata', 'de', 'ser', 'lo', 'más', 'breve', '
(['School', 'begins', 'on', 'the', 'April', '8th', '.'], ['El', 'colegio', 'empieza', 'el', 'ocho', 'de', 'abril', '.'])
(['We', "'re", 'out', 'of', 'money', '.'], ['No', 'tenemos', 'más', 'dinero', '.'])
(['This', 'show', 'is', 'too', 'racy', 'for', 'teenagers', '.'], ['Este', 'programa', 'es', 'demasiado', 'picante', 'para', 'los', '
```

```
train_pairs_mod = [{"src":i[1],"trg":i[0]} for i in train_pairs]
val_pairs_mod = [{"src":i[1],"trg":i[0]} for i in val_pairs]
test_pairs_mod = [{"src":i[1],"trg":i[0]} for i in test_pairs]

# import json
# with open('train.json', 'w') as fp:
#     fp.write(
#         '\n'.join(json.dumps(i) for i in train_pairs_mod)
#         )

# with open('test.json', 'w') as fp:
#     fp.write(
#         '\n'.join(json.dumps(i) for i in test_pairs_mod)
#         )

# with open('valid.json', 'w') as fp:
#     fp.write(
#         '\n'.join(json.dumps(i) for i in val_pairs_mod)
#         )
```

We then load the data.

```
# NOTE: this line takes a long time to run on Colab so
# instead we'll load the already tokenized json dataset
# from Github
#
# train_data, valid_data, test_data = Multi30k.splits(exts = ('.es', '.en'),
#                                                     fields = (SRC, TRG))

# fetch from Github repo
# !wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/train.json
# !wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/valid.json
# !wget https://raw.githubusercontent.com/tberg12/cse291spr21/main/assignment1/test.json

# and load to same variables
fields = {'src': ('src', SRC), 'trg': ('trg', TRG)}
train_data, valid_data, test_data = TabularDataset.splits(
    path = '.',
    train = 'train.json',
    validation = 'valid.json',
    test = 'test.json',
    format = 'json',
    fields = fields
)
# train_data, valid_data, test_data = TabularDataset.splits(
#     path = '.',
#     train = train_pairs,
#     validation = val_pairs,
#     test = test_pairs,
#     # format = 'json',
#     fields = fields
# )
```

And build the vocabulary.

```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

Next, we handle the iterators.

One quirk about packed padded sequences is that all elements in the batch need to be sorted by their non-padded lengths in descending order, i.e. the first sentence in the batch needs to be the longest. We use two arguments of the iterator to handle this, `sort_within_batch` which tells the iterator that the contents of the batch need to be sorted, and `sort_key` a function which tells the iterator how to sort the elements in the batch. Here, we sort by the length of the `src` sentence.

```
BATCH_SIZE = 128

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

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
     batch_size = BATCH_SIZE,
     sort_within_batch = True,
     sort_key = lambda x : len(x.src),
     device = device)
```

Building the Model

Encoder

Next up, we define the encoder.

The changes here all within the `forward` method. It now accepts the lengths of the source sentences as well as the sentences themselves.

After the source sentence (padded automatically within the iterator) has been embedded, we can then use `pack_padded_sequence` on it with the lengths of the sentences. Note that the tensor containing the lengths of the sequences must be a CPU tensor as of the latest version of PyTorch, which we explicitly do so with `to('cpu')`. `packed_embedded` will then be our packed padded sequence. This can be then fed to our RNN as normal which will return `packed_outputs`, a packed tensor containing all of the hidden states from the sequence, and `hidden` which is simply the final hidden state from our sequence. `hidden` is a standard tensor and not packed in any way, the only difference is that as the input was a packed sequence, this tensor is from the final **non-padded element** in the sequence.

We then unpack our `packed_outputs` using `pad_packed_sequence` which returns the `outputs` and the lengths of each, which we don't need.

The first dimension of `outputs` is the padded sequence lengths however due to using a packed padded sequence the values of tensors when a padding token was the input will be all zeros.

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_len):

        #src = [src len, batch size]
        #src_len = [batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        #need to explicitly put lengths on cpu!
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, src_len.to('cpu'))

        packed_outputs, hidden = self.rnn(packed_embedded)

        #packed_outputs is a packed sequence containing all hidden states
        #hidden is now from the final non-padded element in the batch

        outputs, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs)

        #outputs is now a non-packed sequence, all hidden states obtained
```

```
        #  when the input is a pad token are all zeros

        #outputs = [src len, batch size, hid dim * num directions]
        #hidden = [n layers * num directions, batch size, hid dim]

        #hidden is stacked [forward_1, backward_1, forward_2, backward_2, ...]
        #outputs are always from the last layer

        #hidden [-2, :, : ] is the last of the forwards RNN
        #hidden [-1, :, : ] is the last of the backwards RNN

        #initial decoder hidden is final hidden state of the forwards and backwards
        #  encoder RNNs fed through a linear layer
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)))

        #outputs = [src len, batch size, enc hid dim * 2]
        #hidden = [batch size, dec hid dim]

        return outputs, hidden
```

Attention

The attention module is where we calculate the attention values over the source sentence.

Previously, we allowed this module to "pay attention" to padding tokens within the source sentence. However, using *masking*, we can force the attention to only be over non-padding elements.

The `forward` method now takes a `mask` input. This is a **[batch size, source sentence length]** tensor that is 1 when the source sentence token is not a padding token, and 0 when it is a padding token. For example, if the source sentence is: ["hello", "how", "are", "you", "?", `<pad>`, `<pad>`], then the mask would be [1, 1, 1, 1, 1, 0, 0].

We apply the mask after the attention has been calculated, but before it has been normalized by the `softmax` function. It is applied using `masked_fill`. This fills the tensor at each element where the first argument (`mask == 0`) is true, with the value given by the second argument (`-1e10`). In other words, it will take the un-normalized attention values, and change the attention values over padded elements to be `-1e10`. As these numbers will be miniscule compared to the other values they will become zero when passed through the `softmax` layer, ensuring no attention is payed to padding tokens in the source sentence.

```
[ ]  ↳ 1 cell hidden
```

Decoder

The decoder only needs a few small changes. It needs to accept a mask over the source sentence and pass this to the attention module. As we want to view the values of attention during inference, we also return the attention tensor.

```
[ ]  ↳ 1 cell hidden
```

Seq2Seq

The overarching seq2seq model also needs a few changes for packed padded sequences, masking and inference.

We need to tell it what the indexes are for the pad token and also pass the source sentence lengths as input to the `forward` method.

We use the pad token index to create the masks, by creating a mask tensor that is 1 wherever the source sentence is not equal to the pad token. This is all done within the `create_mask` function.

The sequence lengths as needed to pass to the encoder to use packed padded sequences.

The attention at each time-step is stored in the `attentions`

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, src_pad_idx, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.device = device

    def create_mask(self, src):
        mask = (src != self.src_pad_idx).permute(1, 0)
        return mask

    def forward(self, src, src_len, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #src_len = [batch size]
```

```
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use teacher forcing 75% of the time

        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #encoder_outputs is all hidden states of the input sequence, back and forwards
        #hidden is the final forward and backward hidden states, passed through a linear layer
        encoder_outputs, hidden = self.encoder(src, src_len)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        mask = self.create_mask(src)

        #mask = [batch size, src len]

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden state, all encoder hidden states
            #  and mask
            #receive output tensor (predictions) and new hidden state
            output, hidden, _ = self.decoder(input, hidden, encoder_outputs, mask)

            #place predictions in a tensor holding predictions for each token
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        return outputs
```

Training the Seq2Seq Model

Next up, initializing the model and placing it on the GPU.

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
ENC_HID_DIM = 512
DEC_HID_DIM = 512
ENC_DROPOUT = 0.6
DEC_DROPOUT = 0.6
SRC_PAD_IDX = SRC.vocab.stoi[SRC.pad_token]

attn = Attention(ENC_HID_DIM, DEC_HID_DIM)
enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM, DEC_HID_DIM, DEC_DROPOUT, attn)

model = Seq2Seq(enc, dec, SRC_PAD_IDX, device).to(device)
```

Then, we initialize the model parameters.

```
def init_weights(m):
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.normal_(param.data, mean=0, std=0.01)
        else:
            nn.init.constant_(param.data, 0)

model.apply(init_weights)
```

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(12644, 256)
```

```
    (rnn): GRU(256, 512, bidirectional=True)
    (fc): Linear(in_features=1024, out_features=512, bias=True)
    (dropout): Dropout(p=0.6, inplace=False)
  )
  (decoder): Decoder(
    (attention): Attention(
      (attn): Linear(in_features=1536, out_features=512, bias=True)
      (v): Linear(in_features=512, out_features=1, bias=False)
    )
    (embedding): Embedding(7651, 256)
    (rnn): GRU(1280, 512)
    (fc_out): Linear(in_features=1792, out_features=7651, bias=True)
    (dropout): Dropout(p=0.6, inplace=False)
  )
)
```

We'll print out the number of trainable parameters in the model, noticing that it has the exact same amount of parameters as the model without these improvements.

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 25,347,043 trainable parameters
```

Then we define our optimizer and criterion.

The `ignore_index` for the criterion needs to be the index of the pad token for the target language, not the source language.

```
optimizer = optim.Adam(model.parameters())
```

```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

Next, we'll define our training and evaluation loops.

As we are using `include_lengths = True` for our source field, `batch.src` is now a tuple with the first element being the numericalized tensor representing the sentence and the second element being the lengths of each sentence within the batch.

Our model also returns the attention vectors over the batch of source source sentences for each decoding time-step. We won't use these during the training/evaluation, but we will later for inference.

```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src, src_len = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, src_len, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

```

```
        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src, src_len = batch.src
            trg = batch.trg

            output = model(src, src_len, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Then, we'll define a useful function for timing how long epochs take.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

The penultimate step is to train our model. Notice how it takes almost half the time as our model without the improvements added in this notebook.

```
N_EPOCHS = 6
CLIP = 1

best_valid_loss = float('inf')

loss_di = {"epoch_num":[],"train_loss":[],"valid_loss":[],"train_perp":[],"valid_perp":[]}

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut4_es-model.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    loss_di["epoch_num"].append(epoch+1)
    loss_di["train_loss"].append(train_loss)
    loss_di["train_perp"].append(math.exp(train_loss))
    loss_di["valid_loss"].append(valid_loss)
    loss_di["valid_perp"].append(math.exp(valid_loss))
```

```
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')

# with open("train_loss.pkl","wb") as f:
#     pickle.dump(loss_di,f)
```

```
Epoch: 01 | Time: 1m 32s
        Train Loss: 4.295 | Train PPL:  73.327
         Val. Loss: 3.547 |  Val. PPL:  34.718
Epoch: 02 | Time: 1m 32s
        Train Loss: 2.657 | Train PPL:  14.259
         Val. Loss: 2.521 |  Val. PPL:  12.439
Epoch: 03 | Time: 1m 32s
        Train Loss: 1.902 | Train PPL:   6.697
         Val. Loss: 2.229 |  Val. PPL:   9.294
Epoch: 04 | Time: 1m 32s
        Train Loss: 1.516 | Train PPL:   4.555
         Val. Loss: 2.108 |  Val. PPL:   8.228
Epoch: 05 | Time: 1m 32s
        Train Loss: 1.265 | Train PPL:   3.544
         Val. Loss: 2.065 |  Val. PPL:   7.886
Epoch: 06 | Time: 1m 32s
        Train Loss: 1.109 | Train PPL:   3.033
         Val. Loss: 2.052 |  Val. PPL:   7.782
```

Finally, we load the parameters from our best validation loss and get our results on the test set.

We get the improved test perplexity whilst almost being twice as fast!

```
model.load_state_dict(torch.load('tut4_es-model.pt'))

test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
| Test Loss: 2.030 | Test PPL:   7.614 |
```

Inference

Now we can use our trained model to generate translations.

Note: these translations will be poor compared to examples shown in paper as they use hidden dimension sizes of 1000 and train for 4 days!
They have been cherry picked in order to show off what attention should look like on a sufficiently sized model.

Our `translate_sentence` will do the following:

- ensure our model is in evaluation mode, which it should always be for inference
- tokenize the source sentence if it has not been tokenized (is a string)
- numericalize the source sentence
- convert it to a tensor and add a batch dimension
- get the length of the source sentence and convert to a tensor
- feed the source sentence into the encoder
- create the mask for the source sentence
- create a list to hold the output sentence, initialized with an `<sos>` token
- create a tensor to hold the attention values
- while we have not hit a maximum length
    - get the input tensor, which should be either `<sos>` or the last predicted token
    - feed the input, all encoder outputs, hidden state and mask into the decoder
    - store attention values
    - get the predicted next token
    - add prediction to current output sentence prediction
    - break if the prediction was an `<eos>` token
- convert the output sentence from indexes to tokens
- return the output sentence (with the `<sos>` token removed) and the attention values over the sequence

```
def translate_sentence(sentence, src_field, trg_field, model, device, max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
```

```
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]


    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)


    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)

    for i in range(max_len):

        trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)

        with torch.no_grad():
            output, hidden, attention = model.decoder(trg_tensor, hidden, encoder_outputs, mask)

        attentions[i] = attention
        pred_token = output.argmax(1).item()

        trg_indexes.append(pred_token)

        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]
    # print("Log_prob:",np.log(output.max().item()))

    return trg_tokens[1:], attentions[:len(trg_tokens)-1]
```

Next, we'll make a function that displays the model's attention over the source sentence for each target token generated.

```
def display_attention(sentence, translation, attention):

    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)

    attention = attention.squeeze(1).cpu().detach().numpy()

    cax = ax.matshow(attention, cmap='bone')

    ax.tick_params(labelsize=15)

    x_ticks = [''] + ['<sos>'] + [t.lower() for t in sentence] + ['<eos>']
    y_ticks = [''] + translation

    ax.set_xticklabels(x_ticks, rotation=45)
    ax.set_yticklabels(y_ticks)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
    plt.close()
```

Now, we'll grab some translations from our dataset and see how well our model did. Note, we're going to cherry pick examples here so it gives us something interesting to look at, but feel free to change the `example_idx` value to look at different examples.

First, we'll get a source and target from our dataset.

```
example_idx = 12

src = vars(train_data.examples[example_idx])['src']
trg = vars(train_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['te', 'amaré', 'por', 'siempre', '.']
trg = ['i', "'ll", 'love', 'you', 'forever', '.']
```

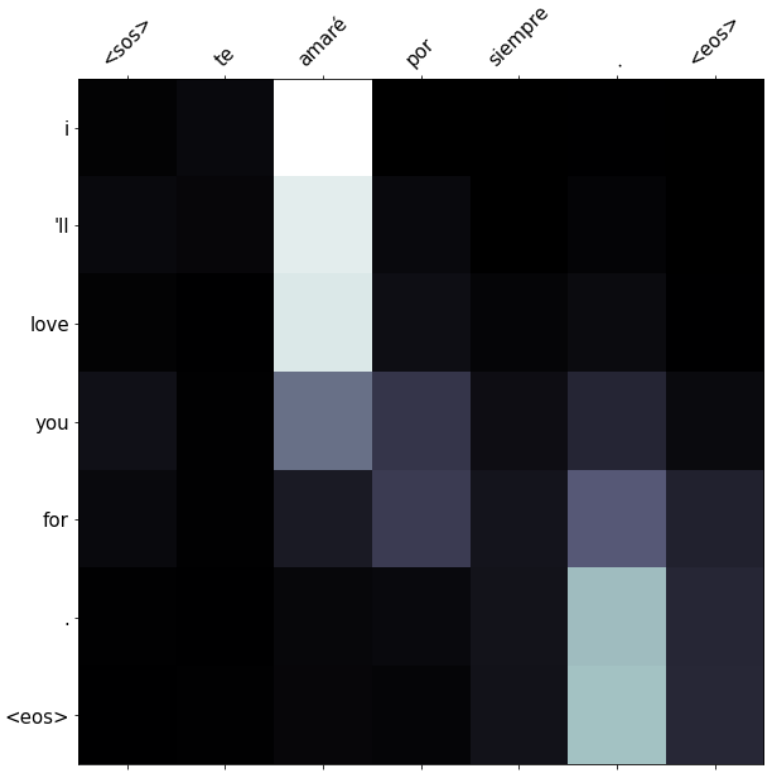
Then we'll use our `translate_sentence` function to get our predicted translation and attention. We show this graphically by having the source sentence on the x-axis and the predicted translation on the y-axis. The lighter the square at the intersection between two words, the more attention the model gave to that source word when translating that target word.

Below is an example the model attempted to translate, it gets the translation correct except changes *are fighting* to just *fighting*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
predicted trg = ['i', "'ll", 'love', 'you', 'for', '.', '<eos>']
```

```
display_attention(src, translation, attention)
```



Translations from the training set could simply be memorized by the model. So it's only fair we look at translations from the validation and testing set too.

Starting with the validation set, let's get an example.

```
example_idx = 14

src = vars(valid_data.examples[example_idx])['src']
trg = vars(valid_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

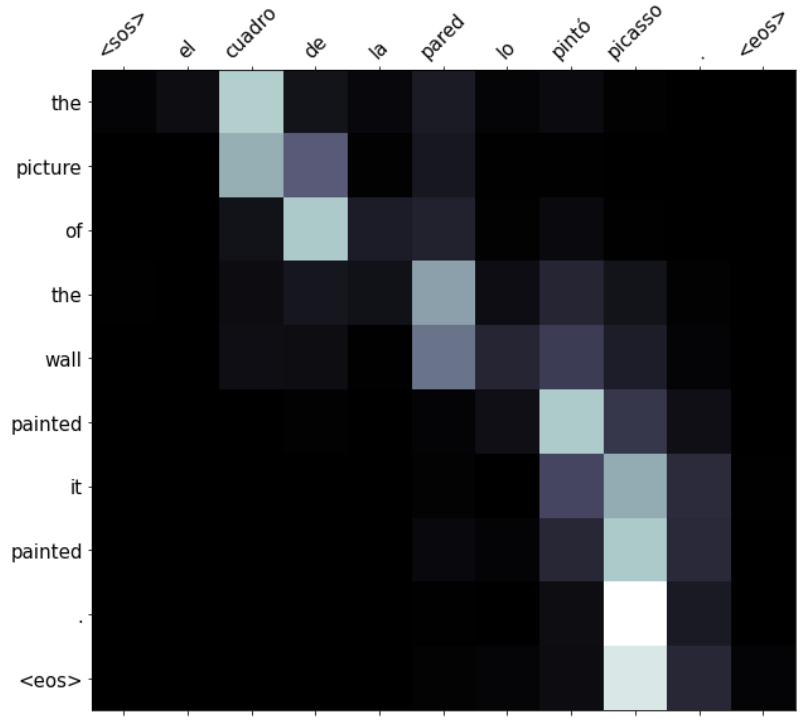
```
src = ['el', 'cuadro', 'de', 'la', 'pared', 'lo', 'pintó', 'picasso', '.']
trg = ['the', 'picture', 'on', 'the', 'wall', 'was', 'painted', 'by', 'picasso', '.']
```

Then let's generate our translation and view the attention.

Here, we can see the translation is the same except for swapping *female* with *woman*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
predicted trg = ['the', 'picture', 'of', 'the', 'wall', 'painted', 'it', 'painted', '.', '<eos>']
```



Finally, let's get an example from the test set.

```
example_idx = 18

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['¿', 'por', 'qué', 'tom', 'querría', 'lastimar', 'a', 'maría', '?']
trg = ['why', 'would', 'tom', 'want', 'to', 'hurt', 'mary', '?']
```

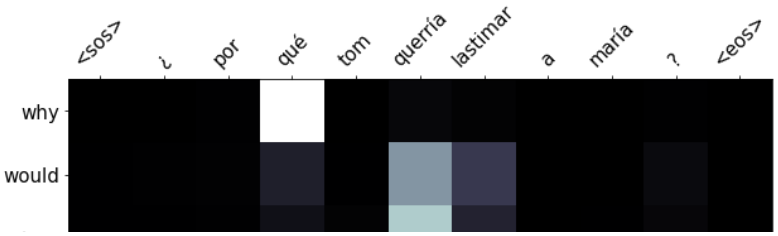
```
print(' '.join(trg))
```

```
why would tom want to hurt mary ?
```

Again, it produces a slightly different translation than target, a more literal version of the source sentence. It swaps *mountain climbing* for *climbing a mountain*.

```
translation, attention = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')

display_attention(src, translation, attention)
```

```
predicted trg = ['why', 'would', 'tom', 'want', 'to', 'hurt', 'mary', '?', '<eos>']
```



BLEU

Previously we have only cared about the loss/perplexity of the model. However there metrics that are specifically designed for measuring the quality of a translation - the most popular is *BLEU*. Without going into too much detail, BLEU looks at the overlap in the predicted and actual target sequences in terms of their n-grams. It will give us a number between 0 and 1 for each sequence, where 1 means there is perfect overlap, i.e. a perfect translation, although is usually shown between 0 and 100. BLEU was designed for multiple candidate translations per source sequence, however in this dataset we only have one candidate per source.

We define a `calculate_bleu` function which calculates the BLEU score over a provided TorchText dataset. This function creates a corpus of the actual and predicted translation for each source sentence and then calculates the BLEU score.

```
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = translate_sentence(src, src_field, trg_field, model, device, max_len)

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)

def calculate_bleu_beam(data, src_field, trg_field, model, device,beam_size = 5, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in data:

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg, _ = beam_search_decoder(src, src_field, trg_field, model, device,beam_size, max_len)

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return bleu_score(pred_trgs, trgs)
```

We get a BLEU of around 28. If we compare it to the paper that the attention model is attempting to replicate, they achieve a BLEU score of 26.75. This is similar to our score, however they are using a completely different dataset and their model size is much larger - 1000 hidden dimensions which takes 4 days to train! - so we cannot really compare against that either.

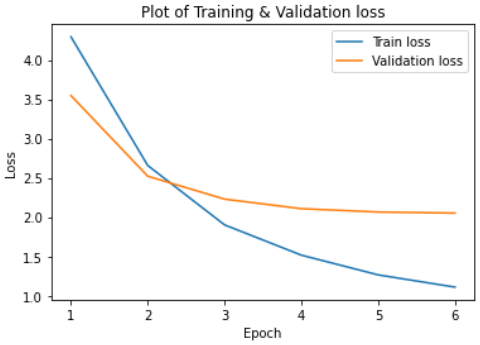
This number isn't really interpretable, we can't really say much about it. The most useful part of a BLEU score is that it can be used to compare different models on the same dataset, where the one with the **higher** BLEU score is "better".

```
bleu_score_greedy = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score_greedy*100:.2f}')
```

```
BLEU score = 43.03
```

In the next tutorials we will be moving away from using recurrent neural networks and start looking at other ways to construct sequence-to-sequence models. Specifically, in the next tutorial we will be using convolutional neural networks.

```
# with open("train_loss.pkl","rb") as f:
#     loss_di = pickle.load(f)

import matplotlib.pyplot as plt

plt.plot(loss_di['epoch_num'],loss_di['train_loss'],label = "Train loss")
plt.plot(loss_di['epoch_num'],loss_di['valid_loss'],label = "Validation loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Plot of Training & Validation loss")
plt.legend()
plt.show()
```




```
#Beam search implementation function

def beam_search_decoder(sentence, src_field, trg_field, model, device, beam_size = 5,max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('de')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]

    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    src_len = torch.LongTensor([len(src_indexes)])

    #obtaining encoder output once (outside of beam_search and re-used later everytime)
    with torch.no_grad():
        encoder_outputs, hidden_inp_dec = model.encoder(src_tensor, src_len)

    mask = model.create_mask(src_tensor)

    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    attentions = torch.zeros(max_len, 1, len(src_indexes)).to(device)


    #Sequences hold the log probs, decoders hidden state and current decoded sequence
    sequences = [[0.0,hidden_inp_dec,trg_indexes]] #0.0, hidden, [4]

    for i in range(max_len):
      all_candidates = [] #to capture the new beams

      for beam in range(len(sequences)):
        current_beam = sequences[beam]
        beam_index = current_beam[2]
        beam_hidden = current_beam[1]

        trg_tensor = torch.LongTensor([beam_index[-1]]).to(device)
        if trg_field.vocab.stoi[trg_field.eos_token] in beam_index: #if end of sentence token is encountered prune that branch
          all_candidates.append(current_beam)
          continue
```

```

        with torch.no_grad():
          output, hidden, attention = model.decoder(trg_tensor, beam_hidden, encoder_outputs, mask)


        attentions[i] = attention
        # output_new = (output - output.min()) / (output.max() - output.min())
        # output_new = output_new + 0.00000001
        # log_prob_vocab = torch.log(output_new)
        log_prob_vocab = F.log_softmax(output)
        log_prob_vocab = current_beam[0] + log_prob_vocab
        # print("After:",log_prob_vocab[0][0])
        log_probs_k, indices_k = torch.topk(log_prob_vocab, beam_size)

        for i in range(len(log_probs_k[0])):
          temp = [log_probs_k[0][i].item(),hidden,beam_index + [indices_k[0][i].item()]]
          # print(temp)
          all_candidates.append(temp)

      sequences = sorted(all_candidates,key = lambda x: x[0],reverse = True)[:beam_size]

    trg_tokens_list = []
    for i in range(len(sequences)):
      trg_tokens = [trg_field.vocab.itos[i] for i in sequences[i][2]]
      trg_tokens_list.append(trg_tokens[1:])
      # print("Prob_value:",sequences[i][0],trg_tokens[1:])

    # return trg_tokens_list,attentions[:len(trg_tokens)-1]
    trg_tokens_highest = [trg_field.vocab.itos[i] for i in sequences[0][2]]
    # print("Target_toks:",trg_tokens[1:])
    return trg_tokens_highest[1:], attentions[:len(trg_tokens_highest)-1]



```

```
#Eg 1

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
# for i in translation:
#   print(f'predicted trg = {i}')
# beam_search_decoder(src, SRC, TRG, model, device,3,50)
```

```
Prob_value: -1.0082758665084839 ['why', 'would', 'tom', 'want', 'to', 'hurt', 'mary', '?', '<eos>']
Prob_value: -1.8484323024749756 ['why', 'would', 'tom', 'want', 'to', 'kill', 'mary', '?', '<eos>']
Prob_value: -4.083009719848633 ['why', 'would', 'tom', 'want', 'to', 'hurt', '?', '?', '<eos>']
Prob_value: -4.154170036315918 ['why', 'would', 'tom', 'want', 'to', 'wait', '?', '<eos>']
Prob_value: -4.232594966888428 ['why', 'would', 'tom', 'want', 'to', 'drive', 'mary', '?', '<eos>']
predicted trg = ['why', 'would', 'tom', 'want', 'to', 'hurt', 'mary', '?', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Eg 2

example_idx = 20

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

```
src = ['para', 'mí', 'es', 'difícil', 'levantarme', 'antes', 'de', 'las', 'seis', '.']
trg = ['it', 'is', 'difficult', 'for', 'me', 'to', 'get', 'up', 'before', 'six', '.']
predicted trg = ['it', "'s", 'difficult', 'for', 'me', 'to', 'get', 'up', 'to', 'six', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Eg 3

example_idx = 26

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)
```

```
print(f'predicted trg = {translation}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

```
src = ['este', 'sombrero', 'es', 'tuyo', '.']
trg = ['this', 'hat', 'is', 'yours', '.']
predicted trg = ['this', 'hat', 'is', 'yours', '.', '<eos>']
Prob_value: -0.1334570199251175 ['this', 'hat', 'is', 'yours', '.', '<eos>']
Prob_value: -3.1929967403411865 ['this', 'hat', 'is', 'mine', '.', '<eos>']
Prob_value: -4.146904468536377 ['this', 'hat', "'s", 'yours', '.', '<eos>']
Prob_value: -4.989144325256348 ['this', 'hat', 'is', 'yours', '.', '.', '<eos>']
Prob_value: -5.349325656890869 ['that', 'hat', 'is', 'yours', '.', '<eos>']
predicted trg = ['this', 'hat', 'is', 'yours', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
#Example 4
example_idx = 125

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')

translation, attention = translate_sentence(src, SRC, TRG, model, device)
print(f'predicted trg = {translation}')

translation, attention = beam_search_decoder(src, SRC, TRG, model, device,5)
print(f'predicted trg = {translation}')
```

```
src = ['tengo', 'tiempo', '.']
trg = ['i', 'have', 'time', '.']
predicted trg = ['i', 'have', 'time', '.', '<eos>']
Prob_value: -1.0723193883895874 ['i', 'have', 'time', '.', '<eos>']
Prob_value: -1.7950924634933472 ['i', 'have', 'time', 'time', '.', '<eos>']
Prob_value: -1.8896795511245728 ['i', "'ve", 'got', 'time', '.', '<eos>']
Prob_value: -3.16991925239563 ['i', "'ve", 'time', '.', '<eos>']
Prob_value: -3.303312063217163 ['i', "'ve", 'time', 'time', '.', '<eos>']
predicted trg = ['i', 'have', 'time', '.', '<eos>']
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:56: UserWarning: Implicit dimension choice for log_softmax has been dep
```

```
bleu_score_b = calculate_bleu_beam(test_data, SRC, TRG, model, device,5)

print(f'BLEU score = {bleu_score_b*100:.2f}')
```

```
BLEU score = 44.82
```