Assignment 4

Fabian Gobet, gobetf@usi.ch

January 14, 2024

Conversational model with Transformers

1 Data (40 pts)

1. To check the content of the files i created a function named check_content_txt_files() that prints out the first 5 elements of each file.

```python
def check_content_txt_files(convo_path, lines_path, num_elements):
    with open(convo_path, 'r') as conv_file:
        for i in range(num_elements):
            print(conv_file.readline())
    with open(lines_path, 'r') as lines_file:
        for i in range(num_elements):
            print(lines_file.readline())
```

2. For this part of the exercise I chose to consider all possible pairs as to enrich the dataset with more questions and answers. Because many of these sentences will be repeated in pairs, I considered pairs of references instead, since these tend to be smaller in string length. This will also simplify the construction of strings for the dataset further down the road. Hence, I implemented the function get_reference_pairs() to achieve this.

```python
def get_reference_pairs(convo_path):
    ref_pairs = []
    with open(convo_path, 'r') as conv_file:
        for line in conv_file:
            conversation = line.strip().split(' +++$+++ ')
                [-1][1:-1].replace("'", "").split(",")
            for i in range(len(conversation) - 1):
                ref_pairs.append((conversation[i].strip(),
                    conversation[i+1].strip()))
    return ref_pairs
```

3. Next, to set the sentences in a format congruent to our needs I created the functions normalize_sentences() and process_sentence(). The former function splits the line, processes the utterance and filters it to either consider it or count as a non considered string. The latter function does some pre-process, removing some html tags that I found in the lines, unrolling grammatical shortcuts to their full extent equivalent, removing any symbol that is not a letter, number or acceptable punctuation, stripping extra spaces and lowering the case of the string. If the resulting string still has content, then the TreebankWordTokenizer is used to tokenize the sentence.

```python
def normalize_sentences(path_lines):
    lines_dict = {}
    empty_lines = 0
    with open(path_lines, 'r', encoding='cp1252') as lines_file:
        for full_line in lines_file:
            line_split = full_line.split(' +++$+++ ')
            line = line_split[-1]
            line, is_line_empty = process_sentence(line)
            if is_line_empty == 1:
                empty_lines += 1
            else:
                lines_dict.update({line_split[0] : line})
    return lines_dict, empty_lines
```

```python
def process_sentence(line):
    is_line = 0
    line = line.replace('\n','').replace("<u>","").replace("</u>",
        "").lower()
    line = re.sub('-+','',line)
    line = re.sub(r"\'s", " is", line, flags=re.I)
    line = re.sub(r"can't", "can not", line, flags=re.I)
    line = re.sub(r"\'bout", "about", line, flags=re.I)
    line = re.sub(r"\'til", "until", line, flags=re.I)
    line = re.sub(r"\'till", "until", line, flags=re.I)
    line = re.sub(r"\'m", " am", line, flags=re.I)
    line = re.sub(r"\'cause", "because", line, flags=re.I)
    line = re.sub(r"\'ll", " will", line, flags=re.I)
    line = re.sub(r"\'em", " them", line, flags=re.I)
    line = re.sub(r"\'ve", " have", line, flags=re.I)
    line = re.sub(r"\'re", " are", line, flags=re.I)
    line = re.sub(r"\'d", " would", line, flags=re.I)
    line = re.sub(r"n't", " not", line, flags=re.I)
    line = re.sub(r"[^a-zA-Z.?,!]+", r" ", line)
    line = line.strip()
    if line == '' or line.isspace():
        is_line += 1
    else:
        line = TreebankWordTokenizer().tokenize(line)
    return line, is_line
```

After this process, a function named generate_primitive_valid_pairs() is used to generate both the pair of references that are possible from the remaining sentences and a dictionary that maps the reference to the sentence with the added ['¡EOS¿'] token at the end. Notice that all sentences have these tokens but only answers have the ['¡SOS¿'] token, Therefore, to take advatnage of the non repetition we can simply add this token before converting the sentence into the vocabulary form inside the dataset.

```python
def generate_primitive_valid_pairs(all_ref_pairs, lines):
    chosen_sentences = {}
    chosen_ref_pairs = []
    for p in all_ref_pairs:
        if p[0] in lines and p[1] in lines:
            chosen_sentences.update({p[0]: lines[p[0]]+['<EOS>'],
                p[1]: lines[p[1]]+['<EOS>']})
            chosen_ref_pairs.append(p)
    return chosen_sentences, chosen_ref_pairs
```

Every now and then I check whether my results are optimal by printing out a few random elements from my processed data. To achieve this I implemented the print_random_elements() function.

```python
def print_random_elements(collection, k=5):
    random_elements = random.sample(collection, k=k)
    for e in random_elements:
        print(e)
```

4. To decide on the sentence maximum length I used the statistics library to check for the mean and standard deviation of the sentences lengths using the follow code.

```
    sentence_lengths = []
    for p in chosen_ref_pairs:
        sentence_lengths.append(len(chosen_sentences[p[0]]))
        sentence_lengths.append(len(chosen_sentences[p[1]]))
    mean_length = statistics.mean(sentence_lengths)
    std_dev = statistics.stdev(sentence_lengths)
    print('Mean sentence length: {}'.format(mean_length))
    print('Standard deviation: {}'.format(std_dev))
    print('Max sentence length: {}'.format(max(sentence_lengths)))
    print('Min sentence length: {}'.format(min(sentence_lengths)))
    plt.hist(sentence_lengths, density=True, bins=40)
    plt.xlabel('Sentence Length')
    plt.ylabel('Frequency (log scale)')
    plt.title('Sentence Length Distribution')
    plt.yscale('log')
    plt.axvline(x=mean_length, color='r', linestyle='--', label='
        Mean')
    plt.legend()
    plt.show()
    length_counts = {}
    for length in sentence_lengths:
        if length in length_counts:
            length_counts[length] += 1
        else:
            length_counts[length] = 1
    sorted_lengths = sorted(length_counts.keys())
    total_sentences = len(sentence_lengths)
    accumulated_percentage = 0
    percentage_values = []
    for length in sorted_lengths:
        frequency = length_counts[length]
        percentage = (frequency / total_sentences) * 100
        accumulated_percentage += percentage
        percentage_values.append(accumulated_percentage)
    plt.plot(sorted_lengths, percentage_values)
    plt.xlabel('Sentence Length')
    plt.ylabel('Accumulated Frequency Percentage')
    plt.title('Sentence Length Distribution')
    plt.show()
initial_num_sentences = len(chosen_sentences)
initial_num_pairs = len(chosen_ref_pairs)
chosen_sentences2,chosen_ref_pairs2,rule_out_sentences_refs =
    eliminate_long_sentences(chosen_sentences, chosen_ref_pairs,
    max_length)
```

The results showed a mean of 12.8 and a standard deviation of 13.33. Furthermore, we can see in the following plots that more than 80% of the sentences have a length under 26. Therefore, the chosen maximum length was 26.

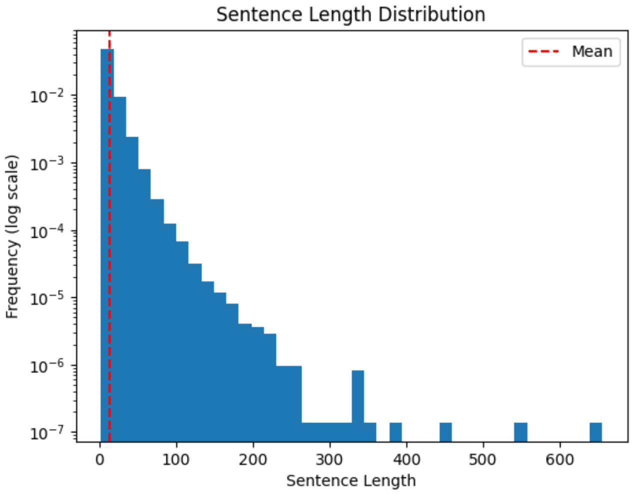

Figure 1: Sentence Length Histogram Distribution

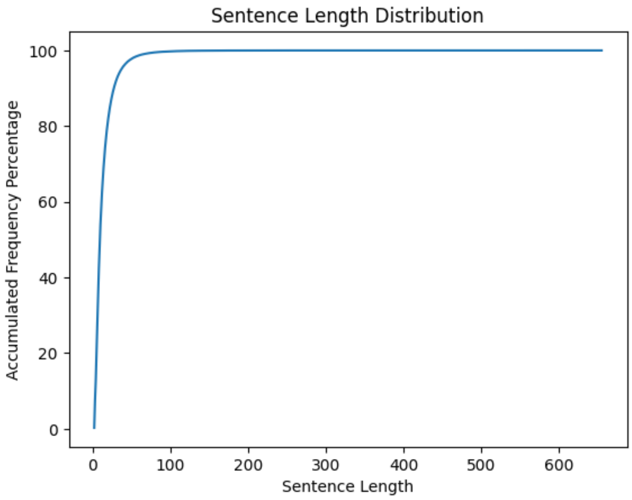

Figure 2: Sentence Length Accumulated Frequency Distribution

After having this information, I proceeded to implement a function that eliminates all sentences that are longer than our chosen max length from our sentences and references.

```
def eliminate_long_sentences(chosen_sentences, chosen_ref_pairs,
    max_length):
    rule_out_sentences_refs = set()
    chosen_sentences2 = {}
    chosen_ref_pairs2 = []
    for k,v in chosen_sentences.items():
        if len(v) > max_length:
            rule_out_sentences_refs.add(k)
    for p in chosen_ref_pairs:
```

```
        if p[0] not in rule_out_sentences_refs and p[1] not in
            rule_out_sentences_refs:
            chosen_sentences2.update({p[0]: chosen_sentences[p
                [0]]})
            chosen_sentences2.update({p[1]: chosen_sentences[p
                [1]]})
            chosen_ref_pairs2.append(p)
    return chosen_sentences2, chosen_ref_pairs2,
        rule_out_sentences_refs
```

5. In order to save and load files in pickle format. Thw following function were implemented

```
def pickle_dump(obj, PATH, name):
    if not os.path.exists(PATH):
        os.makedirs(PATH)
    with open(PATH+name, 'wb') as f:
        pickle.dump(obj, f)
```

```
def pickle_load(PATH):
    with open(PATH, 'rb') as f:
        obj = pickle.load(f)
    return obj
```

6. To determine what words I want to keep a similar analysis to the sentence length is in order. As such, The function count$_{w}ords()computesboththetotalnumberofwordsandthefre$

Having done this, I compute the mean and some plots to see whats happening.

```
print('Number of words: {}'.format(num_words))
print('Number of unique words: {}'.format(len(word_counts)))
plt.plot(range(len(word_counts)), list(word_counts.values()))
plt.xlabel('Word index')
plt.ylabel('Word frequency')
plt.suptitle('Word frequency distribution')
plt.title('Frequency per word')
plt.show()
# Compute the mean and standard deviation for word counts
mean_value = statistics.mean(word_counts.values())
print("Mean value:", mean_value)
print("Max value:", max(word_counts.values()))
print("Min value:", min(word_counts.values()))
# Plot the distribution of word counts < mean
dict_lower_mean = {k: v for k, v in word_counts.items() if v <
    mean_value}
sorted_dict_lower_mean = dict(sorted(dict_lower_mean.items(), key=
    lambda x: x[1], reverse=True))
plt.plot(range(len(sorted_dict_lower_mean)),
    sorted_dict_lower_mean.values())
plt.xlabel('Word index')
plt.ylabel('Word count')
plt.suptitle('Word count distribution')
plt.title('Words with count < mean')
plt.show()
```

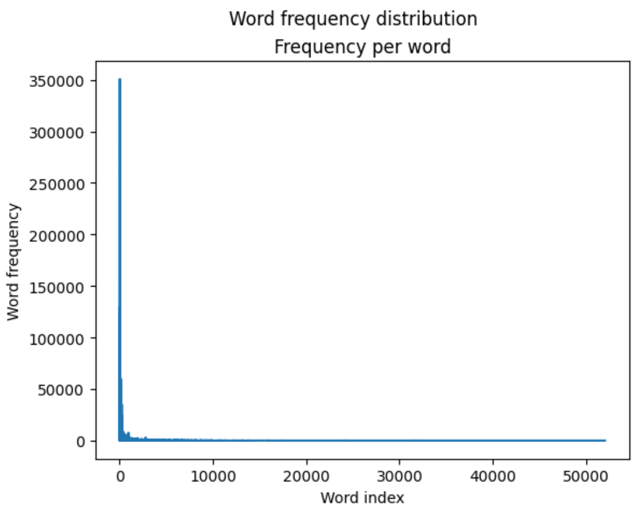

Figure 3: Word frequency distribution

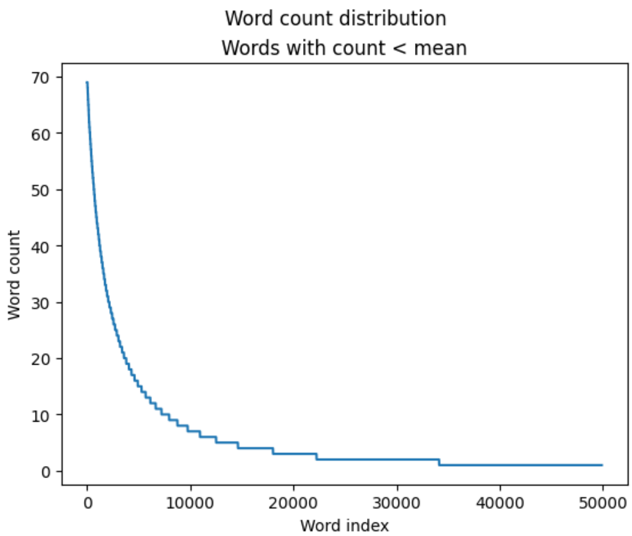

Figure 4: Word frequency distribution under mean

We can see from the plots that for words with frequency above 10 we can still retain a rich vocabulary. Hence the chosen value was 10. Like in the sentences length part, the function eliminate_sentences_with_rare_words() filters out all references and sentences that cointain rare words.

```
rule_out_words = [k for k, v in word_counts.items() if v <
    word_frequency_discard]
chosen_sentences3, chosen_ref_pairs3, rule_out_sentences_refs =
    eliminate_sentences_with_rare_words(chosen_sentences2,
    chosen_ref_pairs2, rule_out_words)
```

7. After all this process, both the sentences and reference pairs are saved into a pickle file.

```
pickle_dump(chosen_sentences3,path,savename+"_sentences.pkl")
pickle_dump(chosen_ref_pairs3,path,savename+"_ref_pairs.pkl")
```

8. The final number of sentences is 139111 (45.71% of total) and of pairs is 113547 (51.31% of total). After experimenting for a while, a sample of 40960 seemed to provide a dataset with a vast vocabulary and prevented some over-fitting when training. Furthermore, 40960 is a number that can generate a a train/validation/test split with powers of 2. In order to achieve reproducibility, once the splits were done I saved all reference pairs and sentences into a pickle file.

```
rand_sample_num = 40960
rand_sample_num = min(rand_sample_num,len(chosen_ref_pairs))
chosen_ref_pairs = random.sample(chosen_ref_pairs,rand_sample_num)
all_refs = set()
for p in chosen_ref_pairs:
  all_refs.add(p[0])
  all_refs.add(p[1])
chosen_sentences = {k:v for k,v in chosen_sentences.items() if k
    in all_refs}
train_size = (int(0.8 * rand_sample_num)//batch_size)*batch_size
val_size = (rand_sample_num-train_size)//2
test_size = rand_sample_num-train_size-val_size
train_pairs, val_pairs, test_pairs = random_split(chosen_ref_pairs
    ,[train_size,val_size,test_size])
PATH = './'
pickle_dump(chosen_sentences, PATH,'chosen_sentences_'+str(
    rand_sample_num)+'.pkl')
pickle_dump(train_pairs, PATH,'train_pairs_'+str(rand_sample_num)+
    '.pkl')
pickle_dump(val_pairs, PATH,'val_pairs_'+str(rand_sample_num)+'.
    pkl')
pickle_dump(test_pairs, PATH,'test_pairs_'+str(rand_sample_num)+'.
    pkl')
```

9. Since the vocabulary is indexed at 0, every new word will occupy the index at the current length.

```
...
def add_word(self, word):
if not word in self.word2index:
    self.word2index[word] = len(self.word2index)
    self.index2word[len(self.index2word)] = word
def add_sentence(self, sentence):
    for word in sentence:
        self.add_word(word)
```

10. It was mentioned before that the SOS token would be added inside the dataset for simplicity of the implementation. Therefore, generating the tensors to return, the token is added.

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, vocabulary, pairs_refs, sentences):
        self.vocabulary = vocabulary
        self.pairs = pairs_refs
        self.sentences = sentences
    def __len__(self):
        return len(self.pairs)
    def __getitem__(self, ix):
        q,a = self.pairs[ix]
        q = torch.tensor([self.vocabulary.word2index[word] for
            word in self.sentences[q]])
        a = torch.tensor([self.vocabulary.word2index[word] for
            word in ["<SOS>"]+self.sentences[a]])
        return q,a
```

As final regards, all the above code is put into a convenient function that generates the sentences and pairs given a set of parameters, whilst also admitting a flag called verbose if we also need to generate the plots and statistics again

```
def create_pairs(path='./MyFiles/',savename="result",max_length=26,
    word_frequency_discard=10,verbose=True):
    # All of the aboce explained code goes here !
    ...
    return chosen_sentences3,chosen_ref_pairs3
```

2 Model & Tools for training (35 pts)

1. This implementation of PositionalEncoding in PyTorch adds sinusoidal positional encodings to input embeddings, a technique commonly used in transformers. The positional encodings are precomputed and stored in a buffer pe, allowing for efficient retrieval. It's a straightforward and efficient implementation, adhering closely to the original Transformer model described in "Attention Is All You Need".

2. The embedding module maps elements from the vocabulary space to the model dimensional space. The positional encoding then maps elements within the same model dimensional space, adding the positional encoding to the embedded vectors. Then, the transformer layer produces the functionalities of the encoder and decoder parts in a same fashion as "Attention is All you Need", and finally the linear layer maps the produced vectors from the model dimensional space to the vocabulary. By setting the batch_first flag to true we can input the batch in the first dimension and the returnng output will also have the batch in the first dimension.

```
self.embedding = nn.Embedding(vocab_size, d_model)
self.pos_encoder = PositionalEncoding(d_model,dropout=
    dropout_posenconding)
self.transformer = nn.Transformer(d_model=d_model, nhead=num_heads
    , num_encoder_layers=encoder_layers,
                                    num_decoder_layers=
                                        decoder_layers,
                                        dim_feedforward=
                                        dim_feedforward, dropout=
                                        dropout_transformer,
                                    batch_first=True)
self.linear = nn.Linear(d_model, vocab_size)
```

3. in this function we just have to generate a bool tensor that has True for positions with padding and false otherwise

```
def create_padding_mask(self, x, pad_id=0):
    return x.eq(pad_id)
```

4. This exercise was misleading because there were no TODO flags inside the forward method. Regarding the masks, the tgt_mask prevents the decoder of the model from peeking at future tokens in the target sequence during training (with the self attention mechanism; the src_pad_masks serves as a filter for the pad tokens in the encoder attention mechanism, similarly the tgt_pad_mask serves the same purpose the in the decoder part; the memory_key_padding_masks is applied to the encoders output when in the decoder part.

5. They serve the same purpose, True equates to 0 whereas False equates to -inf.

3 Training (35 pts)

1. For the training I first defined the following initial hyperparameters, which were tuned after a few tries. Even with higher model dimensionality, more encoder and decoder layers and bigger feed forward layer, the model still performed the same but at a slower pace, some times showing more signs of over-fitting without the potential to get better training scores.

```
device = torch.device("mps" if torch.mps.is_available() else "cpu"
    )
batch_size = 128
rand_sample_num = 40960
learning_rate = 1e-4
d_model = 256
encoder_layers = 4
decoder_layers = 4
feed_forward_dim = 1024
nheads = 8
dropout_transformer = 0.4
dropout_posenconding = 0.1
patience = 3
epochs = 10
eval_every_n_batches = 32
```

To generate the datasets I also had to define a function for the collation of the sentences and a method that extracts the sentences from the chosen sentences given all reference pairs.

```
def collate_fn(batch,pad_idx):
  data, targets = zip(*batch)
  padded_data = nn.utils.rnn.pad_sequence(data, batch_first=True,
      padding_value=pad_idx)
  padded_targets = nn.utils.rnn.pad_sequence(targets, batch_first=
      True, padding_value=pad_idx)
  return padded_data, padded_targets
```

```
def extract_sentences_from_refs(chosen_ref_pairs, chosen_sentences
    ):
    chosen_sentences2 = {}
    for p in chosen_ref_pairs:
        chosen_sentences2.update({p[0]: chosen_sentences[p[0]]})
        chosen_sentences2.update({p[1]: chosen_sentences[p[1]]})
    return chosen_sentences2
```

Then, I defined my vocabulary and all the needed datasets from the previous random generated sampling for training, validation and testing (80%, 10%, 10%).

```
vocab = Vocabulary("English",chosen_sentences.values())
train_dataset = Dataset(vocab,train_pairs,
    extract_sentences_from_refs(train_pairs, chosen_sentences))
train_loader = DataLoader(train_dataset,batch_size=batch_size,
    shuffle=True,collate_fn=lambda batch: collate_fn(batch,vocab.
    word2index["<PAD>"]))
val_dataset = Dataset(vocab,val_pairs,extract_sentences_from_refs(
    val_pairs, chosen_sentences))
val_loader = DataLoader(val_dataset,batch_size=batch_size,shuffle=
    False,collate_fn=lambda batch: collate_fn(batch,vocab.
    word2index["<PAD>"]))
test_dataset = Dataset(vocab,test_pairs,
    extract_sentences_from_refs(test_pairs, chosen_sentences))
test_loader = DataLoader(test_dataset,batch_size=batch_size,
    shuffle=False,collate_fn=lambda batch: collate_fn(batch,vocab.
    word2index["<PAD>"]))
```

Next, because both training method make use of an evaluation part, I defined a function that achieves this, returning the calculated loss value

```
def evaluate(val_loader, model, criterion, device):
    model.eval()
    total_loss = 0
    with torch.no_grad():
        for data, targets in val_loader:
            data = data.to(device)
            targets_trim = targets[:,:-1].contiguous().to(device
                )
            outputs = model(data,targets_trim)
            targets_trim = targets[:,1:].contiguous().to(device)
                .view(-1 )
            loss = criterion(outputs.view(-1,outputs.size(-1)),
                targets_trim)
            total_loss += loss.item()
    model.train()
    return total_loss / len(val_loader)
```

And finally, I defined a function to do the actual training. This function takes in all necessary parameters for a standard training, printing out a few sentences during training and returns the lists of losses, the model and the optimizer

```
def train(epochs, model, criterion, optimizer, train_loader,
    device, val_loader, lr_scheduler, eval_every_n_batches):
    train_losses = []
    val_losses = []
    vocab = train_loader.dataset.vocabulary
    optimizer.zero_grad()
    running_loss = 0
    for epoch in range(epochs):
        steps=0
        for i, (data, targets) in enumerate(train_loader):
            steps += 1
            model.train()
            data = data.to(device)
            targets_trim = targets[:,:-1].contiguous().to(device)
            outputs = model(data,targets_trim)
            targets_trim = targets[:,1:].contiguous().to(device)
            loss = criterion(outputs.view(outputs.size(0)*outputs.
                size(1),outputs.size(2)),targets_trim.view(
                targets_trim.size(0)*targets_trim.size(1)))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
            if (i+1) % eval_every_n_batches == 0 or (i+1) == len(
                train_loader):
                train_losses.append(running_loss/steps)
                running_loss = 0
                steps = 0
                val_losses.append(evaluate(val_loader, model,
                    criterion, device))
                if lr_scheduler:
                    lr_scheduler.step(val_losses[-1])
                random_index = random.randint(0, len(targets)-1)
                target_sentence = " ".join([vocab.index2word[idx.
                    item()] for idx in targets[random_index][1:]])
                output_sentence = " ".join([vocab.index2word[idx.
                    item()] for idx in outputs.argmax(dim=-1)[
                    random_index]])
                print(f"Epoch: {epoch+1}/{epochs}, Batch: {i+1}/{
                    len(train_loader)}")
                print(f"Train Loss: {train_losses[-1]:.4f}")
                print(f"Validation Loss: {val_losses[-1]:.4f}")
                print(f"Random Target Sentence: {target_sentence}"
                    )
                print(f"Random Output Sentence: {output_sentence}\
                    n")
            optimizer.zero_grad()
```

```
    return train_losses , val_losses , model, optimizer
```

We are now in position of defining the model, the criterion, the optimizer and learning rate scheduler. For this exercise I used the cross entropy loss function, the Adam optimizer and the ReduceOnPlateau scheduler. The cross entropy choice is a standard choice for classification problems, whereas the choice of the optimizer came after researching about the choice of optimizers that do well on transformers. The scheduler was chosen so as to further decrease the loss function when the learning rate is still high and the losses start bouncing back and forth between the same values at the end of training. A comment on the ignore_index flag and loss values will be made after.

```
model = TransformerModel(vocab_size=len(vocab.word2index),d_model=
    d_model,pad_id=vocab.word2index["<PAD>"],encoder_layers=
    encoder_layers,decoder_layers=decoder_layers,dim_feedforward=
    feed_forward_dim,num_heads=nheads,dropout_transformer=
    dropout_transformer,dropout_posenconding=dropout_posenconding)
    .to(device)
criterion = nn.CrossEntropyLoss(ignore_index=vocab.word2index["<
    PAD>"])
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,'min',patience=patience, factor=0.8)
```

After running training, a plot is done to verify the losses.

```
train_losses , val_losses , model, optimizer = train(epochs, model,
    criterion , optimizer , train_loader , device , val_loader ,
    lr_scheduler , eval_every_n_batches)
# Plot train losses and validation losses
plt.plot(np.linspace(1,epochs,len(train_losses)),train_losses ,
    label='Train Loss')
plt.plot(np.linspace(1,epochs,len(val_losses)),val_losses , label='
    Validation Loss')
plt.xlabel('epochs')
plt.ylabel('Loss')
plt.title('Train Loss vs Validation Loss')
plt.legend()
plt.show()
```

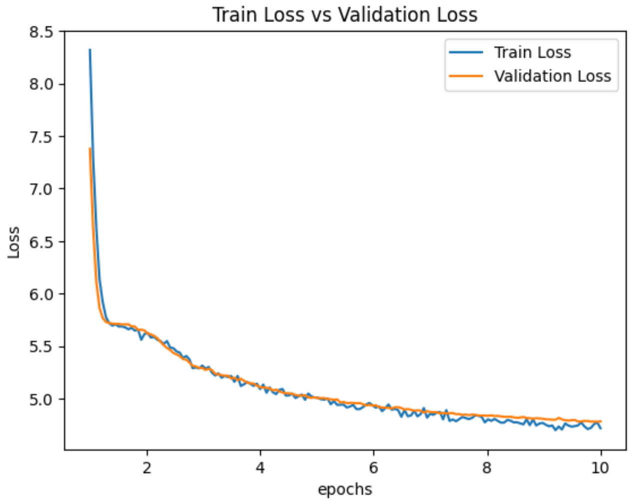

Figure 5: standard train, 10 epochs

On a first note I would like to discuss the actual value for the loss. We can see that this value isn't near the 1.5 threshold. If we were to remove the ignore_index flag from the criterion then we would be evaluating the pads as part of the total loss. Because of the masks, pad indexes are always print in the end of predicted sentences, which are then congruent with the target hence minimizing the loss. But that reduction in loss is merely illusive and doesn't translate into model performance. If the flag were to be removed, a 1.6 loss value could be achieved without any real performance gain, maybe even worse. Hence I chose to use it, even though the loss value is far from optimal.

We can see by analizing the plot that a tendency to over-fit could be arising at the very end, and it does in fact arise if we choose to continue training. On a further notice, training pretty much stagnates at around 4.5 without any real decrease for further epochs.
Due to lack of available resources (GPUs and time) I had to choose to go with this model, but it should be noticed that there is room for a lof of improving.

2. The gradient accumulation training doesn't differ a whole lot from the standard training. They are pretty much the same, except we now accept a parameter for the number of batches to be accumulated.

```
def train_ga(epochs, model, criterion, optimizer, train_loader,
    device, val_loader, accumulation_batches, lr_scheduler,
    eval_every_n_batches=32):
    assert eval_every_n_batches % accumulation_batches == 0 and
        eval_every_n_batches >= accumulation_batches, "
        eval_every_n_batches must be a multiple of
        accumulation_batches"
    assert len(train_loader) % eval_every_n_batches == 0 and len(
        train_loader) >= eval_every_n_batches, "
        eval_every_n_batches must be a multiple of len(
        train_loader)"
```

```
    train_losses = []
    val_losses = []
    vocab = train_loader.dataset.vocabulary
    optimizer.zero_grad()
    running_loss = 0

    for epoch in range(epochs):
        for i, (data, targets) in enumerate(train_loader):
          model.train()
          data = data.to(device)
          targets_trim = targets[:,:-1].contiguous().to(device)
          outputs = model(data, targets_trim)
          targets_trim = targets[:,1:].contiguous().to(device)
          loss = criterion(outputs.view(outputs.size(0)*outputs.
              size(1),outputs.size(2)),targets_trim.view(
              targets_trim.size(0)*targets_trim.size(1)))
          loss = loss / accumulation_batches
          running_loss += loss.item()
          loss.backward()

          if (i+1) % accumulation_batches == 0 or (i+1) == len(
              train_loader):
              train_losses.append(running_loss)
              running_loss = 0
              if (i+1) % eval_every_n_batches == 0 or (i+1) == len
                  (train_loader):
                val_losses.append(evaluate(val_loader, model,
                    criterion, device))
                optimizer.step()
                if lr_scheduler:
                  lr_scheduler.step(val_losses[-1])
              else:
                optimizer.step()
              optimizer.zero_grad()

              if (i+1) % eval_every_n_batches == 0 or (i+1) == len
                  (train_loader):
                random_index = random.randint(0, len(targets)-1)
                target_sentence = " ".join([vocab.index2word[idx.
                    item()] for idx in targets[random_index][1:]])
                output_sentence = " ".join([vocab.index2word[idx.
                    item()] for idx in outputs.argmax(dim=-1)[
                    random_index]])

                print(f"Epoch: {epoch+1}/{epochs}, Batch: {i+1}/{
                    len(train_loader)}")
                print(f"Train Loss: {train_losses[-1]:.4f}")
                print(f"Validation Loss: {val_losses[-1]:.4f}")
                print(f"Random Target Sentence: {target_sentence}"
                    )
                print(f"Random Output Sentence: {output_sentence}\
                    n")

    return train_losses, val_losses, model, optimizer
```

We can then set up the same hyper-parameters as before with the addition of number of batches to accumulate and run training.

```
accumulation_batches = 4
epochs = 20

model = TransformerModel(vocab_size=len(vocab.word2index),d_model=
    d_model,pad_id=vocab.word2index["<PAD>"],encoder_layers=
    encoder_layers,decoder_layers=decoder_layers,dim_feedforward=
    feed_forward_dim,num_heads=nheads,dropout_transformer=
    dropout_transformer,dropout_posenconding=dropout_posenconding)
    .to(device)

criterion = nn.CrossEntropyLoss(ignore_index=vocab.word2index["<
    PAD>"])
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,'min',patience=patience, factor=0.8)

train_losses, val_losses, model, optimizer = train_ga(epochs,
    model, criterion, optimizer, train_loader,device, val_loader,
    accumulation_batches, lr_scheduler, eval_every_n_batches)

plt.plot(np.linspace(1,epochs,len(train_losses)),train_losses,
    label='Train Loss')
```

```
plt.plot(np.linspace(1,epochs,len(val_losses)),val_losses, label='
    Validation Loss')
plt.xlabel('Steps')
plt.ylabel('Loss')
plt.title('Train Loss vs Validation Loss')
plt.legend()
plt.show()
```

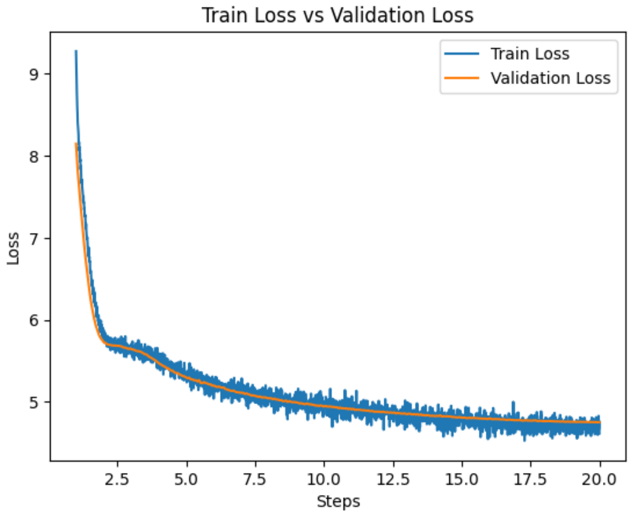

Figure 6: gradient accumulation train, 20 epochs

We can see from the above plot the train losses became more bouncy because we purposely increased the rate of updates. But nevertheless, the evaluation loss still remains unchanged, even with 10 more epochs than regular training. The inability to further decrease the loss and the closeness of the train and evaluation loss might suggest that the model is under-fitted. But after several tries with bigger models, to the point where i ran out of gpu time on google colab with two accounts, there were no significant improves with higher models.

This might suggest that some other underlying problem might be occurring. My data pre-processing and training pipleline are correct, the only possible variable might be from the implementation of the model itself. It was suggested by the coordinators of this course that we could use the nn.Transformer module directly and I'd like to leave a note on this regard. There are no examples on the internet for the use of the nn.transformer module directly (this is, without explicitly specifying the encoder and decoder apart), not even on the documentation of nn.Transformers itselt. The example they have there uses separate encoder and decoder from within the nn.Transformer module and is outdated (this is observable by the use of masks they have). As a student, I find transformers interesting and think of them as important, and I'm very disappointed that we didn't have more in-class time to properly explore these, as well as be given a working example without jumping straight to a huggingface implementation that doesn't really correlate well enough in terms of implementation with what we had to do in this assignment. Furthermore, the use of the nn.Transformer module fallsback into CPU when using CUDA because it's still in a prototype version for macbooks. The use of a flag before importing the torch library must be used to get around this error.

```
import os
os.environ['PYTORCH_ENABLE_MPS_FALLBACK'] = '1'
```

4 Evaluation (10 pts)

- The greedy strategy sample the most probable word index and concatenates it to the target tensor for the next computation, until the length of the sentence is achieved of the EOS token is found.

```
def get_greedy_answer(model, question, vocab, device, max_length=20):
    line,_ = process_sentence(question)
    line = line+['<EOS>']
    line = [vocab.word2index.get(word, vocab.word2index["<PAD>"]) for word in line]
    line = torch.tensor(line, dtype=torch.long).unsqueeze(0).to(device)
    target = torch.tensor([vocab.word2index["<SOS>"]], dtype=torch.long).unsqueeze(0).to(device)
    model.eval()
    with torch.no_grad():
        for _ in range(max_length):
            output = model(line,target)
            output = output.argmax(dim=-1)[:,-1]
            if output.item() == vocab.word2index["<EOS>"]:
                break
            target = torch.cat((target,output.unsqueeze(0)),dim=1)
    answer = " ".join([vocab.index2word[idx.item()] for idx in target.squeeze()[1:]])
    return answer
```

- Similarly to the greedy, we now sample through the distribution composed by the top k most probable words indexes and sample one of those indexes.

```
def get_topk_answers(model, question, vocab, device, k=5, max_length=20):
    line,_ = process_sentence(question)
    line = line+['<EOS>']
    line = [vocab.word2index.get(word, vocab.word2index["<PAD>"]) for word in line]
    line = torch.tensor(line).unsqueeze(0).to(device)
    target = torch.tensor([vocab.word2index["<SOS>"]], dtype=torch.long).unsqueeze(0).to(device)
    model.eval()
    with torch.no_grad():
        for _ in range(max_length):
            output = model(line,target)
            output = F.softmax(output, dim=-1)[0,-1,:].detach().cpu()
            topk_logits, topk_indices = torch.topk(output,k=k,dim=-1)
            p = F.softmax(topk_logits, dim=-1).numpy()
            w_idx = np.random.choice(topk_indices.squeeze(), p=p)
            if w_idx == vocab.word2index["<EOS>"]:
                break
            w_idx = torch.tensor(w_idx, dtype=torch.long).unsqueeze(0).unsqueeze(0).to(device)
            target = torch.cat((target,w_idx),dim=1)
```

```
    target = [vocab.index2word[idx.item()] for idx in target.
        squeeze()[1:]]
    return " ".join(target)
```

- Unfortunately, for reasons that I cannot understand why, but probably highly related to the still high loss value, for the greedy strategy the output is always the sames. For the topk strategy, because of stochasticy we get different sentences.

```
input1 = "Hello sir, how are you?"
input2 = "What is your name?"
input3 = "How old are you?"
print(get_greedy_answer(model, input1, vocab, device, max_length
    =20))
print(get_greedy_answer(model, input2, vocab, device, max_length
    =20))
print(get_greedy_answer(model, input3, vocab, device, max_length
    =20))
print(get_topk_answers(model, input1, vocab, device, max_length
    =20))
print(get_topk_answers(model, input2, vocab, device, max_length
    =20))
print(get_topk_answers(model, input3, vocab, device, max_length
    =20))
print(get_topk_answers(model, input1, vocab, device, max_length
    =20))
print(get_topk_answers(model, input2, vocab, device, max_length
    =20))
print(get_topk_answers(model, input3, vocab, device, max_length
    =20))
```

```
i am sorry
i am sorry
i am sorry
what
you can get out of a car and you are gon
you are a lot and i do it was going
i have a minute in this time in this way to get a little way to be
    in your little
no no you do not know that i do
i can not know that is all right and i am going to be in your way
    i am a
```

5 Bonus questions* (30 pts)

1. The accelarator module from huggingface accelarate handles the the gradient accumulation. This is achieved when instantiating the Accelarator. Hence, I implemented a small working dummy code to show how this can be done.

```
def train_ga_hf(epochs, model, optimizer, criterion, train_loader,
     val_loader, device, lr_scheduler, accumulation_steps):
    accelerator = Accelerator(gradient_accumulation_steps=
        accumulation_steps)
    model, optimizer, train_loader, lr_scheduler = accelerator.
        prepare(model, optimizer, train_loader, lr_scheduler)
    train_losses = []
    val_losses = []
    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        for i, (data, targets) in enumerate(train_loader):
            with accelerator.accumulate(model):
                data, targets = data.to(device), targets.to(device
                    )
```

```
                    output = model(data, targets[:, :-1])
                    loss = criterion(output.view(-1, output.size(-1)),
                        targets[:, 1:].contiguous().view(-1))
                    train_losses.append(loss.item())
                    accelerator.backward(loss)
                    optimizer.step()
                    if lr_scheduler and (i+1) % accumulation_steps==0:
                        val_losses.append(evaluate(model, criterion,
                            val_loader, device))
                        lr_scheduler.step()
                    optimizer.zero_grad()
    return train_losses, val_losses, model
```

2. Not done.

3. Not done.