CS187 Lab 4-5: Sequence-to-sequence models with attention

September 9, 2023

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab4-5.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

In lab 4-4, you built a sequence-to-sequence model in its most basic form and applied it to the task of words-to-numbers conversion. That model first encodes the source sequence into a fixed-size vector (encoder final states), and then decodes based on that vector. Since the only way information from the source side can flow to the target side is through this fixed-size vector, it presents a bottleneck in the encoder-decoder model: no matter how long the source sentence is, it must always be compressed into this fixed-size vector.

An *attention mechanism* (proposed in this seminal paper) offers a workaround by providing the decoder a dynamic view of the source-side as the decoding proceeds. Instead of compressing the source sequence into a *fixed-size* vector, we preserve the "resolution" and encode the source sequence into a *set of vectors* (usually with the same size as the source sequence) which is sometimes called a *memory bank*. When predicting each word, the decoder "attends to" this memory bank and assigns a weight to each vector in the set, and the weighted sum of those vectors will be used to make a prediction. Hopefully, the decoder will assign higher weights to more relevant source words when predicting a target word, which we'll test in this lab.

New bits of Pytorch used in this lab, and which you may find useful include:

- torch.bmm: Performs batched matrix multiplication.
- torch.nn.utils.rnn.pack_padded_sequence (imported as `pack`): Handles paddings. A more detailed explanation can be found here.
- torch.nn.utils.rnn.pad_packed_sequence (imported as `unpack`): Handles paddings.
- torch.masked_fill: Fills tensor elements with a value in spots where mask is `True`.
- torch.softmax: Computes softmax.
- torch.repeat: Repeats a tensor along the specified dimensions.
- torch.triu: Returns the upper triangular part of a matrix.

Preparation - Loading data

We use the same data as in lab 4-4.

[ ]:
```
import copy
import math
import matplotlib
import matplotlib.pyplot as plt
import os
import wget

import torch
import torch.nn as nn
import torchtext.legacy as tt

from tqdm import tqdm

from torch.nn.utils.rnn import pack_padded_sequence as pack
from torch.nn.utils.rnn import pad_packed_sequence as unpack
```

```
[ ]: # Spcify matplotlib configuration
%matplotlib inline
plt.style.use('tableau-colorblind10')

# GPU check, make sure to use GPU where available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
[ ]: # Download data
local_dir = "data/"
remote_dir = "https://github.com/nlp-course/data/raw/master/Words2Num/"
os.makedirs(local_dir, exist_ok=True)

for filename in [
    "train.src",
    "train.tgt",
    "dev.src",
    "dev.tgt",
    "test.src",
    "test.tgt",
]:
    wget.download(remote_dir + filename, out=local_dir)
```

As before, we use torchtext to load data. We use two fields: `SRC` for processing the source side (the English number phrases) and `TGT` for processing the target side (the digit sequences). And as in lab 4-4 , we prepend `<bos>` and appended `<eos>` to target sentences.

```
[ ]: SRC = tt.data.Field(include_lengths=True,          # include lengths
                    batch_first=False,             # batches will be max_src_len␣
 ↪x bsz
                    tokenize=lambda x: x.split(), # use split to tokenize
                   )
TGT = tt.data.Field(include_lengths=False,
                    batch_first=False,             # batches will be max_tgt_len␣
 ↪x bsz
                    tokenize=lambda x: x.split(), # use split to tokenize
                    init_token="<bos>",            # prepend <bos>
                    eos_token="<eos>")             # append <eos>
fields = [('src', SRC), ('tgt', TGT)]
```

```
[ ]: # Make splits for data
train_data, val_data, test_data = tt.datasets.TranslationDataset.splits(
    (".src", ".tgt"),
    fields,
    path=local_dir,
    train="train",
    validation="dev",
```

```
    test="test",
)

# Build vocabulary
SRC.build_vocab(train_data.src)
TGT.build_vocab(train_data.tgt)

print(f"Size of src vocab: {len(SRC.vocab)}")
print(f"Size of tgt vocab: {len(TGT.vocab)}")
print(f"Index for src padding: {SRC.vocab.stoi[SRC.pad_token]}")
print(f"Index for tgt padding: {TGT.vocab.stoi[TGT.pad_token]}")
print(f"Index for start of sequence token: {TGT.vocab.stoi[TGT.init_token]}")
print(f"Index for end of sequence token: {TGT.vocab.stoi[TGT.eos_token]}")
```

We batch training and validation data into minibatches, but for the test set, we use a batch size of 1, to make decoding implementation easier.

[ ]:
```
BATCH_SIZE = 32      # batch size for training and validation
TEST_BATCH_SIZE = 1 # batch size for test; we use 1 to make implementation␣
 ↪easier

train_iter, val_iter = tt.data.BucketIterator.splits((train_data, val_data),
                                                     batch_size=BATCH_SIZE,
                                                     device=device,
                                                     repeat=False,
                                                     sort_key=lambda x: len(x.
 ↪src), # sort by length to minimize padding
                                                     sort_within_batch=True)
test_iter = tt.data.BucketIterator(test_data,
                                   batch_size=TEST_BATCH_SIZE,
                                   device=device,
                                   repeat=False,
                                   sort=False,
                                   train=False)
```

Let’s take a look at a batch from these iterators.

[ ]:
```
batch = next(iter(train_iter))
src, src_lengths = batch.src
print (f"Size of src batch: {src.shape}")
print (f"Third src sentence in batch: {src[:, 2]}")
print (f"Length of the third src sentence in batch: {src_lengths[2]}")
print (f"Converted back to string: {' '.join([SRC.vocab.itos[i] for i in src[:,␣
 ↪2]])}")

tgt = batch.tgt
print (f"Size of tgt batch: {tgt.shape}")
```

```
print (f"Third tgt sentence in batch: {tgt[:, 2]}")
print (f"Converted back to string: {' '.join([TGT.vocab.itos[i] for i in tgt[:,␣
↪2]])}")
```

1 The attention mechanism

Attention works by *querying* a (dynamically sized) set of *keys* associated with *values*. As usual, the query, keys, and values are represented as vectors. The query process provides a score that specifies how much each key should be attended to. The attention can then be summarized by taking an average of the values weighted by the attention score of the corresponding keys. This *context vector* can then be used as another input to other processes.

More formally, let's suppose we have a query vector $\mathbf{q} \in \mathbb{R}^D$, a set of S key-value pairs $\{(\mathbf{k}_i, \mathbf{v}_i) \in \mathbb{R}^D \times \mathbb{R}^D : i \in \{1, 2, \cdots, S\}\}$, where D is the hidden size. What we want to do through the attention mechanism is to use the query to attend to the keys, and summarize those values associated with the "relevant" keys into a fixed-size context vector $\mathbf{c} \in \mathbb{R}^D$. Note that this is different from directly compressing the key-value pairs into a fixed-size vector, since depending on the query, we might end up with different context vectors.

To determine the score for a given query and key, it is standard to use a measure of similarity between the query and key. You've seen such similarity measures before, in labs 1-1 and 1-2. A good choice is simply the normalized dot product between query and key. We'll thus take the attention score for query $\mathbf{q}$ and key $\mathbf{k}_i$ to be

$$a_i = \frac{\exp(\mathbf{q} \cdot \mathbf{k}_i)}{Z},$$

where $\cdot$ denotes the dot product (inner product) and exp is exponentiation which ensures that all scores are nonnegative, and

$$Z = \sum_{i=1}^{S} \exp(\mathbf{q} \cdot \mathbf{k}_i)$$

is the normalizer to guarantee the scores all sum to one. (There are multiple ways of parameterizing the attention function, but the form we present here is the most popular one.) You might have noticed that the operation above is essentially a softmax over $\mathbf{q} \cdot \mathbf{k}$.

The attention scores $\mathbf{a}$ lie on a *simplex* (meaning $a_i \geq 0$ and $\sum_i a_i = 1$), which lends it some interpretability: the closer a_i is to 1, the more "relevant" a key k_i (and hence its value v_i) is to the given query. We will observe this later in the lab: When we are about to predict the target word "3", a_i is close to 1 for the source word $x_i =$ "three".

To compute the context vector $\mathbf{c}$, we take the weighted sum of values using the corresponding attention scores as weights:

$$\mathbf{c} = \sum_{i=1}^{S} a_i \mathbf{v}_i$$

The closer a_i is to 1, the higher the weight $\mathbf{v}_i$ receives.

Question: In the extreme, if there exists i for which a_i is 1, then what will the value of $\mathbf{c}$ be?

Type your answer here, replacing this text.

In practice, instead of computing the context vector once for each query, we want to batch computations for different queries together for parallel processing on GPUs. This will become especially useful for the transformer implementation. We use a matrix $Q \in \mathbb{R}^{T\times D}$ to store T queries, a matrix $K \in \mathbb{R}^{S\times D}$ to store S keys, and a matrix $V \in \mathbb{R}^{S\times D}$ to store the corresponding values. Then we can write down how we compute the attention scores $A \in \mathbb{R}^{T\times S}$ in a matrix form:

$$A = \text{softmax}(QK^\top, \text{dim} = -1),$$

Question: What is the shape of A? What does A_{ij} represent?

Type your answer here, replacing this text.

To get the context matrix $C \in \mathbb{R}^{T\times D}$:

$$C = AV$$

Your first job is to implement this calculation by finishing the attention function below, which takes the Q, K, and V matrices and returns the A and C matrices. Note that for these matrices, there is one additional dimension for the batching, so instead of $Q \in \mathbb{R}^{T\times D}$, $K, V \in \mathbb{R}^{S\times D}$, $A \in \mathbb{R}^{T\times S}$, $C \in \mathbb{R}^{T\times D}$, we have $Q \in \mathbb{R}^{T\times B\times D}$, $K, V \in \mathbb{R}^{S\times B\times D}$, $A \in \mathbb{R}^{B\times T\times S}$, $C \in \mathbb{R}^{T\times B\times D}$, where B is the batch size. In addition, the function below also takes an argument `mask` of size $\mathbb{R}^{B\times T\times S}$ to mark where attentions are disallowed. This is useful not only in disallowing attending to padding symbols, but also in implementing the transformer model which we'll see later in this lab.

> **Hint:** Notice that the batch dimension is the second dimension in Q, K, V, and C, but it is the first dimension in A and `mask`.
>
> **Hint:** You might find `torch.bmm` helpful for batched matrix multiplications. You might need to transpose and reshape tensors to be able to use this function.
>
> **Hint:** As mentioned in the beginning of the lab, you might also find `torch.transpose`, `torch.reshape`, `torch.masked_fill`, and `torch.softmax` useful.
>
> **Hint:** A simple trick for masking an attention score is to set it to negative infinity before normalization.

[ ]:
```
#TODO - finish implementing this function.
def attention(batched_Q, batched_K, batched_V, mask=None):
  """
  Performs the attention operation and returns the attention matrix
  `batched_A` and the context matrix `batched_C` using queries
  `batched_Q`, keys `batched_K`, and values `batched_V`.

  Arguments:
      batched_Q: (q_len, bsz, D)
      batched_K: (k_len, bsz, D)
```

```
        batched_V: (k_len, bsz, D)
        mask: (bsz, q_len, k_len). An optional boolean mask *disallowing*
              attentions where the mask value is *`False`*.
    Returns:
        batched_A: the normalized attention scores (bsz, q_len, k_ken)
        batched_C: a tensor of size (q_len, bsz, D).
    """
    # Check sizes
    D = batched_Q.size(-1)
    bsz = batched_Q.size(1)
    q_len = batched_Q.size(0)
    k_len = batched_K.size(0)
    assert batched_K.size(-1) == D and batched_V.size(-1) == D
    assert batched_K.size(1) == bsz and batched_V.size(1) == bsz
    assert batched_V.size(0) == k_len
    if mask is not None:
      assert mask.size() == torch.Size([bsz, q_len, k_len])
    ...
    batched_A = ...
    batched_C = ...
    # Verify that things sum up to one properly.
    assert torch.all(torch.isclose(batched_A.sum(-1),
                               torch.ones(bsz, q_len).to(device)))
    return batched_A, batched_C
```

[ ]:
```
grader.check("attention")
```

1.1 Neural encoder-decoder models with attention

Now we can add an attention mechanism to our encoder-decoder model. As in lab 4-4, we use a bidirectional LSTM as the encoder, and a unidirectional LSTM as the decoder, and initialize the decoder state with the encoder final state. However, instead of directly projecting the decoder hidden state to logits, we use it as a query vector and attend to all encoder outputs (used as both keys and values), and then concatanate the resulting context vector with the query vector, and project to logits. In addition, we add the context vector to the word embedding at the next time step, so that the LSTM can be aware of the previous attention results.

In the above illustration, at the first time step, we use q_1 to denote the decoder output. Instead of directly projecting that to logits as in lab 4-4, we use q_1 as the query vector, and use it to attend to the memory bank (which is the set of encoder outputs) and get the context vector c_1. We concatenate c_1 with q_1, and project the result to the vocabulary size to get logits. At the next step, we first embed y_1 into embeddings, and then **add** c_1 to it (via componentwise addition) and use the sum as the decoder input. This process continues until an end-of-sequence is produced.

You'll need to implement `forward_encoder` and `forward_decoder_incrementally` in the code below. The `forward_encoder` function will return a "memory bank" in addition to the final states. The "memory bank" is simply the encoder outputs at all time steps, which is the first returned value of `torch.nn.LSTM`.

The `forward_decoder_incrementally` function forwards the LSTM cell for a single time step. It takes the initial decoder state, the memory bank, and the input word at the current time step and returns logits for this time step. In addition, it needs to return the context vector and the updated decoder state, which will be used for the next time step. Note that here you need to consider **batch sizes greater than 1**, as this function is used in `forward_decoder`, which is used during training.

In summary, the steps in decoding are:

1. Map the target words to word embeddings. Add the context vector from the previous time step if any. Use the result as the input to the decoder.
2. Forward the decoder RNN for one time step. Use the decoder output as query, the memory bank as **both keys and values**, and compute the context vector through the attention mechanism. Since we don't want to attend to padding symbols at the source side, we also need to pass in a proper `mask` to the attention function.
3. Concatenate the context vector with the decoder output, and project the concatenation to vocabulary size as (unnormalized) logits. Normalize them using `torch.log_softmax` if `normalize` is `True`.
4. Update the decoder hidden state and the context vector, which will be used in the next time step.

Before proceeding, let's consider a simple question: in lab 4-4, we tried to avoid `for` loops, but if you read the code of `forward_decoder` in this lab, you might notice a `for` loop. Is this unavoidable?

Question: Recall that in the `forward_decoder` function in lab 4-4 we didn't use any for loops but instead used a single call to `self.decoder_rnn`. Why do we need a `for` loop in the function `forward_decoder` below? Is it possible to get rid of the for loop to make the code more efficient?

Type your answer here, replacing this text.

Now let's implement `forward_encoder` and `forward_decoder_incrementally`.

> Hint on using `pack`: if you use `pack` to handle paddings and pass the result as encoder inputs, you need to use `unpack` and extract the first returned value as the memory bank. An example can be found here, but note that our input is already the padded sequences, and that we set `batch_first` to `False`. Hint on ignoring source-side paddings in the attention mechanism: what `mask` should we pass into the `attention` function??

[ ]:
```
#TODO - implement `forward_encoder` and `forward_decoder_incrementally`.
class AttnEncoderDecoder(nn.Module):
  def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
    """
    Initializer. Creates network modules and loss function.
    Arguments:
        src_field: src field
        tgt_field: tgt field
```

```python
        hidden_size: hidden layer size of both encoder and decoder
        layers: number of layers of both encoder and decoder
    """
    super().__init__()
    self.src_field = src_field
    self.tgt_field = tgt_field

    # Keep the vocabulary sizes available
    self.V_src = len(src_field.vocab.itos)
    self.V_tgt = len(tgt_field.vocab.itos)

    # Get special word ids
    self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
    self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
    self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
    self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]

    # Keep hyper-parameters available
    self.embedding_size = hidden_size
    self.hidden_size = hidden_size
    self.layers = layers

    # Create essential modules
    self.word_embeddings_src = nn.Embedding(self.V_src, self.embedding_size)
    self.word_embeddings_tgt = nn.Embedding(self.V_tgt, self.embedding_size)

    # RNN cells
    self.encoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size // 2, # to match decoder hidden size
      num_layers    = layers,
      bidirectional = True              # bidirectional encoder
    )
    self.decoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size,
      num_layers    = layers,
      bidirectional = False             # unidirectional decoder
    )

    # Final projection layer
    self.hidden2output = nn.Linear(2*hidden_size, self.V_tgt) # project the␣
↪concatenation to logits

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum',
                                             ignore_index=self.padding_id_tgt)
```

```python
def forward_encoder(self, src, src_lengths):
    """
    Encodes source words `src`.
    Arguments:
        src: src batch of size (max_src_len, bsz)
        src_lengths: src lengths of size (bsz)
    Returns:
        memory_bank: a tensor of size (src_len, bsz, hidden_size)
        (final_state, context): `final_state` is a tuple (h, c) where h/c is of␣
↪size
                                (layers, bsz, hidden_size), and `context` is␣
↪`None`.
    """
    #TODO
    ...
    memory_bank = ...
    final_state = ...
    context = None
    return memory_bank, (final_state, context)

def forward_decoder(self, encoder_final_state, tgt_in, memory_bank, src_mask):
    """
    Decodes based on encoder final state, memory bank, src_mask, and ground␣
↪truth
    target words.
    Arguments:
        encoder_final_state: (final_state, None) where final_state is the␣
↪encoder
                             final state used to initialize decoder. None is the
                             initial context (there's no previous context at the
                             first step).
        tgt_in: a tensor of size (tgt_len, bsz)
        memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder␣
↪outputs
                     at every position
        src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False`␣
↪where
                  src is padding (we disallow decoder to attend to those␣
↪places).
    Returns:
        Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
    """
    max_tgt_length = tgt_in.size(0)

    # Initialize decoder state, note that it's a tuple (state, context) here
```

```
        decoder_states = encoder_final_state

        all_logits = []
        for i in range(max_tgt_length):
            logits, decoder_states, attn = \
                self.forward_decoder_incrementally(decoder_states,
                                                   tgt_in[i],
                                                   memory_bank,
                                                   src_mask,
                                                   normalize=False)
            all_logits.append(logits)              # list of bsz, vocab_tgt
        all_logits = torch.stack(all_logits, 0) # tgt_len, bsz, vocab_tgt
        return all_logits

    def forward(self, src, src_lengths, tgt_in):
        """
        Performs forward computation, returns logits.
        Arguments:
            src: src batch of size (max_src_len, bsz)
            src_lengths: src lengths of size (bsz)
            tgt_in:  a tensor of size (tgt_len, bsz)
        """
        src_mask = src.ne(self.padding_id_src) # max_src_len, bsz
        # Forward encoder
        memory_bank, encoder_final_state = self.forward_encoder(src, src_lengths)
        # Forward decoder
        logits = self.forward_decoder(encoder_final_state, tgt_in, memory_bank,␣
↪src_mask)
        return logits

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask,
                                      normalize=True):
        """
        Forward the decoder for a single step with token `tgt_in_onestep`.
        This function will be used both in `forward_decoder` and in beam search.
        Note that bsz can be greater than 1.
        Arguments:
            prev_decoder_states: a tuple (prev_decoder_state, prev_context).␣
↪`prev_context`
                                 is `None` for the first step
            tgt_in_onestep: a tensor of size (bsz), tokens at one step
            memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder␣
↪outputs
                         at every position
            src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False`␣
↪where
```

```
                 src is padding (we disallow decoder to attend to those␣
↪places).
        normalize: use log_softmax to normalize or not. Beam search needs to␣
↪normalize,
                  while `forward_decoder` does not
    Returns:
        logits: log probabilities for `tgt_in_token` of size (bsz, V_tgt)
        decoder_states: (`decoder_state`, `context`) which will be used for the
                        next incremental update
        attn: normalized attention scores at this step (bsz, src_len)
    """
    prev_decoder_state, prev_context = prev_decoder_states
    #TODO
    ...
    decoder_states = (decoder_state, context)
    if normalize:
      logits = torch.log_softmax(logits, dim=-1)
    return logits, decoder_states, attn

  def evaluate_ppl(self, iterator):
    """Returns the model's perplexity on a given dataset `iterator`."""
    # Switch to eval mode
    self.eval()
    total_loss = 0
    total_words = 0
    for batch in iterator:
      # Input and target
      src, src_lengths = batch.src
      tgt = batch.tgt # max_length_sql, bsz
      tgt_in = tgt[:-1] # remove <eos> for decode input (y_0=<bos>, y_1, y_2)
      tgt_out = tgt[1:] # remove <bos> as target        (y_1, y_2, y_3=<eos>)
      # Forward to get logits
      logits = self.forward(src, src_lengths, tgt_in)
      # Compute cross entropy loss
      loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
      total_loss += loss.item()
      total_words += tgt_out.ne(self.padding_id_tgt).float().sum().item()
    return math.exp(total_loss/total_words)

  def train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001):
    """Train the model."""
    # Switch the module to training mode
    self.train()
    # Use Adam to optimize the parameters
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_ppl = float('inf')
    best_model = None
```

```
    # Run the optimization for multiple epochs
    for epoch in range(epochs):
      total_words = 0
      total_loss = 0.0
      for batch in tqdm(train_iter):
        # Zero the parameter gradients
        self.zero_grad()
        # Input and target
        src, src_lengths = batch.src # text: max_src_length, bsz
        tgt = batch.tgt # max_tgt_length, bsz
        tgt_in = tgt[:-1] # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
        tgt_out = tgt[1:] # Remove <bos> as target        (y_1, y_2, y_3=<eos>)
        bsz = tgt.size(1)
        # Run forward pass and compute loss along the way.
        logits = self.forward(src, src_lengths, tgt_in)
        loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
        # Training stats
        num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum().item()
        total_words += num_tgt_words
        total_loss += loss.item()
        # Perform backpropagation
        loss.div(bsz).backward()
        optim.step()

      # Evaluate and track improvements on the validation dataset
      validation_ppl = self.evaluate_ppl(val_iter)
      self.train()
      if validation_ppl < best_validation_ppl:
        best_validation_ppl = validation_ppl
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = total_loss / total_words
      print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
             f'Validation Perplexity: {validation_ppl:.4f}')
```

[ ]:
```
EPOCHS = 2 # epochs, we highly recommend starting with a smaller number like 1
LEARNING_RATE = 2e-3 # learning rate

# Instantiate and train classifier
model = AttnEncoderDecoder(SRC, TGT,
  hidden_size    = 64,
  layers         = 3,
).to(device)

model.train_all(train_iter, val_iter, epochs=EPOCHS,␣
 ↪learning_rate=LEARNING_RATE)
model.load_state_dict(model.best_model)
```

Since the task we consider here is very simple, we should expect a perplexity very close to 1.

[ ]:
```
# Evaluate model performance, the expected value should be < 1.05
print (f'Test perplexity: {model.evaluate_ppl(test_iter):.3f}')
```

[ ]:
```
grader.check("encoder_decoder_ppl")
```

1.2 Beam search decoding

We can reuse most of our beam search code in lab 4-4 here: we only need to modify the code a bit to pass in `memory_bank` and `src_mask`. For reference here is the same pseudo-code used in lab 4-4, where we want to decode a single example `x` of maximum length `max_T` using a beam size of `K`.

```
1.  def beam_search(x, K, max_T):
2.      finished = []       # for storing completed hypotheses
        # Initialize the beam
3.      beams = [Beam(hyp=(bos), score=0)] # initial hypothesis: bos, initial score: 0

4.      for t in [1..max_T]  # main body of search over time steps
5.          hypotheses = []

            # Expand each beam by all possible tokens y_{t+1}
6.          for beam in beams:
7.              y_{1:t}, score = beam.hyp, beam.score
8.              for y_{t+1} in V:
9.                  y_{1:t+1} = y_{1:t} + [y_{t+1}]
10.                 new_score = score + log P(y_{t+1} | y_{1:t}, x)
11.                 hypotheses.append(Beam(hyp=y_{1:t+1}, score=new_score))

            # Find K best next beams
12.         beams = sorted(hypotheses, key=lambda beam: -beam.score)[:K]

            # Set aside finished beams (those that end in <eos>)
13.         for beam in beams:
14.             y_{t+1} = beam.hyp[-1]
15.             if y_{t+1} == eos:
16.                 finished.append(beam)
17.                 beams.remove(beam)

            # Break the loop if everything is finished
18.         if len(beams) == 0:
19.             break
20.     return sorted(finished, key=lambda beam: -beam.score)[0] # return the best finished hy
```

Implement function `beam_search` in the code below. In addition to the predicted target sequence, this function also returns a list of attentions `all_attns`.

[ ]:

```python
# max target length
MAX_T = 15
class Beam():
  """
  Helper class for storing a hypothesis, its score and its decoder hidden state.
  """
  def __init__(self, decoder_state, tokens, score):
    self.decoder_state = decoder_state
    self.tokens = tokens
    self.score = score

class BeamSearcher():
  """
  Main class for beam search.
  """
  def __init__(self, model):
    self.model = model
    self.bos_id = model.bos_id
    self.eos_id = model.eos_id
    self.padding_id_src = model.padding_id_src
    self.V = model.V_tgt


  def beam_search(self, src, src_lengths, K, max_T=MAX_T):
    """
    Performs beam search decoding.
    Arguments:
        src: src batch of size (max_src_len, 1)
        src_lengths: src lengths of size (1)
        K: beam size
        max_T: max possible target length considered
    Returns:
        a list of token ids and a list of attentions
    """
    finished = []
    all_attns = []
    # Initialize the beam
    self.model.eval()
    #TODO - fill in `memory_bank`, `encoder_final_state`, and `init_beam` below
    ...
    memory_bank = ...
    encoder_final_state = ...
    init_beam = ...
    beams = [init_beam]

    with torch.no_grad():
      for t in range(max_T): # main body of search over time steps
```

```
        # Expand each beam by all possible tokens y_{t+1}
        all_total_scores = []
        for beam in beams:
          y_1_to_t, score, decoder_state = beam.tokens, beam.score, beam.decoder_state
          y_t = y_1_to_t[-1]
          #TODO - finish the code below
          # Hint: you might want to use `model.forward_decoder_incrementally` with `normalize=True`
          src_mask = src.ne(self.padding_id_src)
          ...
          logits = ...
          decoder_state = ...
          attn = ...
          total_scores = ...
          all_total_scores.append(total_scores)
          all_attns.append(attn) # keep attentions for visualization
          beam.decoder_state = decoder_state # update decoder state in the beam
        all_total_scores = torch.stack(all_total_scores) # (K, V) when t>0, (1, V) when t=0

        # Find K best next beams
        # The code below has the same functionality as line 6-12, but is more efficient
        all_scores_flattened = all_total_scores.view(-1) # K*V when t>0, 1*V when t=0
        topk_scores, topk_ids = all_scores_flattened.topk(K, 0)
        beam_ids = topk_ids.div(self.V, rounding_mode='floor')
        next_tokens = topk_ids - beam_ids * self.V
        new_beams = []
        for k in range(K):
          beam_id = beam_ids[k]       # which beam it comes from
          y_t_plus_1 = next_tokens[k] # which y_{t+1}
          score = topk_scores[k]
          beam = beams[beam_id]
          decoder_state = beam.decoder_state
          y_1_to_t = beam.tokens
          #TODO
          new_beam = ...
          new_beams.append(new_beam)
        beams = new_beams

        # Set aside completed beams
        # TODO - move completed beams to `finished` (and remove them from `beams`)
```

```
        ...

        # Break the loop if everything is completed
        if len(beams) == 0:
            break

    # Return the best hypothesis
    if len(finished) > 0:
      finished = sorted(finished, key=lambda beam: -beam.score)
      return finished[0].tokens, all_attns
    else: # when nothing is finished, return an unfinished hypothesis
      return beams[0].tokens, all_attns
```

[ ]:
```
grader.check("beam_search")
```

Now we can use beam search decoding to predict the outputs for the test set inputs using the trained model. You should expect an accuracy close to 100%.

[ ]:
```
DEBUG_FIRST = 10  # set to 0 to disable printing predictions
K = 1             # beam size 1

correct = 0
total = 0

# create beam searcher
beam_searcher = BeamSearcher(model)

for index, batch in enumerate(test_iter, start=1):
  # Input and output
  src, src_lengths = batch.src
  # Predict
  prediction, _ = beam_searcher.beam_search(src, src_lengths, K)
  # Convert to string
  prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
  prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
  ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.
↪view(-1)])
  ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
  if DEBUG_FIRST > index:
    src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
    print (f'Source: {src}')
    print (f'Prediction:   {prediction}')
    print (f'Ground truth: {ground_truth}')
  if ground_truth == prediction:
    correct += 1
  total += 1
```

```
print (f'Accuracy: {correct/total:.2f}')
```

2 Visualizing attention

We can visualize how each query distributes its attention scores over each source word.

[ ]:
```
K = 1 # this code only works for beam size 1

# Create beam searcher
beam_searcher = BeamSearcher(model)
batch = next(iter(test_iter))
# Input and output
src, src_lengths = batch.src
# Predict and get attentions
prediction, all_attns = beam_searcher.beam_search(src, src_lengths, K)
all_attns = torch.stack(all_attns, 0)
# Convert to string
prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.view(-1)])
ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
print (f'Source: {src}')
print (f'Prediction:   {prediction}')
print (f'Ground truth: {ground_truth}')

# Plot
fig, ax = plt.subplots(figsize=(8, 6))

ax.imshow(all_attns[:,0,:].detach().cpu())
ax.set_yticks(list(range(1+len(prediction.split()))));
ax.set_yticklabels(prediction.split() + ['eos']);
ax.set_xticks(list(range(len(src.split()))));
ax.set_xticklabels(src.split());

# Uncomment the line below if the plot does not show up
# Make sure to comment that before submitting to gradescope
# since there would be some autograder issues with plt.show()
#plt.show()
```

Do these attentions make sense? Do you see how the attention mechanism solves the bottleneck problem in vanilla seq2seq?

3 The transformer architecture

In RNN-based neural encoder-decoder models, we used recurrence to model the dependencies among words. For example, by running a unidirectional RNN from y_1 to y_t, we can consider the past history when predicting y_{t+1}. However, running an RNN over a sequence is a serial process: we need to wait for it to finish running from y_1 to y_t before being able to compute the outputs at y_{t+1}. This serial process cannot be parallelized on GPUs along the sequence length dimension: even during training where all y_t's are available, we cannot compute the logits for y_t and the logits for y_{t+1} in parallel.

The attention mechanism provides an alternative, and most importantly, parallelizable solution. The transformer model completely gets rid of recurrence and only uses attention to model the dependencies among words. For example, we can use attention to incorporate the representations from y_1 to y_t when predicting y_{t+1}, simply by attending to their word embeddings. This is called *decoder self-attention*.

Question: By getting rid of recurrence and only using decoder self-attention, can we compute the logits for any two different words y_{t_1} and y_{t_2} in parallel at training time (only consider decoder for now)? Why?

Type your answer here, replacing this text.

Similarly, at the encoder side, for each word x_i, we let it attend to the embeddings of $x_1, \ldots, x_S$, to model the context in which x_i appears. This is called *encoder self-attention*. It is different from decoder self-attention in that here every word attends to all words, but at the decoder side, every word can only attend to the previous words (since the prediction of word y_t cannot use the information from any $y_{\geq t}$).

To incorporate source-side information at the decoder side, at each time step, we let the decoder attend to the top-layer encoder outputs, as we did in the RNN-based encoder-decoder model above. This is called *cross-attention*. Note that there's no initialization of decoder hidden state here, since we no longer use an RNN.

The process we describe above is only a single layer of attention. In practice, transformers stack multiple layers of attention and feedforward layers, using the outputs from the layer below as the inputs to the layer above, as shown in the illustration below.

In the above illustration, due to space limits, we ommited the details of encoder self-attention and decoder self-attention, and we describe it here, using encoder-self-attention at layer 0 as an example. First, we use three linear projections to project each hidden state $h_{0,i}$ to a query vector $q_{0,i}$, a key vector $k_{0,i}$, and a value vector $v_{0,i}$. Then at each position i, we use q_i as the query, and $\{(k_{0,j}, v_{0,j}) : j \in \{1, \ldots, S\}\}$ as keys/values to produce a context vector $c_{0,i}$. Note that the keys/values are the same for different positions, and the only difference is that a different query vector is used for each position.

A clear difference between the transformer architecture and the RNN-based encoder decoder architecture is that there are no horizontal arrows in the transformer model: transformers only use position-wise operations and attention operations. The dependencies among words are **only introduced by the attention operations**, while the other operations such as feedforwad, nonlinearity, and normalization are position-wise, that is, they do not depend on other positions, and can thus be performed in parallel.

Question: In the above transformer model, if we shuffle the input words $x_1, \ldots, x_4$, would we get a different distribution over y? Why or why not?

Type your answer here, replacing this text.

Since the transformer model itself doesn't have any sense of position/order, we encode the position of the word in the sentence, and add it to the word embedding as the input representation, as illustrated below.

> The illustrations above also omitted residual connections, which add the inputs to certain operations (such as attention and feedforward) to the outputs. More details can be found in the code below.

3.1 Causal attention mask

To efficiently train the transformer model, we want to batch the attention operations together such that they can be fully parallelized along the sequence length dimension. (The non-attention operations are position-wise so they are trivally parallelizable.) This is quite straightforward for encoder self-attention and decoder-encoder cross-attention given our batched implementation of the `attention` function. However, things are a bit trickier for the decoder: each word y_t attends to $t-1$ previous words $y_1, \ldots, y_{t-1}$, which means each word y_t has a different set of key-value pairs. Is it possible to batch them together?

The solution is to use *attention masks*. For every word y_t, we give it all key-value pairs at $y_1, \ldots, y_T$, and we disallow attending to future words $y_t, y_{t+1}, \ldots, y_T$ through an attention mask. (Recall that the `attention` function takes a `mask` argument.) We usually call this attention mask a *causal attention mask*, as it prevents the leakage of information from the future into the past. Since every y_t has the same set of (key, value) pairs, we can batch them and compute the context vectors using a single call to the function `attention`.

What should such a mask be? Implement the `causal_mask` function below to generate this mask.

> Hint: you might find `torch.triu` useful.

[ ]:
```python
#TODO - implement this function, which returns a causal attention mask
def causal_mask(T):
    """
    Generate a causal mask.
    Arguments:
        T: the length of target sequence
    Returns:
        mask: a T x T tensor, where `mask[i, j]` should be `True`
        if y_i can attend to y_{j-1} (there's a "-1" since the first
        token in decoder input is <bos>) and `False` if y_i cannot
        attend to y_{j-1}
    """
    mask = ...
    return mask.to(device)
```

[ ]:
```python
grader.check("causal_attention_mask")
```

We can visualize the attention mask and manually check if it's what we expected.

[ ]:
```
fig, ax = plt.subplots(figsize=(8, 6))

T = 7
mask = causal_mask(T)
ax.imshow(mask.cpu())

# Uncomment the line below if the plot does not show up
# Make sure to comment that before submitting to gradescope
# since there would be some autograder issues with `plt.show()`
#plt.show()
```

As we have emphasized multiple times, unlike RNN-based encoder-decoders, transformer encoder/decoders are parallelizable in the sequence length dimension, even for the decoder: by using causal masks, all positions (at the same layer) can be computed all at once (if the lower layer has been computed). The parallelizability of transformers is the key to its success since it allows for training it on vast amounts of data.

Now we are ready to complete the implementation of the transformer model. The code is structured as a set of classes: `TransformerEncoderLayer`*, `TransformerEncoder`, `TransformDecoderLayer`*, `TransformDecoder`, `PositionalEmbedding`, and `TransformerEncoderDecoder`*. We've provided almost all the necessary code. In particular, we provide code for all position-wise operations. Your job is only to implement the parts involving attention and to figure out the correct attention masks, which involves only the three classes marked above with a star.

> Hint: Completing this transformer implementation should require very little code, just a few lines.
>
> Hint: The causal mask is a 2-D matrix, but we want to add a batch dimension, and expand it to be of the desired size. For this purpose, you can use `torch.repeat`.

[ ]:
```
#TODO - implement `forward_encoder` and `forward_decoder`.
# `TransformerEncoderDecoder` inherits most functions from `AttnEncoderDecoder`
class TransformerEncoderDecoder(AttnEncoderDecoder):
  def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
    """
    Initializer. Creates network modules and loss function.
    Arguments:
        src_field: src field
        tgt_field: tgt field
        hidden_size: hidden layer size of both encoder and decoder
        layers: number of layers of both encoder and decoder
    """
    super(AttnEncoderDecoder, self).__init__()
    self.src_field = src_field
    self.tgt_field = tgt_field

    # Keep the vocabulary sizes available
```

```
        self.V_src = len(src_field.vocab.itos)
        self.V_tgt = len(tgt_field.vocab.itos)

        # Get special word ids
        self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
        self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
        self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
        self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]

        # Keep hyper-parameters available
        self.embedding_size = hidden_size
        self.hidden_size = hidden_size
        self.layers = layers

        # Create essential modules
        self.encoder = TransformerEncoder(self.V_src, hidden_size, layers)
        self.decoder = TransformerDecoder(self.V_tgt, hidden_size, layers)

        # Final projection layer
        self.hidden2output = nn.Linear(hidden_size, self.V_tgt)

        # Create loss function
        self.loss_function = nn.CrossEntropyLoss(reduction='sum',
                                                 ignore_index=self.padding_id_tgt)

    def forward_encoder(self, src, src_lengths):
        """
        Encodes source words `src`.
        Arguments:
            src: src batch of size (max_src_len, bsz)
            src_lengths: src lengths (bsz)
        Returns:
            memory_bank: a tensor of size (src_len, bsz, hidden_size)
        """
        # The reason we don't directly pass in src_mask as in `forward_decoder` is␣
↪to
        # enable us to reuse beam search implemented for RNN-based encoder-decoder
        src_len = src.size(0)
        #TODO - compute `encoder_self_attn_mask`
        encoder_self_attn_mask = ...
        memory_bank = self.encoder(src, encoder_self_attn_mask)
        return memory_bank, None

    def forward_decoder(self, tgt_in, memory_bank, src_mask):
        """
        Decodes based on memory bank, and ground truth target words.
        Arguments:
```

```
            tgt_in: a tensor of size (tgt_len, bsz)
            memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder␣
↪outputs
                         at every position
            src_mask: a tensor of size (src_len, bsz) which is `False` for source␣
↪paddings
        Returns:
            Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
        """
        tgt_len = tgt_in.size(0)
        bsz = tgt_in.size(1)
        #TODO - compute `cross_attn_mask` and `decoder_self_attn_mask`
        cross_attn_mask = ...
        decoder_self_attn_mask = ...

        outputs = self.decoder(tgt_in, memory_bank, cross_attn_mask,␣
↪decoder_self_attn_mask)
        logits = self.hidden2output(outputs)
        return logits

    def forward(self, src, src_lengths, tgt_in):
        """
        Performs forward computation, returns logits.
        Arguments:
            src: src batch of size (max_src_len, bsz)
            src_lengths: src lengths of size (bsz)
            tgt_in:  a tensor of size (tgt_len, bsz)
        """
        src_mask = src.ne(self.padding_id_src) # max_src_len, bsz
        # Forward encoder
        memory_bank, _ = self.forward_encoder(src, src_lengths)
        # Forward decoder
        logits = self.forward_decoder(tgt_in, memory_bank, src_mask)
        return logits

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask, normalize=True):
        """
        Forward the decoder at `decoder_state` for a single step with token␣
↪`tgt_in_onestep`.
        This function will be used in beam search. Note that the implementation␣
↪here is
        very inefficient, since we do not cache any decoder state, but instead we␣
↪only
        cache previously generated tokens in `prev_decoder_states`, and do a fresh
        `forward_decoder`.
```

```
    Arguments:
        prev_decoder_states: previous tgt words. None for the first step.
        tgt_in_onestep: a tensor of size (bsz), tokens at one step
        memory_bank: a tensor of size (src_len, bsz, hidden_size), src hidden␣
↪states
                    at every position
        src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False`␣
↪where
                  src is padding.
        normalize: use log_softmax to normalize or not. Beam search needs to␣
↪normalize,
                   while `forward_decoder` does not
    Returns:
        logits: Log probabilities for `tgt_in_token` of size (bsz, V_tgt)
        decoder_states: we use tgt words up to now as states, a tensor of size␣
↪(len, bsz)
        None: to keep output format the same as AttnEncoderDecoder, such that␣
↪we can
              reuse beam search code

    """
    prev_tgt_in = prev_decoder_states # tgt_len, bsz
    src_len = memory_bank.size(0)
    bsz = memory_bank.size(1)
    tgt_in_onestep = tgt_in_onestep.view(1, -1) # 1, bsz
    if prev_tgt_in is not None:
      tgt_in = torch.cat((prev_tgt_in, tgt_in_onestep), 0) # tgt_len+1, bsz
    else:
      tgt_in = tgt_in_onestep
    tgt_len = tgt_in.size(1)

    logits = self.forward_decoder(tgt_in, memory_bank, src_mask)
    logits = logits[-1]
    if normalize:
      logits = torch.log_softmax(logits, dim=-1)
    decoder_states = tgt_in
    return logits, decoder_states, None
```

[ ]:
```
class TransformerEncoder(nn.Module):
  r"""TransformerEncoder is an embedding layer and a stack of N encoder layers.
  Arguments:
      hidden_size: hidden size.
      layers: the number of encoder layers.
  """

  def __init__(self, vocab_size, hidden_size, layers):
    super().__init__()
```

```
    self.embed = PositionalEmbedding(vocab_size, hidden_size)
    encoder_layer = TransformerEncoderLayer(hidden_size)
    self.layers = _get_clones(encoder_layer, layers)
    self.norm = nn.LayerNorm(hidden_size)

  def forward(self, src, encoder_self_attn_mask):
    r"""Pass the input through the word embedding layer, followed by
    the encoder layers in turn.
    Arguments:
        src: src batch of size (max_src_len, bsz)
        encoder_self_attn_mask: the mask for encoder self-attention, it's of size
                                (bsz, max_src_len, max_src_len)
    Returns:
        a tensor of size (max_src_len, bsz, hidden_size)
    """
    output = self.embed(src)
    for mod in self.layers:
      output = mod(output, encoder_self_attn_mask=encoder_self_attn_mask)
    output = self.norm(output)
    return output


class TransformerEncoderLayer(nn.Module):
  r"""TransformerEncoderLayer is made up of self-attn and feedforward network.
  Arguments:
      hidden_size: hidden size.
  """

  def __init__(self, hidden_size):
    super(TransformerEncoderLayer, self).__init__()
    self.hidden_size = hidden_size
    fwd_hidden_size = hidden_size * 4

    # Create modules
    self.linear1 = nn.Linear(hidden_size, fwd_hidden_size)
    self.linear2 = nn.Linear(fwd_hidden_size, hidden_size)
    self.norm1 = nn.LayerNorm(hidden_size)
    self.norm2 = nn.LayerNorm(hidden_size)
    self.activation = nn.ReLU()
    # Attention related
    self.q_proj = nn.Linear(hidden_size, hidden_size)
    self.k_proj = nn.Linear(hidden_size, hidden_size)
    self.v_proj = nn.Linear(hidden_size, hidden_size)
    self.context_proj = nn.Linear(hidden_size, hidden_size)
```

```
  def forward(self, src, encoder_self_attn_mask):
    r"""Pass the input through the encoder layer.
    Arguments:
        src: an input tensor of size (max_src_len, bsz, hidden_size).
        encoder_self_attn_mask: attention mask of size (bsz, max_src_len,␣
↪max_src_len),
                                it's `False` where the corresponding attention␣
↪is disabled
    Returns:
        a tensor of size (max_src_len, bsz, hidden_size).
    """
    # Attend
    q = self.q_proj(src) / math.sqrt(self.hidden_size) # a trick needed to make␣
↪transformer work
    k = self.k_proj(src)
    v = self.v_proj(src)
    #TODO - compute `context`
    context = ...
    src2 = self.context_proj(context)
    # Residual connection
    src = src + src2
    src = self.norm1(src)
    # Feedforward for each position
    src2 = self.linear2(self.activation(self.linear1(src)))
    src = src + src2
    src = self.norm2(src)
    return src


class TransformerDecoder(nn.Module):
  r"""TransformerDecoder is an embedding layer and a stack of N decoder layers.
  Arguments:
      hidden_size: hidden size.
      layers: the number of sub-encoder-layers in the encoder.
  """
  def __init__(self, vocab_size, hidden_size, layers):
    super(TransformerDecoder, self).__init__()
    self.embed = PositionalEmbedding(vocab_size, hidden_size)
    decoder_layer = TransformerDecoderLayer(hidden_size)
    self.layers = _get_clones(decoder_layer, layers)
    self.norm = nn.LayerNorm(hidden_size)

  def forward(self, tgt_in, memory, cross_attn_mask, decoder_self_attn_mask):
    r"""Pass the inputs (and mask) through the word embedding layer, followed by
    the decoder layer in turn.
    Arguments:
        tgt_in: tgt batch of size (max_tgt_len, bsz)
```

```
        memory: the outputs of the encoder (max_src_len, bsz, hidden_size)
        cross_attn_mask: attention mask of size (bsz, max_tgt_len,␣
↪max_src_len),
                         it's `False` where the cross-attention is disallowed.
        decoder_self_attn_mask: attention mask of size (bsz, max_tgt_len,␣
↪max_tgt_len),
                                it's `False` where the self-attention is␣
↪disallowed.
    Returns:
        a tensor of size (max_tgt_len, bsz, hidden_size)
    """
    output = self.embed(tgt_in)
    for mod in self.layers:
      output = mod(output, memory, cross_attn_mask=cross_attn_mask, \
                   decoder_self_attn_mask=decoder_self_attn_mask)

    output = self.norm(output)
    return output


class TransformerDecoderLayer(nn.Module):
  r"""TransformerDecoderLayer is made up of self-attn, cross-attn, and
  feedforward network.
  Arguments:
      hidden_size: hidden size.
  """

  def __init__(self, hidden_size):
    super(TransformerDecoderLayer, self).__init__()
    self.hidden_size = hidden_size
    fwd_hidden_size = hidden_size * 4

    # Create modules
    self.linear1 = nn.Linear(hidden_size, fwd_hidden_size)
    self.linear2 = nn.Linear(fwd_hidden_size, hidden_size)

    self.activation = nn.ReLU()

    self.norm1 = nn.LayerNorm(hidden_size)
    self.norm2 = nn.LayerNorm(hidden_size)
    self.norm3 = nn.LayerNorm(hidden_size)

    # Attention related
    self.q_proj_self = nn.Linear(hidden_size, hidden_size)
    self.k_proj_self = nn.Linear(hidden_size, hidden_size)
    self.v_proj_self = nn.Linear(hidden_size, hidden_size)
    self.context_proj_self = nn.Linear(hidden_size, hidden_size)
```

```
        self.q_proj_cross = nn.Linear(hidden_size, hidden_size)
        self.k_proj_cross = nn.Linear(hidden_size, hidden_size)
        self.v_proj_cross = nn.Linear(hidden_size, hidden_size)
        self.context_proj_cross = nn.Linear(hidden_size, hidden_size)

    def forward(self, tgt, memory, cross_attn_mask, decoder_self_attn_mask):
        r"""Pass the inputs (and mask) through the decoder layer.
        Arguments:
            tgt: an input tensor of size (max_tgt_len, bsz, hidden_size).
            memory: encoder outputs of size (max_src_len, bsz, hidden_size).
            cross_attn_mask: attention mask of size (bsz, max_tgt_len,␣
↪max_src_len),
                                 it's `False` where the cross-attention is disallowed.
            decoder_self_attn_mask: attention mask of size (bsz, max_tgt_len,␣
↪max_tgt_len),
                                     it's `False` where the self-attention is␣
↪disallowed.
        Returns:
            a tensor of size (max_tgt_len, bsz, hidden_size)
        """
        # Self attention (decoder-side)
        q = self.q_proj_self(tgt) / math.sqrt(self.hidden_size)
        k = self.k_proj_self(tgt)
        v = self.v_proj_self(tgt)
        #TODO - compute `context`
        context = ...
        tgt2 = self.context_proj_self(context)
        tgt = tgt + tgt2
        tgt = self.norm1(tgt)
        # Cross attention (decoder attends to encoder)
        q = self.q_proj_cross(tgt) / math.sqrt(self.hidden_size)
        k = self.k_proj_cross(memory)
        v = self.v_proj_cross(memory)
        #TODO - compute `context`
        context = ...
        tgt2 = self.context_proj_cross(context)
        tgt = tgt + tgt2
        tgt = self.norm2(tgt)
        tgt2 = self.linear2(self.activation(self.linear1(tgt)))
        tgt = tgt + tgt2
        tgt = self.norm3(tgt)
        return tgt

class PositionalEmbedding(nn.Module):
    """"Embeds a word both by its word id and by its position in the sentence."""
    def __init__(self, vocab_size, embedding_size, max_len=1024):
```

```
    super(PositionalEmbedding, self).__init__()
    self.embedding_size = embedding_size

    self.embed = nn.Embedding(vocab_size, embedding_size)
    pe = torch.zeros(max_len, embedding_size)
    position = torch.arange(0, max_len).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, embedding_size, 2) *
                         -(math.log(10000.0) / embedding_size))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    pe = pe.unsqueeze(1) # max_len, 1, embedding_size
    self.register_buffer('pe', pe)

  def forward(self, batch):
    x = self.embed(batch) * math.sqrt(self.embedding_size) # type embedding
    # Add positional encoding to type embedding
    x = x + self.pe[:x.size(0)].detach()
    return x


def _get_clones(module, N):
  """Copies a module `N` times"""
  return nn.ModuleList([copy.deepcopy(module) for i in range(N)])
```

[ ]:
```
EPOCHS = 2 # epochs, we highly recommend starting with a smaller number like 1
LEARNING_RATE = 2e-3 # learning rate

# Instantiate and train classifier
model_transformer = TransformerEncoderDecoder(SRC, TGT,
  hidden_size    = 64,
  layers         = 3,
).to(device)

model_transformer.train_all(train_iter, val_iter, epochs=EPOCHS,␣
 ↪learning_rate=LEARNING_RATE)
model_transformer.load_state_dict(model_transformer.best_model)
```

You might notice that in these experiments training transformers doesn't appear to be faster than training RNNs. There are two reasons for that: first, we are not using GPUs; second, even if you use GPUs, the sequences here are too short to observe the benefits of parallelizing along the horizontal direction. In real datasets with long sentences, training transformers is much faster than training RNNs, so under the same computational budget, using transformers allows for training on much larger datasets. This is one of the primary reasons transformers dominate NLP research these days.

Question: Would there be any speed advantage of decoding (generation) using transformers compared to RNNs? Why or why not?

Type your answer here, replacing this text.

[ ]:
```
# Evaluate model performance, the expected value should be < 1.5
print (f'Test perplexity: {model_transformer.evaluate_ppl(test_iter):.3f}')
```

[ ]:
```
grader.check("transformer_ppl")
```

Now that we have a trained model, we can decode from it using our previously implemented beam search function. If the code below throws any errors, you might need to modify your beam search code such that it generalizes here.

[ ]:
```
grader.check("transformer_beam_search")
```

[ ]:
```
DEBUG_FIRST = 10 # set to False to disable printing predictions
K = 1 # beam size 1

correct = 0
total = 0

# create beam searcher
beam_searcher = BeamSearcher(model_transformer)

for index, batch in enumerate(test_iter, start=1):
  # Input and output
  src, src_lengths = batch.src
  # Predict
  model.all_attns = []
  prediction, _ = beam_searcher.beam_search(src, src_lengths, K)
  # Convert to string
  prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
  prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()
  ground_truth = ' '.join([TGT.vocab.itos[token] for token in batch.tgt.
↪view(-1)])
  ground_truth = ground_truth.lstrip('<bos>').rstrip('<eos>').strip()
  if DEBUG_FIRST > index:
    src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
    print (f'Source: {src}')
    print (f'Prediction:   {prediction}')
    print (f'Ground truth: {ground_truth}')
  if ground_truth == prediction:
    correct += 1
  total += 1

print (f'Accuracy: {correct/total:.2f}')
```

Question: When we first introduced attention above, adding it to an RNN model, we noted that

> The attention scores **a** lie on a *simplex* (meaning $a_i \geq 0$ and $\sum_i a_i = 1$), which lends it some interpretability: the closer a_i is to 1, the more "relevant" a key k_i (and hence

its value v_i) is to the given query. We will observe this later in the lab: When we are about to predict the target word “3”, a_i is close to 1 for the source word $x_i =$ ”three”.

Can we interpret the attentions in a multi-layer transformer similarly? If so, what would you expect the attention scores to correspond to? If not, explain why.

Type your answer here, replacing this text.

You might have noticed that the transformer model underperforms the RNN-based encoder-decoder on this particular task. This might be due to several reasons:

- Transformers tend to be data hungry, sometimes requiring billions of words to train.
- The transformer formulation presented in this lab is not in its full form: for instance, instead of only doing attention once at each position for each layer, researchers usually use multiple attention operations in the hope of capturing different aspects of “relevance”, which is called “multi-headed attention”. For example, one attention head might be focusing on pronoun resolution, while the other might be looking for similar contexts before.
- Transformers are usually sensitive to hyper-parameters and require heavy tuning. For example, while we used a fixed learning rate, researchers usually use a customized learning rate scheduler which first warms up the learning rate, and then gradually decreases it. If you are interested, more details can be found in the original paper.

In real-world applications, many state-of-the-art NLP approaches are based on transformers, such as the fake news generator used by GROVER that you’ve seen in the Embedded EthiCS class. For further readings if you are interested, we recommend BERT and GPT-3.

4 Lab debrief – for consensus submission only

Question: We’re interested in any thoughts your group has about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on might include the following:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

but you should comment on whatever aspects you found especially positive or negative.

Type your answer here, replacing this text.

End of Lab 4-5

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```