CS187 Lab 4-4: Sequence-to-sequence models

September 9, 2023

[ ]:
```python
# 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/lab4-4.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

In lab 4-2, we used a syntactic-semantic grammar for semantic parsing to convert natural language to meaning representations in SQL. In this lab, we consider an alternative approach, sequence-to-sequence models, which can solve this task by directly learning the mapping from a sequence of inputs to a sequence of outputs. Since sequence-to-sequence models make few assumptions about the data, they can be applied to a variety of tasks, including machine translation, document summarization, and speech recognition.

In this lab, we will implement a sequence-to-sequence model in its most basic form (as in this seminal paper), and apply it to the task of converting English number phrases to numbers, as exemplified in the table below.

Input	Output
seven thousand nine hundred and twenty nine	7929
eight hundred and forty two thousand two hundred and fifty nine	842259
five hundred and eight thousand two hundred and seventeen	508217

For this simple task, it is possible to write a rule-based program to do the conversion. However, here we take a learning-based approach and learn the mapping from demonstrations, with the benefit that the system we implement here can be applied to other sequence-to-sequence tasks as well (including the ATIS-to-SQL problem in project segment 4).

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

- torch.transpose: Swaps two dimensions of a tensor.
- torch.reshape: Redistributes the elements of a tensor to form a tensor of a different shape, e.g., from 3 x 4 to 6 x 2.
- torch.nn.utils.rnn.pack_padded_sequence (imported as `pack`): Handles paddings. A more detailed explanation can be found here.

Preparation - Loading data

[ ]:
```
import copy
import math
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
```

[ ]:
```
# 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)
```

1 The dataset

Let's take a first look at the dataset of English number phrases and their translations into digit-sequence form.

```
[ ]: with open(local_dir + "dev.src") as fsrc:
    with open(local_dir + "dev.tgt") as ftgt:
        print (f'{"Source":70s} {"Target":>12s}')
        for src, tgt, _ in zip(fsrc, ftgt, range(3)):
            print (f'{src.strip():70s} {tgt.strip():>12s}')
```

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).

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

Note that we prepended `<bos>` and appended `<eos>` to target sentences. The purpose for introducing them will become clear in later parts of this lab.

```
[ ]: # 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 the 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]])}")
```

2 Neural Encoder-Decoder Models

Sequence-to-sequence models are sometimes called neural encoder-decoder models, as they consist of an encoder, which maps a sequence of source tokens into some vector representations, and a decoder, which generates a sequence of output words from those encoded vectors.

Formally, given a sequence of source tokens $\mathbf{x} = x_1, \ldots, x_S$, the goal is to map it to a sequence of target tokens $\mathbf{y} = y_1, \ldots, y_T$.

In practice, we prepend a special beginning-of-sequence symbol y_0=`<bos>` to the target sequence. Further, in order to provide a way of knowing when to stop generating $\mathbf{y}$, we append a special end-of-sequence symbol y_{T+1}=`<eos>` to the target sequence, such that when it is produced by the model, the generation process stops.

The generation process is structured as a generative model:

$$\Pr(y_0, \ldots, y_{T+1} \mid x_1, \ldots, x_S) = \prod_{t=1}^{T+1} \Pr(y_t \mid y_{<t}, x_1, \ldots, x_S),$$

where $y_{<t}$ denotes the tokens before y_t (that is, $y_0, \ldots, y_{t-1}$).

We use a recurrent neural network with parameters θ to parameterize $\Pr(y_t \mid y_{<t}, x_1, \ldots, x_S)$:

$$\Pr(y_t \mid y_{<t}, x_1, \ldots, x_S) \approx \Pr_\theta(y_t \mid y_{<t}, x_1, \ldots, x_S),$$

or equivalently,

$$\Pr_\theta(y_1, \ldots, y_T \mid x_1, \ldots, x_S) = \prod_{t=1}^{T+1} \Pr_\theta(y_t \mid y_{<t}, x_1, \ldots, x_S)$$

In neural encoder-decoder models, we first use an encoder to encode $\mathbf{x}$ into some vectors (either of fixed length as we'll see in this lab, or of varying length as we'll see in the next lab). Based on the encoded vectors, we use a decoder to generate $\mathbf{y}$:

$$\Pr_\theta(y_t \mid y_{<t}, x_1, \ldots, x_S) = \text{decode}(\text{encode}(x_1, \ldots, x_S), y_{<t})$$

2.0.1 RNN Encoder-Decoders

We can use any recurrent neural networks such as LSTMs as encoders and decoders. In this lab, we will use a bidirectional LSTM as the encoder, and a unidirectional LSTM as the decoder, as shown in the illustration below.

In the above illustration, $S = 4$, $T = 3$, and there are two encoder/decoder layers. Since we are using a bidirectional encoder, for each layer there are two final states, one for the cell running from left to right (such as $h_{0,4}$), and the other for the cell running from right to left (such as $h'_{0,4}$). We concatenate these two states and use the result to initialize the corresponding layer of the decoder. (In the example, we concatenate $h_{0,4}$ and $h'_{0,4}$ to initialize layer 0, and we concatenate $h_{1,4}$ and $h'_{1,4}$ to initialize layer 1.) Therefore, to make the sizes match, we set the hidden state size of the encoder to be half of that of the decoder.

Note that in PyTorch's LSTM implementation, the final hidden state is represented as a tuple `(h, c)` (documentation here), so we want to apply the same operations to `c` to initialize the decoder.

You'll implement `forward_encoder` and `forward_decoder` in the code below. The `forward_encoder` function will be reminiscent of a sequence model from labs 2-* and project segment 2. It operates on a batch of source examples and proceeds as follows:

1. Map the input words to some word embeddings. You'll notice that the embedding size is an argument to the model.
2. Optionally "pack" the sequences to save some computation using `torch.nn.utils.rnn.pack_padded_sequence`, imported above as `pack`.
3. Run the encoder RNN (a bidirectional LSTM) over the batch, generating a batch of output states.
4. Reshape the final state information (which will have `h` and `c` components each of half the size needed to initialize the decoder) so that it is appropriate to initialize the decoder with.

The `forward_decoder` function takes the reshaped encoder final state information and the ground truth target sequences and returns logits (unnormalized log probs) for each target word. (These are ready to be converted to probability distributions via a softmax.)

The steps in decoding are:

1. Map the target words to word embeddings.
2. Run the decoder RNN (a unidirectional LSTM) over the batch, initializing the hidden units from the encoder final states, generating a batch of output states.
3. Map the RNN outputs to vectors of vocabulary size (so that they could be softmaxed into a distribution over the vocabulary).

The components that you'll be plugging together to do all this are already established in the `__init__` method.

> The major exception is the reshaping of the encoder output `h` and `c` to form the decoder input `h` and `c`. **This is the trickiest part.** As usual, your best strategy is to keep careful track of the shapes of each input and output of a layer or operation. We

recommend that you try out just the reshaping code on small sample data to test it out before running any encodings or decodings.

Hint: The total number of `for` loops in our solution code for the parts you are to write is…zero.

[ ]:
```python
# TODO - finish implementing the `forward_encoder` and `forward_decoder` methods
class EncoderDecoder(nn.Module):
    def __init__(self, src_field, tgt_field, embedding_size=64, hidden_size=64,␣
 ↪layers=3):
        """
        Initializer. Creates network modules and loss function.
        Arguments:
            src_field: src field
            tgt_field: tgt field
            embedding_size: word embedding size
            hidden_size: hidden layer size of both encoder and decoder
            layers: number of layers of both encoder and decoder
        """
        super(EncoderDecoder, 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 or tokens
        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 = embedding_size
        self.hidden_size = hidden_size
        self.layers = layers

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

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

```python
        self.decoder_rnn = nn.LSTM(
            input_size=embedding_size,
            hidden_size=hidden_size,
            num_layers=layers,
            bidirectional=False,  # unidirectional decoder
        )

        # 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, batch_size)
            src_lengths: src lengths of size (batch_size)
        Returns:
            a tuple (h, c) where h/c is of size (layers, bsz, hidden_size)
        """
        # TODO - implement this function
        # Optional: use `pack` to deal with paddings (https://discuss.pytorch.
↪org/t/
↪simple-working-example-how-to-use-packing-for-variable-length-sequence-inputs-for-rnn/
↪2120)
        ...
        ...

    def forward_decoder(self, encoder_final_state, tgt_in):
        """
        Decodes based on encoder final state and ground truth target words.
        Arguments:
            encoder_final_state: a tuple (h, c) where h/c is of size (layers,␣
↪bsz, hidden_size)
            tgt_in: a tensor of size (tgt_len, bsz)
        Returns:
            Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
        """
        # TODO - implement this function
        ...
        ...

    def forward(self, src, src_lengths, tgt_in):
```

```
        """
        Performs forward computation, returns logits.
        Arguments:
            src: src batch of size (max_src_len, batch_size)
            src_lengths: src lengths of size (batch_size)
            tgt_in:  a tensor of size (tgt_len, bsz)
        """
        # Forward encoder
        encoder_final_state = self.forward_encoder(src, src_lengths)  # tuple␣
↪(h, c)
        # Forward decoder
        logits = self.forward_decoder(encoder_final_state, tgt_in)
        return logits

    def forward_decoder_incrementally(self, decoder_state, tgt_in_token):
        """
        Forward the decoder at `decoder_state` for a single step with token␣
↪`tgt_in_token`.
        This function will only be used in the beam search section.
        Arguments:
            encoder_final_state: a tuple (h, c) where h/c is of size (layers,␣
↪1, hidden_size)
            tgt_in_token: a tensor of size (1), a single token
        Returns:
            `logits`: Log probabilities for `tgt_in_token` of size (V_tgt)
            `decoder_state`: updated decoder state, ready for next incremental␣
↪update
        """
        bsz = decoder_state[0].size(1)
        assert bsz == 1, "forward_decoder_incrementally only supports batch␣
↪size 1!"
        # Compute word embeddings
        tgt_embeddings = self.word_embeddings_tgt(
            tgt_in_token.view(1, 1)
        )  # tgt_len, bsz, hidden
        # Forward decoder RNN and return all hidden states
        decoder_outs, decoder_state = self.decoder_rnn(tgt_embeddings,␣
↪decoder_state)
        # Project to get logits
        logits = self.hidden2output(decoder_outs)  # tgt_len, bsz, V_tgt
        # Get log probabilities
        logits = torch.log_softmax(logits, -1)
        return logits.view(-1), decoder_state

    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>)
                batch_size = 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(batch_size).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 = EncoderDecoder(
    SRC,
    TGT,
    embedding_size=64,
    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
print (f'Test perplexity: {model.evaluate_ppl(test_iter):.3f}')
```

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

3 Consensus section – Decoding

> This section is a bit more complex, and needs only to be submitted for the consensus submission.

Now that we have a well-trained model, we need to consider how to use it to do the actual conversion. At decoding time, given a source sequence $x_1, \dots, x_S$, we want to find the target sequence $y_1^*, \dots, y_T^*, y_{T+1}^*$ (recall that $y_{T+1} = \langle\text{eos}\rangle$) such that the conditional likelihood is maximized:

$$\begin{aligned} y_1^*, \dots, y_T^*, y_{T+1}^* &= \underset{y_1,\cdots,y_T,y_{T+1}}{\operatorname{argmax}} \Pr_\theta(y_1, \dots, y_T \mid x_1, \dots, x_S) \\ &= \underset{y_1,\cdots,y_T,y_{T+1}}{\operatorname{argmax}} \prod_{t=1}^{T+1} \Pr_\theta(y_t \mid y_{<t}, x_1, \dots, x_S) \end{aligned}$$

In previous labs and project segments, we used *greedy decoding*, i.e., taking $\hat{y}_1 = \underset{y_1}{\operatorname{argmax}} \Pr_\theta(y_1 \mid y_0, x_1, \dots, x_S)$, $\hat{y}_2 = \underset{y_2}{\operatorname{argmax}} \Pr_\theta(y_2 \mid y_0, \hat{y}_1, x_1, \dots, x_S)$, ..., $\hat{y}_{T+1} = \underset{y_{T+1}}{\operatorname{argmax}} \Pr_\theta(y_{T+1} \mid y_0, \hat{y}_1, \dots, \hat{y}_T, x_1, \dots, x_S)$, until $\hat{y}_{T+1} = \langle\text{eos}\rangle$.

Question: Does greedy decoding guarantee finding the optimal sequence (the sequence with the highest conditional likelihood)? Why or why not?

Type your answer here, replacing this text.

3.1 Beam search decoding

Beam search decoding is the most commonly used decoding method in sequence-to-sequence approaches. Like greedy decoding, it uses a left-to-right search process. But instead of only keeping the single argmax at each position, beam search maintains the K best partial hypotheses $H_t = \{(y_1^{(k)}, \dots, y_t^{(k)}) : k \in \{1, \dots, K\}\}$ at every step t. To proceed to $t+1$, we compute the scores of sequences $y_1^{(k)}, \dots, y_t^{(k)}, y_{t+1}$ for every possible extension $y_{t+1} \in \mathcal{V}$ and every possible prefix $(y_1^{(k)}, \dots, y_t^{(k)}) \in H_t$, where $\mathcal{V}$ is the vocabulary. Among these $K \times |\mathcal{V}|$ sequences, we only keep the top K sequences with the best partial scores, and that becomes $H_{t+1} = \{(y_1^{(k)}, \dots, y_{t+1}^{(k)}) : k \in \{1, \dots, K\}\}$. To start at $t = 1$, $H_1 = \{(y) : y \in \text{K-argmax}_{y_1 \in \mathcal{V}} \log P(y_1 | y_0 = bos)\}$. Here K is called the beam size.

To summarize,

$$\begin{aligned} H_1 &= \{(y) : y \in \underset{y_1 \in \mathcal{V}}{\text{K-argmax}} \log P(y_1 | y_0 = bos)\} \\ H_{t+1} &= \underset{\{(y_1, y_2, \dots, y_{t+1}) \in \mathcal{V}^{t+1} : (y_1, \dots, y_t) \in H_t\}}{\text{K-argmax}} \log P(y_1, \dots, y_{t+1} | x) \end{aligned}$$

until we reach a pre-specified maximum search length, and we collect the completed hypotheses along the way. (By completed we mean ending with `<eos>`.) The finished hypothesis with the best score will then be returned.

Question: Is beam search better than greedy search when $K = 1$? Is it better when $K > 1$? Why? How big a K value do we need to get a guarantee that we can find the globally best sequence (assuming a maximum sequence length T and vocabulary size $|\mathcal{V}|$).

Type your answer here, replacing this text.

Under the probabilistic formulation of sequence-to-sequence models, the partial scores are decomposable over time steps: $\log \Pr_\theta(y_1, \ldots, y_T \mid x) = \sum_{t=1}^{T} \log \Pr_\theta(y_t \mid y_{<t}, x)$. Therefore, we can save computation in the above process by maintaining the partial sums $\sum_{t'=1}^{t} \log \Pr_\theta(y_{t'}^{(k)} \mid y_{<t'}^{(k)}, x)$, such that we only need to compute $\log \Pr_\theta(y_{t+1} \mid y_{<t+1}^{(k)})$ when we want to go from t to $t+1$.

Here is pseudo-code for the beam search algorithm to decode a single example `x` of maximum length `max_T` using a beam size of `K`.

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

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

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

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

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

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

Implement function `beam_search` in the below code. Note that there are some differences from the pseudo-code: first, we maintained a `decoder_state` in addition to $y_{1:t}$ and score such that we can compute $P(y_{t+1} \mid y_{<t+1}, x)$ efficiently; second, instead of creating a list of actual hypotheses as in lines 8-11, we use tensors to get pointers to the beam id and y_{t+1} that are among the best K next beams.

[ ]:
```python
MAX_T = 15     # max target length

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.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
    """
    finished = []
    # Initialize the beam
    self.model.eval()
    #TODO - fill in encoder_final_state and init_beam below
    encoder_final_state = ...
    init_beam = ...
    beams = [init_beam]

    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 = []
```

```python
        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`
            ...
            decoder_state = ...
            total_scores = ...
            all_total_scores.append(total_scores)
            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 below code 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
    else: # when nothing is finished, return an unfinished hypothesis
        return beams[0].tokens
```

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

Now we can use beam search decoding to predict the outputs for the test set inputs using the trained model.

[ ]:
```
DEBUG_FIRST = 10  # set to 0 to disable printing predictions
K = 5             # beam size 5

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()
  # Print out the first few examples
  if DEBUG_FIRST >= index :
    src = ' '.join([SRC.vocab.itos[item] for item in src.view(-1)])
    print (f'Source:       {index}. {src}\n'
           f'Prediction:   {prediction}\n'
           f'Ground truth: {ground_truth}\n')
  if ground_truth == prediction:
    correct += 1
  total += 1

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

You might have noticed that using a larger K might lead to very similar performance as using $K = 1$ (greedy decoding). This is largely due to the fact that there are no dependencies among target tokens in our dataset (e.g., knowing that y_1 is 1 does not affect our prediction on y_2 conditioned on the source). In real world applications, people usually find using a fixed value of $K > 1$ (such as $K = 5$) performs better than greedy decoding.

Question: Can we use beam search decoding to decode an HMM? For state space Q, sequence length T, what is the complexity of beam search with beam size K? What is the complexity of Viterbi decoding? What are their pros and cons?

Type your answer here, replacing this text.

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 4-4

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

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