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

CS187

Lab 4-4 - Sequence-to-sequence models

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

# Only install packages if not in cs187-env
if 'cs187-env' not in os.environ.get('CONDA_PREFIX', ''):
    import subprocess
    import sys
    subprocess.run([sys.executable, '-m', 'pip', 'install', '-q', '-r', 'rec
                   check=True)
```

In lab 4-2, you 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, you 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

In [ ]:
```
import copy
import csv
import math
import os
import random
import sys
import wget

import torch
import torch.nn as nn

from datasets import load_dataset
from itertools import islice

from tokenizers import Tokenizer
from tokenizers.pre_tokenizers import WhitespaceSplit
from tokenizers.processors import TemplateProcessing
from tokenizers import normalizers
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
from transformers import PreTrainedTokenizerFast

from tqdm.notebook import tqdm

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

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

In [ ]:
```
# Set random seeds for reproducibility
SEED = 1234

def reseed():
    torch.manual_seed(SEED)
    random.seed(SEED)

reseed()
```

In [ ]:
```
# Prepare to download needed data
def download_if_needed(source, dest, filename, add_to_path=True):
    os.makedirs(dest, exist_ok=True)  # ensure destination
    if add_to_path:
        sys.path.insert(1, dest)      # add local to path
    if os.path.exists(f"./{dest}{filename}"):
```

```
        print(f"Skipping {filename}")
    else:
        print(f"Downloading {filename} from {source}")
        wget.download(source + filename, out=dest)
        print("", flush=True)


remote_dir = "https://github.com/nlp-course/data/raw/refs/heads/master/Words
local_dir = "./data/"

for filename in [
    "train.src",
    "train.tgt",
    "dev.src",
    "dev.tgt",
    "test.src",
    "test.tgt",
]:
    download_if_needed(remote_dir, local_dir, filename)
```

Next, we process the dataset by extracting the sequences and their corresponding labels and save it in CSV format.

In [ ]:
```
# Process data
for split in ['train', 'dev', 'test']:
    src_in_file = f'{local_dir}{split}.src'
    tgt_in_file = f'{local_dir}{split}.tgt'
    out_file = f'{local_dir}{split}.csv'

    with open(src_in_file, 'r') as f_src_in, open(tgt_in_file, 'r') as f_tgt
        with open(out_file, 'w') as f_out:
            src, tgt= [], []
            writer = csv.writer(f_out)
            writer.writerow(('src','tgt'))
            for src_line, tgt_line in zip(f_src_in, f_tgt_in):
                writer.writerow((src_line.strip(), tgt_line.strip()))
```

The dataset

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

In [ ]:
```
with open(local_dir + "dev.csv") as f:
    for line, _ in zip(f, range(3)):
        src, tgt = line.split(',')
        print (f'{src.strip():70s} {tgt.strip():>12s}')
```

As before, we use HuggingFace's `datasets` 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).

```
In [ ]: dataset = load_dataset(
            "csv",
            data_files={
                "train": f"{local_dir}train.csv",
                "val": f"{local_dir}dev.csv",
                "test": f"{local_dir}test.csv",
            },
        )
        dataset
```

```
In [ ]: train_data = dataset['train']
        val_data = dataset['val']
        test_data = dataset['test']
```

```
In [ ]: unk_token = '[UNK]'
        pad_token = '[PAD]'
        bos_token = '<bos>'
        eos_token = '<eos>'
        src_tokenizer = Tokenizer(WordLevel(unk_token=unk_token))
        src_tokenizer.pre_tokenizer = WhitespaceSplit()

        src_trainer = WordLevelTrainer(special_tokens=[pad_token, unk_token])
        src_tokenizer.train_from_iterator(train_data['src'], trainer=src_trainer)

        tgt_tokenizer = Tokenizer(WordLevel(unk_token=unk_token))
        tgt_tokenizer.pre_tokenizer = WhitespaceSplit()

        tgt_trainer = WordLevelTrainer(special_tokens=[pad_token, unk_token, bos_tok

        tgt_tokenizer.train_from_iterator(train_data['tgt'], trainer=tgt_trainer)

        tgt_tokenizer.post_processor = \
          TemplateProcessing(single=f"{bos_token} $A {eos_token}",
                             special_tokens=[(bos_token,
                                              tgt_tokenizer.token_to_id(bos_token)),
                                             (eos_token,
                                              tgt_tokenizer.token_to_id(eos_token))]
```

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.

We use `datasets.Dataset.map` to convert text into word ids. As shown in lab 1-5, first we need to wrap `tokenizer` with the `transformers.PreTrainedTokenizerFast` class to be compatible with the `datasets` library.

```
In [ ]: hf_src_tokenizer = PreTrainedTokenizerFast(tokenizer_object=src_tokenizer,
                                                   pad_token=pad_token,
                                                   unk_token=unk_token)
        hf_tgt_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tgt_tokenizer,
                                                   pad_token=pad_token,
                                                   unk_token=unk_token,
```

```
                                                  bos_token=bos_token,
                                                  eos_token=eos_token)
```

In [ ]:
```
def encode(example):
    example['src_ids'] = hf_src_tokenizer(example['src']).input_ids
    example['tgt_ids'] = hf_tgt_tokenizer(example['tgt']).input_ids
    return example

train_data = train_data.map(encode)
val_data = val_data.map(encode)
test_data = test_data.map(encode)
```

In [ ]:
```
# Compute size of vocabularies
src_vocab = hf_src_tokenizer.get_vocab()
tgt_vocab = hf_tgt_tokenizer.get_vocab()

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[pad_token]}")
print(f"Index for tgt padding: {tgt_vocab[pad_token]}")
print(f"Index for start of sequence token: {tgt_vocab[bos_token]}")
print(f"Index for end of sequence token: {tgt_vocab[eos_token]}")
```

To load data in batched tensors, we use `torch.utils.data.DataLoader` for data splits, which enables us to iterate over the dataset under a given `BATCH_SIZE`. For the test set, we use a batch size of 1, to make the decoding implementation easier.

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

# Defines how to batch a list of examples together
def collate_fn(examples):
    batch = {}
    bsz = len(examples)
    src_ids, tgt_ids = [], []
    for example in examples:
        src_ids.append(example['src_ids'])
        tgt_ids.append(example['tgt_ids'])

    src_len = torch.LongTensor([len(word_ids) for word_ids in src_ids]).to(c
    src_max_length = max(src_len)
    tgt_max_length = max([len(word_ids) for word_ids in tgt_ids])

    src_batch = torch.zeros(bsz, src_max_length).long().fill_(src_vocab[pad_
    tgt_batch = torch.zeros(bsz, tgt_max_length).long().fill_(tgt_vocab[pad_
    for b in range(bsz):
        src_batch[b][:len(src_ids[b])] = torch.LongTensor(src_ids[b]).to(dev
        tgt_batch[b][:len(tgt_ids[b])] = torch.LongTensor(tgt_ids[b]).to(dev

    batch['src_lengths'] = src_len
    batch['src_ids'] = src_batch
    batch['tgt_ids'] = tgt_batch
    return batch
```

```python
train_iter = torch.utils.data.DataLoader(train_data,
                                         batch_size=BATCH_SIZE,
                                         shuffle=True,
                                         collate_fn=collate_fn)
val_iter = torch.utils.data.DataLoader(val_data,
                                       batch_size=BATCH_SIZE,
                                       shuffle=False,
                                       collate_fn=collate_fn)
test_iter = torch.utils.data.DataLoader(test_data,
                                        batch_size=TEST_BATCH_SIZE,
                                        shuffle=False,
                                        collate_fn=collate_fn)
```

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

In [ ]:
```python
batch = next(iter(train_iter))
src_ids = batch['src_ids']
src_example = src_ids[2]
print (f"Size of src batch: {src_ids.size()}")
print (f"Third src sentence in batch: {src_example}")
print (f"Length of the third src sentence in batch: {len(src_example)}")
print (f"Converted back to string: {hf_src_tokenizer.decode(src_example)}")

tgt_ids = batch['tgt_ids']
tgt_example = tgt_ids[2]
print (f"Size of tgt batch: {tgt_ids.size()}")
print (f"Third tgt sentence in batch: {tgt_example}")
print (f"Converted back to string: {hf_tgt_tokenizer.decode(tgt_example)}")
```

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:

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

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

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

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

or equivalently,

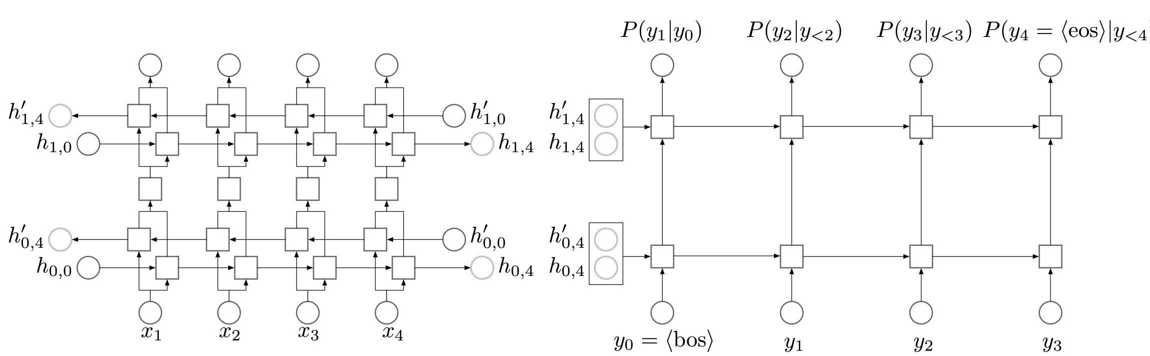
$$\Pr_\theta(y_1, \dots, y_T \mid x_1, \dots, x_S) = \prod_{t=1}^{T+1} \Pr_\theta(y_t \mid y_{<t}, x_1, \dots, x_S)$$

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 $\mathbf{y}$:

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

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 #1: We've provided an auxiliary notebook, called `lab4-4-reshaping.ipynb`, that discusses the reshaping issue in some detail. You'll want to look it over.

Hint #2: The total number of `for` loops in our solution code for the parts you are to write is...zero.

Hint #3: According to the documentation of torch.nn.LSTM, its outputs are: `outputs, (h, c)`. `outputs` contains all the intermediate states, which you don't need in this lab. You will need `h` and `c`; both of them have the shape: `(num_layers * num_directions, batch_size, hidden_size)`.

In [ ]:
```
# TODO - finish implementing the `forward_encoder` and `forward_decoder` met
class EncoderDecoder(nn.Module):
    def __init__(
        self,
        hf_src_tokenizer,
        hf_tgt_tokenizer,
        embedding_size=64,
        hidden_size=64,
        layers=3,
    ):
        """
        Initializer. Creates network modules and loss function.
        Arguments:
            hf_src_tokenizer: src field information
            hf_tgt_tokenizer: tgt field information
            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.hf_src_tokenizer = hf_src_tokenizer
        self.hf_tgt_tokenizer = hf_tgt_tokenizer

        # Keep the vocabulary sizes available
        self.V_src = len(hf_src_tokenizer)
        self.V_tgt = len(hf_tgt_tokenizer)

        # Get special word ids or tokens
        self.padding_id_src = self.hf_src_tokenizer.pad_token_id
        self.padding_id_tgt = self.hf_tgt_tokenizer.pad_token_id
        self.bos_id = self.hf_tgt_tokenizer.bos_token_id
        self.eos_id = self.hf_tgt_tokenizer.eos_token_id

        # 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)
```

```python
        # RNN cells
        self.encoder_rnn = nn.LSTM(
            input_size=embedding_size,
            hidden_size=hidden_size // 2,  # to match decoder hidden size
            batch_first=True,
            num_layers=layers,
            bidirectional=True,  # bidirectional encoder
        )
        self.decoder_rnn = nn.LSTM(
            input_size=embedding_size,
            hidden_size=hidden_size,
            batch_first=True,
            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 (batch_size, max_src_len)
            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://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_p
        # Note that the batch size is the first dimension, and
        # the sequences are not sorted.
        ...
        ...

    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
                                 (bsz, layers, hidden_size)
            tgt_in: a tensor of size (tgt_len, bsz)
        Returns:
            Logits of size (tgt_len, bsz, V_tgt) (before the softmax operati
        """
        # TODO - implement this function
        ...
        ...

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

```
        """
        Performs forward computation, returns logits.
        Arguments:
            src: src batch of size (batch_size, max_src_len)
            src_lengths: src lengths of size (batch_size)
            tgt_in:  a tensor of size (batch_size, tgt_len)
        """
        # Forward encoder
        encoder_final_state = self.forward_encoder(src, src_lengths)  # tupl
        # Forward decoder
        logits = self.forward_decoder(
            encoder_final_state, tgt_in
        )  # bsz, tgt_len, V_tgt
        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:
            decoder_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
        # Compute word embeddings
        tgt_embeddings = self.word_embeddings_tgt(
            tgt_in_token.view(1, 1)
        )  # bsz, tgt_len, hidden
        # Forward decoder RNN and return all hidden states
        decoder_outs, decoder_state = self.decoder_rnn(tgt_embeddings, decod
        # Project to get logits
        logits = self.hidden2output(decoder_outs)  # bsz, tgt_len, V_tgt
        # Get log probabilities
        logits = torch.log_softmax(logits, -1)
        return logits.view(-1), decoder_state

    def evaluate_ppl(self, iterator, disable_progress=False):
        """Returns the model's perplexity on a given dataset `iterator`. If
        disable the progress bars."""
        # Switch to eval mode
        self.eval()
        total_loss = 0
        total_words = 0
        for batch in tqdm(iterator, desc='Eval batch', leave=False, disable=
            # Input and target
            src = batch["src_ids"]  # bsz, max_src_len
            src_lengths = batch["src_lengths"]  # bsz
```

```
            # Remove <eos> for decoder input (e.g., y_0=<bos>, y_1, y_2)
            tgt_in = batch["tgt_ids"][:, :-1].contiguous()
            # Remove <bos> as decoder output (e.g., y_1, y_2, y_3=<eos>)
            tgt_out = batch["tgt_ids"][:, 1:].contiguous()
            # Forward to get logits
            logits = self.forward(src, src_lengths, tgt_in)  # bsz, tgt_len,
            # Compute cross entropy loss
            loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.v
            total_loss += loss.item()
            total_words += tgt_out.ne(self.padding_id_tgt).float().sum().ite
        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 tqdm(range(epochs), desc='Epoch'):
            total_words = 0
            total_loss = 0.0
            for batch in tqdm(train_iter, desc='Train batch', leave=False):
                # Zero the parameter gradients
                self.zero_grad()
                # Input and target
                tgt = batch["tgt_ids"]  # bsz, max_tgt_len
                src = batch["src_ids"]  # bsz, max_src_len
                src_lengths = batch["src_lengths"]  # bsz
                # Remove <eos> for decoder input (e.g., y_0=<bos>, y_1, y_2)
                tgt_in = tgt[:, :-1]
                # Remove <bos> as decoder output (e.g., y_1, y_2, y_3=<eos>)
                tgt_out = tgt[:, 1:]
                batch_size = src.size(0)
                # Run forward pass and compute loss along the way.
                logits = self.forward(src, src_lengths, tgt_in)  # bsz, tgt_
                loss = self.loss_function(
                    logits.reshape(-1, self.V_tgt), tgt_out.reshape(-1)
                )
                # Training stats
                num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum(
                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):
                f"Validation Perplexity: {validation_ppl:.4f}"
            )
```

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

# Instantiate and train classifier
wide_model = EncoderDecoder(
    hf_src_tokenizer,
    hf_tgt_tokenizer,
    embedding_size=64,
    hidden_size=64,
    layers=3,
).to(device)

wide_model.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEAF
wide_model.load_state_dict(wide_model.best_model)
```

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

In [ ]:
```
# Evaluate model performance
print (f'Test perplexity: {wide_model.evaluate_ppl(test_iter):.3f}')
```

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

Trying a narrower hidden size

A potential problem with this encoder-decoder model is that all communication between the encoder and the decoder falls within the final fixed size h_t vector. To emphasize the problem and allow comparisons for the next lab, we construct another model, `narrow_model`, that uses a very small hidden_size (16). (That's why we called the previous model, with hidden size 64, `wide_model`.) Notice the effect of the reduced dimension on performance.

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

# Instantiate and train classifier
narrow_model = EncoderDecoder(
    hf_src_tokenizer,
    hf_tgt_tokenizer,
    embedding_size=16,
    hidden_size=16,
    layers=3,
).to(device)

narrow_model.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LE
narrow_model.load_state_dict(narrow_model.best_model)
```

```
In [ ]: # Evaluate model performance
        print (f'Test perplexity: {narrow_model.evaluate_ppl(test_iter):.3f}')
```

We'll return to this issue in the next lab, when we address it using *cross-attention*.

Beam search decoding

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 $x_1, \ldots, x_S$, we want to find the target sequence $y_1^*, \ldots, y_T^*, y_{T+1}^*$ (recall that $y_{T+1} = \langle\text{eos}\rangle$) such that the conditional likelihood is maximized:

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

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

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 $H_t = \{(y_1^{(k)}, \ldots, y_t^{(k)}) : k \in \{1, \ldots, K\}\}$ at every step t. To proceed to $t+1$, we compute the scores of sequences $y_1^{(k)}, \ldots, y_t^{(k)}, y_{t+1}$ for every possible extension $y_{t+1} \in \mathcal{V}$ and every possible prefix $(y_1^{(k)}, \ldots, y_t^{(k)}) \in H_t$, 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 $H_{t+1} = \{(y_1^{(k)}, \ldots, y_{t+1}^{(k)}) : k \in \{1, \ldots, K\}\}$. 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,

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

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
 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 hypothesis
```

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.

In [ ]:

```python
MAX_T = 15  # max target length


class Beam:
    """Helper class for storing a hypothesis, its score and its decoder hidd

    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 (1, max_src_len)
            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 = []
            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_increme
                ...
                decoder_state = ...
                total_scores = ...
                all_total_scores.append(total_scores)
                beam.decoder_state = decoder_state  # update decoder state i
            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
            all_scores_flattened = all_total_scores.view(-1)  # K*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 fro
```

```
            ...

            # 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 [token.item() for token in finished[0].tokens]
        else:  # when nothing is finished, return an unfinished hypothesis
            return [token.item() for token in beams[0].tokens]
```

Here's a function to test a model using beam search on a test set. It returns the accuracy of the model on the test set, printing the first few test items if requested.

In [ ]:
```
def test_beam_search(model, test_iter, K=5, print_first=0):
    """Runs beam search with a beam size of `K` on the batches in
    `test_iter` using the provided `model`. Returns the accuracy
    of the model on the test set.

    Arguments:
        model: the `EncoderDecoder` model to test
        test_iter: iterator generating batches of test items
        K: beam width
        print_first: number of test items to print information about,
                     including the source, the predicted output and
                     the ground truth output. Marks errors with "***"
    Returns:
        the accuracy on the test set (correct /total test items)
    """
    correct = 0
    total = 0

    # create beam searcher
    beam_searcher = BeamSearcher(model)

    for index, batch in enumerate(test_iter, start=1):
        # Input and output
        src = batch["src_ids"]
        src_lengths = batch["src_lengths"]
        # Predict
        prediction = beam_searcher.beam_search(src, src_lengths, K)
        # Convert to string
        prediction = hf_tgt_tokenizer.decode(prediction, skip_special_tokens
        ground_truth = hf_tgt_tokenizer.decode(
            batch["tgt_ids"][0], skip_special_tokens=True
        )
        # Print out the first few examples
        if print_first >= index:
            src = hf_src_tokenizer.decode(src[0], skip_special_tokens=True)
            print(
                f"Source:       {index}. {src}\n"
                f"Prediction:   {prediction} {' *** INCORRECT' if predictior
                f"Ground truth: {ground_truth}\n"
```

```
            )

        if ground_truth == prediction:
            correct += 1
        total += 1
    return correct / total
```

In [ ]:
```
grader.check("beam_search_testing")
```

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

In [ ]:
```
print(f'Accuracy of model: {test_beam_search(wide_model, test_iter, K=5, pri
```

To demonstrate the effect of beam size and hidden size, we try some tests with the two models at different beam sizes.

In [ ]:
```
print(f"{'beam':>10}{'wide':>10}{'narrow':>10}")
for K in [1, 5]:
    print(f"{K:>10}", end="")
    for the_model in [wide_model, narrow_model]:
        performance = test_beam_search(the_model, test_iter, K=K, print_firs
        print(f"{performance:>10.5f}", end="")
    print("")
```

Feel free to try other models and other values of K.

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.

** Optional 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.

Lab debrief

Question: We're interested in any thoughts you have 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, but you're not restricted to these:

- 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?

Type your answer here, replacing this text.

Question: What *specific single* change to the lab would have made your learning more efficient? This might be an addition of a concept that was not explained, or an example that would clarify a concept, or a problem that would have captured a concept in a better way, or anything else you can think of that would have made this a better lab.

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.

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