

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/cs236299-2024-winter/lab4-3.git .tmp
    mv .tmp/tests ./
    mv .tmp/requirements.txt ./
    rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

Monter Google Drive pour accéder à vos fichiers

```
from google.colab import drive
drive.mount('/content/drive')
```

Copier les fichiers depuis Google Drive vers l'environnement Colab

```
!cp -r /content/drive/MyDrive/nlp_lab/lab4-3* .
```

Installer les dépendances depuis le fichier requirements.txt

```
!pip install -r /content/drive/MyDrive/nlp_lab/lab4-3/requirements.txt
```

Installer Otter Grader (si nécessaire)

```
!pip install otter-grader
```

Fonction pour exécuter les commandes shell

```
import os
```

```
def shell(commands, warn=True):
    """Exécute des commandes shell et affiche les résultats."""
    file = os.popen(commands)
    print(file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status is not None:
        print(f"Command failed with exit code {exit_status}")
    return exit_status
```

Vérifier si requirements.txt existe et télécharger le dépôt si nécessaire

```
shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
    rm -rf .tmp
    git clone https://github.com/cs236299-2024-winter/lab1-4.git .tmp
    mv .tmp/tests ./tests
    mv .tmp/requirements.txt ./requirements.txt
    rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

Copier les fichiers tests depuis Google Drive si le dossier 'tests' est introuvable

```

if not os.path.exists('tests/token_count.py'):
    print("Téléchargement des fichiers nécessaires depuis Google Drive...")
    !mkdir -p tests
    !cp -r /content/drive/MyDrive/nlp_lab/lab4-3/tests/* ./tests/
else:
    print("Les fichiers de tests sont déjà présents.")

```

```

Uninstalling fsspec-2024.10.0:
  Successfully uninstalled fsspec-2024.10.0
Attempting uninstall: nvidia-cusparse-cu12
  Found existing installation: nvidia-cusparse-cu12 12.5.1.3
  Uninstalling nvidia-cusparse-cu12-12.5.1.3:
    Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
Attempting uninstall: nvidia-cudnn-cu12
  Found existing installation: nvidia-cudnn-cu12 9.3.0.75
  Uninstalling nvidia-cudnn-cu12-9.3.0.75:
    Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
Attempting uninstall: nvidia-cusolver-cu12
  Found existing installation: nvidia-cusolver-cu12 11.6.3.83
  Uninstalling nvidia-cusolver-cu12-11.6.3.83:
    Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This
gcsfs 2024.10.0 requires fsspec==2024.10.0, but you have fsspec 2024.9.0 which is incompatible.
Successfully installed PyPDF2-3.0.1 datasets-3.2.0 dill-0.3.8 docker-7.1.0 fsspec-2024.9.0 jedi-0.19.2 multiproc
Requirement already satisfied: otter-grader in /usr/local/lib/python3.11/dist-packages (1.0.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.11/dist-packages (from otter-grader) (6.0.2)
Requirement already satisfied: nbformat in /usr/local/lib/python3.11/dist-packages (from otter-grader) (5.10.4)
Requirement already satisfied: ipython in /usr/local/lib/python3.11/dist-packages (from otter-grader) (7.34.0)
Requirement already satisfied: nbconvert in /usr/local/lib/python3.11/dist-packages (from otter-grader) (7.16.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from otter-grader) (4.67.1)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from otter-grader) (75.1.0)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from otter-grader) (2.2.2)
Requirement already satisfied: tornado in /usr/local/lib/python3.11/dist-packages (from otter-grader) (6.4.2)
Requirement already satisfied: docker in /usr/local/lib/python3.11/dist-packages (from otter-grader) (7.1.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from otter-grader) (3.1.5)
Requirement already satisfied: dill in /usr/local/lib/python3.11/dist-packages (from otter-grader) (0.3.8)
Requirement already satisfied: pdfkit in /usr/local/lib/python3.11/dist-packages (from otter-grader) (1.0.0)
Requirement already satisfied: PyPDF2 in /usr/local/lib/python3.11/dist-packages (from otter-grader) (3.0.1)
Requirement already satisfied: requests>=2.26.0 in /usr/local/lib/python3.11/dist-packages (from docker->otter-gr
Requirement already satisfied: urllib3>=1.26.0 in /usr/local/lib/python3.11/dist-packages (from docker->otter-gr
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader
Requirement already satisfied: decorator in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grade
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-gr
Requirement already satisfied: prompt-toolkit!=3.0.0,!>=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: pygments in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader)
Requirement already satisfied: backcall in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.11/dist-packages (from ipython->otter
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grade
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->otter-gr
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-
Requirement already satisfied: bleach!=5.0.0 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-gra
Requirement already satisfied: defusedxml in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-gra
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.11/dist-packages (from nbconvert->ott
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.11/dist-packages (from nbconvert->o
Requirement already satisfied: mistune<4,>=2.0.3 in /usr/local/lib/python3.11/dist-packages (from nbconvert->ott
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grade
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.11/dist-packages (from nbconvert->
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/python3.11/dist-packages (from nbformat->o
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.11/dist-packages (from nbformat->otter-
Requirement already satisfied: numpy>=1.23.2 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-gra
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas->o
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-grade
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-gra

```

```

# Initialize Otter
import otter
grader = otter.Notebook()

```

```

%%latex \newcommand{\vect}[1]{\mathbf{#1}} \newcommand{\cnt}[1]{\sharp(#1)} \newcommand{\argmax}[1]{\underset{#1}
{\operatorname{argmax}}} \newcommand{\softmax}{\operatorname{softmax}} \newcommand{\Prob}{\Pr}
\newcommand{\given}{\,|\,}

```

✓ Course 236299

Lab 4-3 - Sequence-to-sequence models

In lab 4-1, you used a syntactic-semantic grammar for semantic parsing to convert natural language to meaning representations. 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 {-}

```
import copy
import csv
import math
import os
import wget

import torch
import torch.nn as nn

from datasets import load_dataset

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 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)
```

 cuda

```
# Download data
def download_if_needed(source, dest, filename):
    os.makedirs(dest, exist_ok=True) # ensure destination
    os.path.exists(f"{dest}/{filename}") or wget.download(source + filename, out=dest)

local_dir = "data/"
remote_dir = "https://github.com/nlp-236299/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",
]:
    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.

```
# 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_in:
        with open(out_file, 'w', newline='') 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()))
```

Let's take a look at what each data file looks like.

```
shell('head "data/dev.csv"')
```


 src,tgt

```
seven thousand nine hundred and twenty nine,7 9 2 9
eight hundred and forty two thousand two hundred and fifty nine,8 4 2 2 5 9
five hundred and eight thousand two hundred and seventeen,5 0 8 2 1 7
one hundred and thirty three million three hundred and sixty two thousand five hundred and twenty,1 3 3 3 6 2 5 2
seventy nine million five hundred and sixty four thousand six hundred and thirty six,7 9 5 6 4 6 3 6
four million seven hundred and forty three thousand nine hundred and twenty nine,4 7 4 3 9 2 9
nine hundred and eighty nine million five hundred and ninety five thousand five hundred and thirteen,9 8 9 5 9 5 5
three hundred and forty four thousand three hundred and seventy three,3 4 4 3 7 3
one thousand six hundred and sixty seven,1 6 6 7
```

✓ 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.

```
with open(local_dir + "dev.csv") as f:
    for line, _ in zip(f, range(4)):
        src, tgt = line.strip().split(',')
        print (f'{src.strip():70s} {tgt.strip():>12s}')
```



src	tgt
seven thousand nine hundred and twenty nine	7 9 2 9
eight hundred and forty two thousand two hundred and fifty nine	8 4 2 2 5 9
five hundred and eight thousand two hundred and seventeen	5 0 8 2 1 7

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).

```
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

Generating train split: 65022/0 [00:00<00:00, 280414.60 examples/s]

Generating val split: 700/0 [00:00<00:00, 20598.83 examples/s]

Generating test split: 700/0 [00:00<00:00, 17846.58 examples/s]

```
DatasetDict({
  train: Dataset({
    features: ['src', 'tgt'],
    num_rows: 65022
  })
  val: Dataset({
    features: ['src', 'tgt'],
    num_rows: 700
  })
  test: Dataset({
    features: ['src', 'tgt'],
    num_rows: 700
  })
})
```

```
train_data = dataset['train']
```

```
val_data = dataset['val']
```

```
test_data = dataset['test']
```

```
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_token, eos_token])
```

```
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.

```
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)

```

```

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)

```

```

Map: 100% 65022/65022 [00:15<00:00, 4441.50 examples/s]
Map: 100% 700/700 [00:00<00:00, 4058.16 examples/s]
Map: 100% 700/700 [00:00<00:00, 3871.36 examples/s]

```

```

# 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]}")

```

```

Size of src vocab: 34
Size of tgt vocab: 14
Index for src padding: 0
Index for tgt padding: 0
Index for start of sequence token: 2
Index for end of sequence token: 3

```

+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.

```

BATCH_SIZE = 32 # batch size for training and validation
TEST_BATCH_SIZE = 1 # batch size for test; we use 1 to make implementation easier

```

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(device)
    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_token]).to(device)
    tgt_batch = torch.zeros(bsz, tgt_max_length).long().fill_(tgt_vocab[pad_token]).to(device)
    for b in range(bsz):
        src_batch[b][:len(src_ids[b])] = torch.LongTensor(src_ids[b]).to(device)
        tgt_batch[b][:len(tgt_ids[b])] = torch.LongTensor(tgt_ids[b]).to(device)

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

```

```

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.

```

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)}")

```

```

➤ Size of src batch: torch.Size([32, 19])
  Third src sentence in batch: tensor([ 9,  3,  2, 16, 10,  4, 12,  3,  2, 22, 12,  0,  0,  0,  0,  0,  0,  0,
    0], device='cuda:0')
  Length of the third src sentence in batch: 19
  Converted back to string: three hundred and ninety nine thousand one hundred and fifty one [PAD] [PAD] [PAD] [PAD]
  Size of tgt batch: torch.Size([32, 12])
  Third tgt sentence in batch: tensor([ 2,  4,  5,  5, 12,  9, 12,  3,  0,  0,  0,  0], device='cuda:0')
  Converted back to string: <bos> 3 9 9 1 5 1 <eos> [PAD] [PAD] [PAD] [PAD]

```

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

In practice, we prepend a special beginning-of-sequence symbol $y_0 = \text{<bos>}$ 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 $y_{T+1} = \text{<eos>}$ 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,

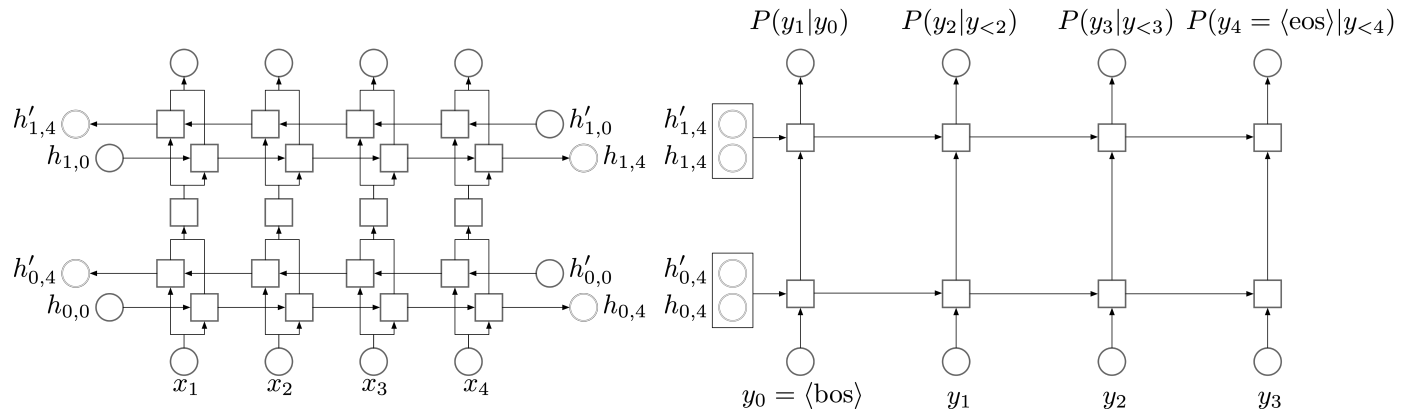
$$\Pr_{\theta}(y_1, \dots, y_T \mid x_1, \dots, x_S) = \prod_{t=1}^{T+1} \Pr_{\theta}(y_t \mid y_{<t}, x_1, \dots, x_S)$$

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 in the course website, called `reshaping.ipynb`, that discusses the reshaping issue in some detail. You'll want to look it over. Another, summarized and more readable option, is to use [einops.rearrange](#).

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)`.

```
# TODO - finish implementing the `forward_encoder` and `forward_decoder` methods
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)

        # 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://discuss.pytorch.org/t/simple-working-example-how-to-use-
    # Note that the batch size is the first dimension, and the sequences are not sorted.
    src_embeddings = self.word_embeddings_src(src) # (batch_size, max_src_len, embedding_size)
    packed_src = pack(src_embeddings, src_lengths.cpu(), batch_first=True, enforce_sorted=False)
    _, (h, c) = self.encoder_rnn(packed_src)

    h = torch.cat([h[0:h.size(0):2], h[1:h.size(0):2]], dim=-1) # (layers, batch_size, hidden_size)
    c = torch.cat([c[0:c.size(0):2], c[1:c.size(0):2]], dim=-1) # (layers, batch_size, hidden_size)
    return (h, c)

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 operation)
    """
    # Get embeddings for target words
    tgt_embeddings = self.word_embeddings_tgt(tgt_in)

    # Pass through decoder RNN
    decoder_outs, _ = self.decoder_rnn(tgt_embeddings, encoder_final_state)

    # Project decoder outputs to logits using hidden2output layer
    logits = self.hidden2output(decoder_outs) # Use decoder_outs instead of h

    return logits # Return the logits

...

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) # tuple (h, c)
    # 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 size 1!"
    # 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, decoder_state)
    # 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):
    """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 = batch['src_ids'] # bsz, max_src_len
        src_lengths = batch['src_lengths'] # bsz
        tgt_in = batch['tgt_ids'][:, :-1].contiguous() # remove <eos> for decoder input (y_0=<bos>, y_1, y_2)
        tgt_out = batch['tgt_ids'][:, 1:].contiguous() # remove <bos> as decoder output (y_1, y_2, y_3=<eos>)
        # Forward to get logits
        logits = self.forward(src, src_lengths, tgt_in) # bsz, tgt_len, V_tgt
        # 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
            tgt = batch['tgt_ids'] # bsz, max_tgt_len
            src = batch['src_ids'] # bsz, max_src_len
            src_lengths = batch['src_lengths'] # bsz
            tgt_in = tgt[:, :-1] # Remove <eos> for decoder input (y_0=<bos>, y_1, y_2)
            tgt_out = tgt[:, 1:] # Remove <bos> as decoder output (y_1, y_2, y_3=<eos>)
            batch_size = src.size(0)
            # Run forward pass and compute loss along the way.
            logits = self.forward(src, src_lengths, tgt_in) # bsz, tgt_len, V_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().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(
    hf_src_tokenizer,
    hf_tgt_tokenizer,
    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)

100%|██████████| 2032/2032 [00:37<00:00, 54.08it/s]
Epoch: 0 Training Perplexity: 1.6708 Validation Perplexity: 1.0580
100%|██████████| 2032/2032 [00:36<00:00, 55.97it/s]
Epoch: 1 Training Perplexity: 1.0382 Validation Perplexity: 1.0409
<All keys matched successfully>

```

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}')

```

```

Test perplexity: 1.044

```

```

grader.check("encoder_decoder_ppl")

```

```

All tests passed!

```

✓ 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, \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:

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

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

Greedy decoding does not guarantee finding the best sequence because it focuses only on local probabilities, selecting the most likely word at each step without considering the global context of the sequence. This can lead to suboptimal results, as it may miss better sequences that require temporarily choosing less likely words to achieve a higher overall probability. Since it does not explore beyond the immediate choices, greedy decoding can get stuck in local optima. Methods like beam search address this limitation by exploring multiple paths and balancing local and global probabilities, increasing the chances of finding the optimal sequence.

✓ 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 $t = 1$, $H_1 = \{(y) : y \in \text{K-argmax}_{y_1 \in \mathcal{V}} \log P(y_1 | y_0 = \text{bos})\}$. Here K is called the beam size.

To summarize,

$$H_1 = \{(y) : y \in \text{K-argmax}_{y_1 \in \mathcal{V}} \log P(y_1 | y_0 = \text{bos})\}$$

$$H_{t+1} = \text{K-argmax}_{\{(y_1, y_2, \dots, y_{t+1}) \in \mathcal{V}^{t+1} : (y_1, \dots, y_t) \in H_t\}} \log P(y_1, \dots, y_{t+1} | x)$$

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 $|\mathcal{V}|$).

When $K=1$, beam search is the same as greedy search because it only keeps one hypothesis, making the same local decisions at each step. It does not improve over greedy search.

When $K>1$, beam search is better because it explores multiple possible sequences, reducing the risk of getting stuck in a local maximum. It increases the chance of finding a higher-probability sequence by considering more options at each step.

To guarantee finding the globally best sequence, we need $K = |\mathcal{V}|^T$, meaning we must explore all possible sequences of length T in the vocabulary \mathcal{V} . However, this is computationally impossible for large vocabularies, so in practice, we use a reasonable K to balance quality and efficiency.

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

```
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 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.

Le code généré est peut-être soumis à une licence |

`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 (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 = self.model.forward_encoder(src, src_lengths)
        init_beam = Beam(decoder_state=encoder_final_state, tokens=[self.bos_id], score=0)
        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_incrementally`
                logits, decoder_state = self.model.forward_decoder_incrementally(decoder_state, torch.tensor([y_t], device=
                total_scores = logits + score

```



```

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=True)
ground_truth = hf_tgt_tokenizer.decode(batch[ 'tgt_ids' ][0],
                                     skip_special_tokens=True)

# Print out the first few examples
if DEBUG_FIRST >= index :
    src = hf_src_tokenizer.decode(src[0], skip_special_tokens=True)
    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}')

```

```

Source:      1. sixteen thousand eight hundred and thirty two
Prediction:   1 6 8 3 2
Ground truth: 1 6 8 3 2

Source:      2. sixty seven million six hundred and eighty five thousand two hundred and thirty
Prediction:   6 7 6 8 5 2 3 0
Ground truth: 6 7 6 8 5 2 3 0

Source:      3. six thousand two hundred and twelve
Prediction:   6 2 1 2
Ground truth: 6 2 1 2

Source:      4. seven hundred and ninety eight million three hundred and thirty one thousand eight hundred and ei
Prediction:   7 9 8 3 3 1 8 1 8
Ground truth: 7 9 8 3 3 1 8 1 8

Source:      5. eighty eight million four hundred and thirteen thousand nine hundred and eighteen
Prediction:   8 8 4 1 3 9 1 8
Ground truth: 8 8 4 1 3 9 1 8

Source:      6. three hundred and seventy four thousand two hundred and seventy
Prediction:   3 7 4 2 7 0
Ground truth: 3 7 4 2 7 0

Source:      7. ninety eight million three hundred and seventy thousand five hundred and forty five
Prediction:   9 8 3 7 0 5 4 5
Ground truth: 9 8 3 7 0 5 4 5

Source:      8. ninety seven thousand seven hundred and sixty two
Prediction:   9 7 7 6 2
Ground truth: 9 7 7 6 2

Source:      9. four hundred and ten thousand two hundred and three
Prediction:   4 1 0 2 0 3
Ground truth: 4 1 0 2 0 3

Source:     10. five hundred and ninety eight thousand three hundred and ninety seven
Prediction:   5 9 8 3 9 7
Ground truth: 5 9 8 3 9 7

Accuracy: 0.92

```

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.

✓ 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:

- 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.
- 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?

The lab was well-structured and provided a solid understanding of sequence-to-sequence models and beam search decoding. One improvement could be adding a step-by-step walkthrough of beam search with a small example, as implementing it required careful attention to tensor operations. The length felt appropriate, balancing theoretical understanding and practical implementation. The readings were relevant and helped reinforce the concepts needed for the exercises. After completing the lab, the objectives became clear, particularly how beam search improves over greedy decoding. Overall, this was a valuable learning experience, and a visual explanation of the beam search process could further enhance clarity.

✓ End of Lab 4-3 {-}

Double-cliquez (ou appuyez sur Entrée) pour modifier

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

```
grader.check_all()
```

```
↻ beam_search:  
All tests passed!  
encoder_decoder_ppl:  
All tests passed!
```