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

```
Completed with errors. Exit status: 256

256
```

```
# 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/lab4-4* .

# Installer les dépendances depuis le fichier requirements.txt
!pip install -r /content/drive/MyDrive/nlp_lab/lab4-4/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/lab4-4/tests/* ./tests/
else:
    print("Les fichiers de tests sont déjà présents.")
```

```
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.11/dist-packages (from jsonschema>=2.6->nbformat->otter-g
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.11/dist-packages (from jupyter-core>=4.7->nbconvert->o
Requirement already satisfied: jupyter-client>=6.1.12 in /usr/local/lib/python3.11/dist-packages (from nbclient>=0.5.0->nbconvert
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.11/dist-packages (from pexpect>4.3->ipython->otter-grade
Requirement already satisfied: wcwidth in /usr/local/lib/python3.11/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-packages (from beautifulsoup4->nbconvert->otter-gr
Requirement already satisfied: pyzmq>=13 in /usr/local/lib/python3.11/dist-packages (from jupyter-client>=6.1.12->nbclient>=0.5.0
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-grader) (2.32.3)
Requirement already satisfied: urllib3>=1.26.0 in /usr/local/lib/python3.11/dist-packages (from docker->otter-grader) (2.3.0)
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (0.19.2)
Requirement already satisfied: decorator in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (4.4.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (0.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from ipyt
Requirement already satisfied: pygments in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (2.18.0)
Requirement already satisfied: backcall in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (0.2.0)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (0.1.7)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.11/dist-packages (from ipython->otter-grader) (4.9.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->otter-grader) (3.0.2)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (4.12.3)
Requirement already satisfied: bleach!=5.0.0 in /usr/local/lib/python3.11/dist-packages (from bleach[css]!=5.0.0->nbconvert->otte
Requirement already satisfied: defusedxml in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (0.7.1)
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (5.7.2
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (0.3
Requirement already satisfied: mistune<4,>=2.0.3 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (3.1.1
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (0.10.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (24.2)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.11/dist-packages (from nbconvert->otter-grader) (1.
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/python3.11/dist-packages (from nbformat->otter-grader) (2.2
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.11/dist-packages (from nbformat->otter-grader) (4.23.0)
Requirement already satisfied: numpy>=1.23.2 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-grader) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-grader) (2.8
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-grader) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->otter-grader) (2025.1)
Requirement already satisfied: webencodings in /usr/local/lib/python3.11/dist-packages (from bleach!=5.0.0->bleach[css]!=5.0.0->n
Requirement already satisfied: tinycss2<1.5,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from bleach[css]!=5.0.0->nbconver
Requirement already satisfied: parso<0.9.0,>=0.8.4 in /usr/local/lib/python3.11/dist-packages (from jedi>=0.16->ipython->otter-gr
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.11/dist-packages (from jsonschema>=2.6->nbformat->otter-gr
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.11/dist-packages (from jsonschema>=
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.11/dist-packages (from jsonschema>=2.6->nbformat->ot
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.11/dist-packages (from jsonschema>=2.6->nbformat->otter-g
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.11/dist-packages (from jupyter-core>=4.7->nbconvert->o
Requirement already satisfied: jupyter-client>=6.1.12 in /usr/local/lib/python3.11/dist-packages (from nbclient>=0.5.0->nbconvert
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.11/dist-packages (from pexpect>4.3->ipython->otter-grade
```

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

Type de cellule non compatible. Double-cliquez pour examiner/modifier le contenu.

Course 236299

Lab 4-4 - Sequence-to-sequence models with attention

In lab 4-3, you built a sequence-to-sequence model in its most basic form and applied it to the task of words-to-numbers conversion. That model first encodes the source sequence into a fixed-size vector (encoder final states), and then decodes based on that vector. Since the only way information from the source side can flow to the target side is through this fixed-size vector, it presents a bottleneck in the encoder-decoder model: no matter how long the source sentence is, it must always be compressed into this fixed-size vector.

An *attention mechanism* (proposed in this seminal paper), and discussed in segment 2, offers a workaround by providing the decoder a dynamic view of the source-side as the decoding proceeds. Instead of compressing the source sequence into a *fixed-size* vector, we preserve

the "resolution" and encode the source sequence into a *set of vectors* (usually with the same size as the source sequence) which is sometimes called a *memory bank*. When predicting each word, the decoder "attends to" this memory bank and assigns a weight to each vector in the set, and the weighted sum of those vectors will be used to make a prediction. Hopefully, the decoder will assign higher weights to more relevant source words when predicting a target word, which we'll test in this lab.

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

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

Preparation - Loading data {-}

We use the same data as in lab 4-3.

```
import copy
import csv
import math
import matplotlib
import matplotlib.pyplot as plt
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
from torch.nn.utils.rnn import pad_packed_sequence as unpack


# Specify matplotlib configuration
%matplotlib inline
plt.style.use('tableau-colorblind10')

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

```
cpu
```

```
# 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)
```

As in lab 4-3, we process the dataset by extracting the sequences and their corresponding labels and save it in CSV format. Then, we load the data from the CSV files, train the tokenizers, prepend `<bos>` and appended `<eos>` to target sentences, and convert the data to sequences of token ids.

```
# 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()))

dataset = load_dataset('csv', data_files={'train':f'{local_dir}train.csv', \
                                          'val': f'{local_dir}dev.csv', \
                                          'test': f'{local_dir}test.csv'})

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

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_toker

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)

# Compute size of vocabulary
src_vocab = src_tokenizer.get_vocab()
tgt_vocab = 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]}")
```

```
Generating train split:  65022/0 [00:00<00:00, 168614.90 examples/s]
Generating val split:  700/0 [00:00<00:00, 7938.77 examples/s]
Generating test split:  700/0 [00:00<00:00, 7971.88 examples/s]
Map: 100%  65022/65022 [00:31<00:00, 2822.67 examples/s]
Map: 100%  700/700 [00:00<00:00, 3240.48 examples/s]
Map: 100%  700/700 [00:00<00:00, 2773.88 examples/s]
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([12,  5,  6,  3,  2, 30,  4, 12,  3,  2, 15, 12,  0,  0,  0,  0,  0,  0,
         0])
```

```
Length of the third src sentence in batch: 19
Converted back to string: one million two hundred and ten thousand one hundred and thirty one [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
Size of tgt batch: torch.Size([32, 12])
Third tgt sentence in batch: tensor([ 2, 12,  7, 12, 13, 12,  4, 12,  3,  0,  0,  0])
Converted back to string: <bos> 1 2 1 0 1 3 1 <eos> [PAD] [PAD] [PAD]
```

The attention mechanism

> **Note:** This is a rehash of the attention mechanism, as presented in segment 2. Use it for reference and feel free to skim it if you remember the mechanism well.

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

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

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

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

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

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

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

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

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

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

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

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

If there is an index (i) where the attention weight (a_i) is exactly 1, the mechanism fully focuses on that specific key-value pair while completely ignoring all others. As a result, the context vector becomes identical to the corresponding value (v_i), without any contribution from the rest of the sequence.

In this case, the attention mechanism acts as a hard selection filter, isolating the most relevant part of the input without blending different pieces of information. While this is useful for one-to-one mappings (such as converting "three" to "3"), it may not be ideal for tasks requiring a broader contextual understanding, where multiple tokens collectively influence the meaning.

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

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

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

the shape of a is T × S where

- T is the number of queries i e the number of decoding time steps
- S is the number of keys i e the number of encoding time steps

each element Aij represents the attention weight assigned to the j th key encoder output when computing the i th query decoder state

in other words Aij indicates how much influence the j th source token has on generating the i th target token the sum of each row in a is equal to 1 as softmax ensures that attention weights are normalized across all source tokens

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

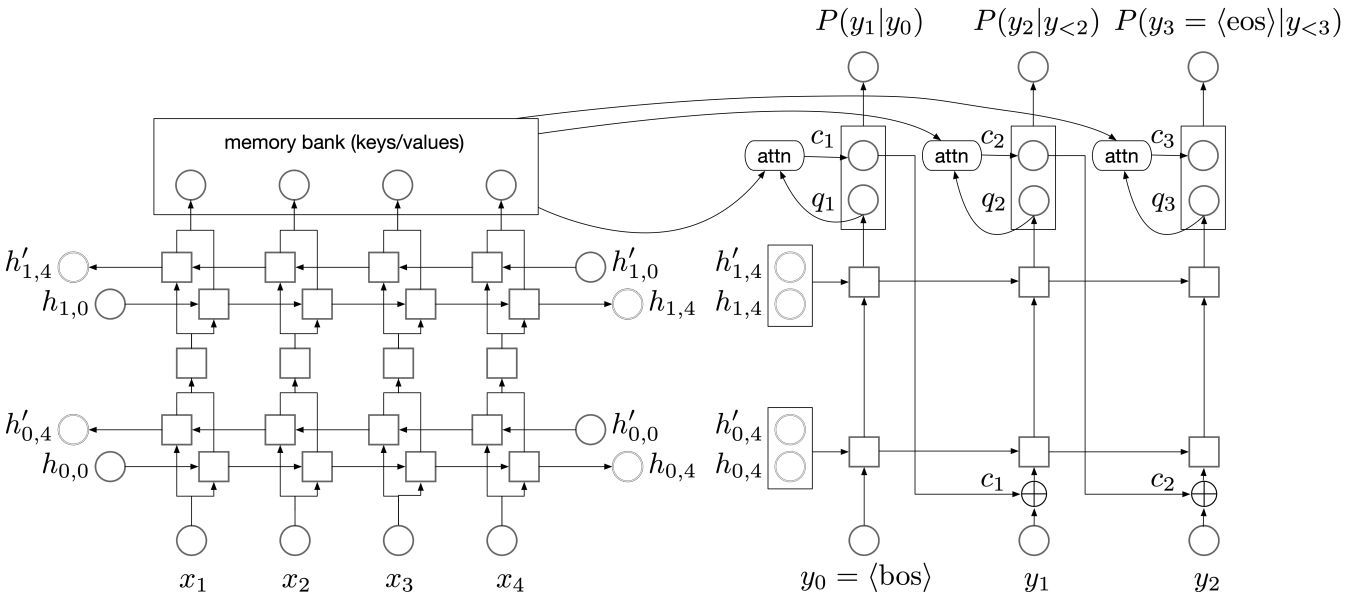
$$C = AV$$

You are given the implementation to the attention function from lab 2-4, which takes the Q, K, and V matrices and returns the A and C matrices. Note that for these matrices, there is one additional dimension for the batching, so instead of $Q \in \mathbb{R}^{T \times D}$, $K, V \in \mathbb{R}^{S \times D}$, $A \in \mathbb{R}^{T \times S}$, $C \in \mathbb{R}^{T \times D}$, we have $Q \in \mathbb{R}^{B \times T \times D}$, $K, V \in \mathbb{R}^{B \times S \times D}$, $A \in \mathbb{R}^{B \times T \times S}$, $C \in \mathbb{R}^{B \times T \times D}$, where B is the batch size. In addition, the function below also takes an argument `mask` of size $\mathbb{R}^{B \times T \times S}$ to mark where attentions are disallowed. This is useful not only in disallowing attending to padding symbols, but also in implementing the transformer model.

```
def attention(batched_Q, batched_K, batched_V, mask=None):
  """
  Performs the attention operation and returns the attention matrix
  `batched_A` and the context matrix `batched_C` using queries
  `batched_Q`, keys `batched_K`, and values `batched_V`.

  Arguments:
      batched_Q: (bsz, q_len, D)
      batched_K: (bsz, k_len, D)
      batched_V: (bsz, k_len, D)
      mask: (bsz, q_len, k_len). An optional boolean mask *disallowing*
            attentions where the mask value is *`False`*.
  Returns:
      batched_A: the normalized attention scores (bsz, q_len, k_len)
      batched_C: a tensor of size (bsz, q_len, D).
  """
 # Check sizes
  D = batched_Q.size(-1)
  bsz = batched_Q.size(0)
  q_len = batched_Q.size(1)
  k_len = batched_K.size(1)

  assert batched_K.size(-1) == D and batched_V.size(-1) == D
  assert batched_K.size(0) == bsz and batched_V.size(0) == bsz
  assert batched_V.size(1) == k_len
  if mask is not None:
    assert mask.size() == torch.Size([bsz, q_len, k_len])

  q = batched_Q                # bsz, q_len, hidden
  k = batched_K.transpose(1, 2) # bsz, hidden, k_len
  # Compute unnormalized attention scores
  scores = torch.bmm(q, k) / math.sqrt(D) # bsz, q_len, k_len
  # Mask attention scores to -inf where mask is False
  if mask is not None:
    scores = scores.masked_fill(mask == False, -torch.inf)
  batched_A = torch.softmax(scores, dim=-1) # bsz, q_len, k_len
  batched_C = torch.bmm(batched_A, batched_V) # bsz, q_len, D
  # Verify that things sum up to one properly.
  assert torch.all(torch.isclose(batched_A.sum(-1),
                                 torch.ones(bsz, q_len).to(device)))
  return batched_A, batched_C
```

Neural encoder-decoder models with attention

Now we can add an attention mechanism to our encoder-decoder model. As in lab 4-3, we use a bidirectional LSTM as the encoder, and a unidirectional LSTM as the decoder, and initialize the decoder state with the encoder final state. However, instead of directly projecting the decoder hidden state to logits, we use it as a query vector and attend to all encoder outputs (used as both keys and values), and then concatanate the resulting context vector with the query vector, and project to logits. In addition, we add the context vector to the word embedding at the next time step, so that the LSTM can be aware of the previous attention results.



In the above illustration, at the first time step, we use q_1 to denote the decoder output. Instead of directly projecting that to logits as in lab 4-3, we use q_1 as the query vector, and use it to attend to the memory bank (which is the set of encoder outputs) and get the context vector c_1. We concatenate c_1 with q_1, and project the result to the vocabulary size to get logits. At the next step, we first embed y_1 into embeddings, and then **add** c_1 to it (via componentwise addition) and use the sum as the decoder input. This process continues until an end-of-sequence is produced.

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

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

In summary, the steps in decoding are:

1. Map the target words to word embeddings. Add the context vector from the previous time step if any. Use the result as the input to the decoder.
2. Forward the decoder RNN for one time step. Use the decoder output as query, the memory bank as **both keys and values**, and compute the context vector through the attention mechanism. Since we don't want to attend to padding symbols at the source side, we also need

---

Question: Recall that in the `forward_decoder` function in lab 4-3 we didn't use any for loops but instead used a single call to `self.decoder_rnn`. Why do we need a `for` loop in the function `forward_decoder` below? Is it possible to get rid of the for loop to make the code more efficient?

4. Update the decoder hidden state and the context vector, which will be used in the next time step.

in lab 4 3 we did not use a for loop in forward decoder because we could process the entire sequence at once by passing it to self decoder rnn this was possible because during training we already had the full target sequence

however in forward decoder here we use **auto regressive decoding** which means each output depends on the previous one during inference we do not know the full target sequence in advance so we must generate it **step by step** feeding each predicted word back into the model this makes it **impossible to process everything at once** and requires a for loop

to remove the for loop we would need **non autoregressive models** which generate all words at once but these models are harder to train and often less accurate most modern models like transformers still use **autoregressive decoding** but optimize it with techniques like **beam search** to make it faster

---

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

> Hint on using `pack`: if you use `pack` to handle paddings and pass the result as encoder inputs, you need to use `unpack` and extract the first returned value as the memory bank. An example for using `pack` can be found here, but note that our input is already the padded sequences.

> Hint on ignoring source-side paddings in the attention mechanism: what `mask` should we pass into the `attention` function??

```
#TODO - implement `forward_encoder` and `forward_decoder_incrementally`.
class AttnEncoderDecoder(nn.Module):
  def __init__(self, hf_src_tokenizer, hf_tgt_tokenizer, hidden_size=64, layers=3):
```

```
        """
        Initializer. Creates network modules and loss function.
        Arguments:
            hf_src_tokenizer: hf src tokenizer
            hf_tgt_tokenizer: hf tgt tokenizer
            hidden_size: hidden layer size of both encoder and decoder
            layers: number of layers of both encoder and decoder
        """
        super().__init__()
        self.hf_src_tokenizer = hf_src_tokenizer
        self.hf_tgt_tokenizer = hf_tgt_tokenizer

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

        # Get special word ids
        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 = hidden_size
        self.hidden_size = hidden_size
        self.layers = layers

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

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

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

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

    def forward_encoder(self, src, src_lengths):
        """
        Encodes source words `src`.
        Arguments:
            src: src batch of size (bsz, max_src_len)
            src_lengths: src lengths of size (bsz)
        Returns:
            memory_bank: a tensor of size (bsz, src_len, hidden_size)
            (final_state, context): `final_state` is a tuple (h, c) where h/c is of size
                                    (layers, bsz, hidden_size), and `context` is `None`.
        """
        #TODO
        embedded = self.word_embeddings_src(src)
        packed = nn.utils.rnn.pack_padded_sequence(embedded, src_lengths.cpu(), batch_first=True, enforce_sorted=False)
        packed_outputs, (h, c) = self.encoder_rnn(packed)
        memory_bank, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs, batch_first=True)
        h = h.view(self.layers, 2, h.size(1), h.size(2))  # (layers, 2, batch, hidden/2)
        h = torch.cat([h[:, 0], h[:, 1]], dim=-1)  # Merge both directions

        c = c.view(self.layers, 2, c.size(1), c.size(2))
        c = torch.cat([c[:, 0], c[:, 1]], dim=-1)  # Merge both directions


        memory_bank = memory_bank
        final_state = (h, c)
        context = None
        return memory_bank, (final_state, context)
```

```
    def forward_decoder(self, encoder_final_state, tgt_in, memory_bank, src_mask):
      """
      Decodes based on encoder final state, memory bank, src_mask, and ground truth
      target words.
      Arguments:
          encoder_final_state: (final_state, None) where final_state is the encoder
                               final state used to initialize decoder. None is the
                               initial context (there's no previous context at the
                               first step).
          tgt_in: a tensor of size (bsz, tgt_len)
          memory_bank: a tensor of size (bsz, src_len, hidden_size), encoder outputs
                       at every position
          src_mask: a tensor of size (bsz, src_len): a boolean tensor, `False` where
                    src is padding (we disallow decoder to attend to those places).
      Returns:
          Logits of size (bsz, tgt_len, V_tgt) (before the softmax operation)
      """
      max_tgt_length = tgt_in.size(1)

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

      all_logits = []
      for i in range(max_tgt_length):
        logits, decoder_states, attn = \
          self.forward_decoder_incrementally(decoder_states,
                                             tgt_in[:, i],
                                             memory_bank,
                                             src_mask,
                                             normalize=False)
        all_logits.append(logits)             # list of bsz, vocab_tgt
      all_logits = torch.stack(all_logits, 1) # bsz, tgt_len, vocab_tgt
      return all_logits

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

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask,
                                      normalize=True):
      """
      Forward the decoder for a single step with token `tgt_in_onestep`.
      This function will be used both in `forward_decoder` and in beam search.
      Note that bsz can be greater than 1.
      Arguments:
          prev_decoder_states: a tuple (prev_decoder_state, prev_context). `prev_context`
                               is `None` for the first step
          tgt_in_onestep: a tensor of size (bsz), tokens at one step
          memory_bank: a tensor of size (bsz, src_len, hidden_size), encoder outputs
                       at every position
          src_mask: a tensor of size (bsz, src_len): a boolean tensor, `False` where
                    src is padding (we disallow decoder to attend to those places).
          normalize: use log_softmax to normalize or not. Beam search needs to normalize,
                     while `forward_decoder` does not
      Returns:
          logits: log probabilities for `tgt_in_token` of size (bsz, V_tgt)
          decoder_states: (`decoder_state`, `context`) which will be used for the
                          next incremental update
          attn: normalized attention scores at this step (bsz, src_len)
      """
      prev_decoder_state, prev_context = prev_decoder_states
      # Initialize decoder_state and context with previous states if available
      decoder_state = prev_decoder_state
      context = prev_context
      #TODO
      tgt_embedded = self.word_embeddings_tgt(tgt_in_onestep).unsqueeze(1)

      # Update decoder_states after they are calculated
      decoder_states = (decoder_state, context)
```

```
        attn = None #attn_scores.squeeze(1) # Initialize attn
        if prev_context is not None:
            tgt_embedded = tgt_embedded + prev_context

        decoder_output, decoder_state = self.decoder_rnn(tgt_embedded, prev_decoder_state)

        # Calculate and update attention scores and context
        attn_scores, context = attention(decoder_output, memory_bank, memory_bank, mask=src_mask.unsqueeze(1))
        attn = attn_scores.squeeze(1)  # Update attn with the calculated scores

        concat_context = torch.cat([decoder_output.squeeze(1), context.squeeze(1)], dim=-1)

        logits = self.hidden2output(concat_context)

        if normalize:
          logits = torch.log_softmax(logits, dim=-1)


        return logits, decoder_states, attn

      def evaluate_ppl(self, iterator):
        """Returns the model's perplexity on a given dataset `iterator`."""
        # Switch to eval mode
        self.eval()
        total_loss = 0
        total_words = 0
        for batch in iterator:
          # Input and target
          src = batch['src_ids']              # bsz, max_src_len
          src_lengths = batch['src_lengths']  # bsz
          tgt_in = batch['tgt_ids'][:, :-1] # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
          tgt_out = batch['tgt_ids'][:, 1:] # Remove <bos> as target        (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.reshape(-1, self.V_tgt), tgt_out.reshape(-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].contiguous() # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
            tgt_out = tgt[:, 1:].contiguous() # Remove <bos> as target        (y_1, y_2, y_3=<eos>)
            bsz = tgt.size(0)
            # Run forward pass and compute loss along the way.
            logits = self.forward(src, src_lengths, tgt_in)
            loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
            # Training stats
            num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum().item()
            total_words += num_tgt_words
            total_loss += loss.item()
            # Perform backpropagation
            loss.div(bsz).backward()
            optim.step()

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

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

# Instantiate and train classifier
model = AttnEncoderDecoder(hf_src_tokenizer, hf_tgt_tokenizer,
  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 [04:49<00:00,  7.02it/s]
Epoch: 0 Training Perplexity: 3.0782 Validation Perplexity: 2.1350
100%|██████████| 2032/2032 [04:44<00:00,  7.13it/s]
Epoch: 1 Training Perplexity: 1.8141 Validation Perplexity: 1.5896
<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, the expected value should be < 1.05
print (f'Test perplexity: {model.evaluate_ppl(test_iter):.3f}')
```

```
Test perplexity: 1.578
```

```
grader.check("encoder_decoder_ppl")
```

```
All tests passed!
```

Beam search decoding

We can reuse most of our beam search code in lab 4-3 here: we only need to modify the code a bit to pass in `memory_bank` and `src_mask`. For reference here is the same pseudo-code used in lab 4-3, where we want 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 hypothesis
```

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

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

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


  def beam_search(self, src, src_lengths, K, max_T=MAX_T):
    """
    Performs beam search decoding.
    Arguments:
        src: src batch of size (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 and a list of attentions
    """
    finished = []
    all_attns = []

    self.model.eval()

    memory_bank, encoder_final_state = self.model.forward_encoder(src, src_lengths)
    src_mask = src.ne(self.padding_id_src)

    init_decoder_state = encoder_final_state
    init_beam = Beam(decoder_state=init_decoder_state, tokens=[self.bos_id], score=0)
    beams = [init_beam]

    with torch.no_grad():
        for t in range(max_T):
            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]

                logits, decoder_state, attn = self.model.forward_decoder_incrementally(
                    prev_decoder_states=decoder_state,
                    tgt_in_onestep=torch.tensor([y_t], device=src.device),
                    memory_bank=memory_bank,
                    src_mask=src_mask,
                    normalize=True
                )

                total_scores = score + logits.squeeze(0)  # (V,)
                all_total_scores.append(total_scores)
                all_attns.append(attn)

                beam.decoder_state = decoder_state

            all_total_scores = torch.stack(all_total_scores)

            all_scores_flattened = all_total_scores.view(-1)
            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].item()
                y_t_plus_1 = next_tokens[k].item()
                score = topk_scores[k].item()
                decoder_state = beams[beam_id].decoder_state
                y_1_to_t = beams[beam_id].tokens
```

```
                new_beam = Beam(decoder_state=decoder_state, tokens=y_1_to_t + [y_t_plus_1], score=score)

                if y_t_plus_1 == self.eos_id:
                    finished.append(new_beam)
                else:
                    new_beams.append(new_beam)

            beams = new_beams  # Only keep non-finished beams

            if len(beams) == 0:
                break

    # Return the best hypothesis
    if len(finished) > 0:
        finished = sorted(finished, key=lambda beam: -beam.score)
        return finished[0].tokens, all_attns
    elif len(beams) > 0:  # If nothing finished, return the highest scoring unfinished beam
        return beams[0].tokens, all_attns
    else:  # In rare cases where no valid beams remain
        return [self.bos_id], all_attns
```

Double-cliquez (ou appuyez sur Entrée) pour modifier

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

```
grader.check("beam_search")
```

All tests passed!

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

correct = 0
total = 0

# create beam searcher
beam_searcher = BeamSearcher(model)

for index, batch in enumerate(test_iter, start=1):
  # Input and output
  src = 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=True)
  ground_truth = hf_tgt_tokenizer.decode(batch['tgt_ids'][0], skip_special_tokens=True)
  if DEBUG_FIRST > index:
    src = hf_src_tokenizer.decode(src[0], skip_special_tokens=True)
    print (f'Source: {src}')
    print (f'Prediction:   {prediction}')
    print (f'Ground truth: {ground_truth}')
  if ground_truth == prediction:
    correct += 1
  total += 1

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

```
Source: sixteen thousand eight hundred and thirty two
Prediction:   1 6 8 3 2
Ground truth: 1 6 8 3 2
Source: sixty seven million six hundred and eighty five thousand two hundred and thirty
Prediction:   6 8 5 2 3 0
Ground truth: 6 7 6 8 5 2 3 0
Source: six thousand two hundred and twelve
Prediction:   6 2
Ground truth: 6 2 1 2
Source: seven hundred and ninety eight million three hundred and thirty one thousand eight hundred and eighteen
Prediction:   7 9 8
Ground truth: 7 9 8 3 3 1 8 1 8
Source: eighty eight million four hundred and thirteen thousand nine hundred and eighteen
Prediction:   8
Ground truth: 8 8 4 1 3 9 1 8
Source: three hundred and seventy four thousand two hundred and seventy
Prediction:   3 7 4 2 7 4 2 7 4 2 7 4 2 7 4
Ground truth: 3 7 4 2 7 0
Source: ninety eight million three hundred and seventy thousand five hundred and forty five
Prediction:   9 8 3 7 0 5 4 5 4 5 4 5 4 5 4
```

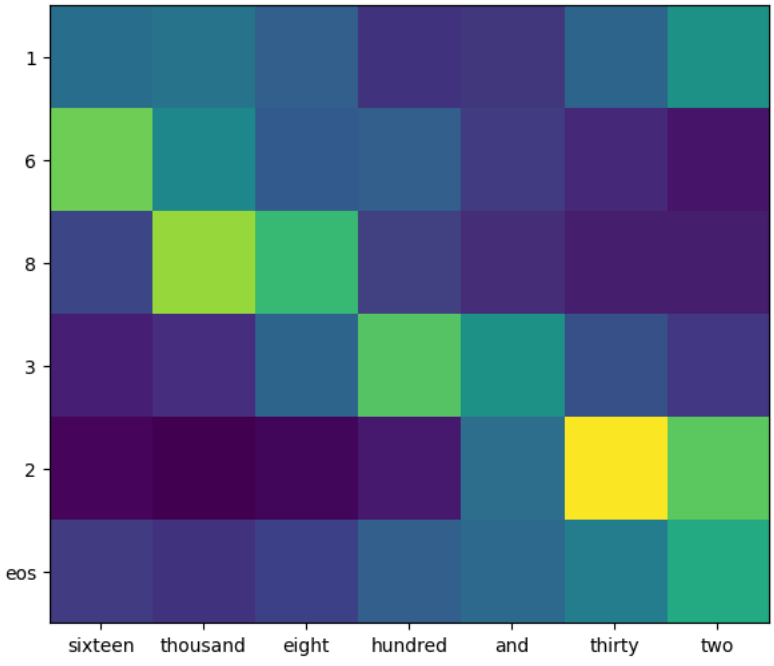
```
Ground truth: 9 8 3 7 0 5 4 5
Source: ninety seven thousand seven hundred and sixty two
Prediction:   9 7 6 2
Ground truth: 9 7 7 6 2
Source: four hundred and ten thousand two hundred and three
Prediction:   4 1 0 3
Ground truth: 4 1 0 2 0 3
Accuracy: 0.16
```

Visualizing attention

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

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

# Create beam searcher
beam_searcher = BeamSearcher(model)
batch = next(iter(test_iter))
# Input and output
src = batch['src_ids']
src_lengths = batch['src_lengths']
# Predict and get attentions
prediction, all_attns = beam_searcher.beam_search(src, src_lengths, K)
all_attns = torch.stack(all_attns, 0)
# Convert to string
prediction = hf_tgt_tokenizer.decode(prediction, skip_special_tokens=True)
ground_truth = hf_tgt_tokenizer.decode(batch['tgt_ids'][0], skip_special_tokens=True)
src =  hf_src_tokenizer.decode(src[0], skip_special_tokens=True)
print (f'Source: {src}')
print (f'Prediction:   {prediction}')
print (f'Ground truth: {ground_truth}')

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

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

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

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



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

The transformer architecture

In lab 2-5 you have used the transformer architecture in a decoder-only model. Here you will use it for both the encoder and decoder.

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

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

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

Yes, we can compute the logits for any two different words (y_{t1}) and (y_{t2}) in parallel at training time when using decoder self-attention.

In an RNN-based decoder, each token must be processed sequentially because the prediction of (y_{t+1}) depends on the hidden state from (y_t). This prevents parallelization, as we need to complete the computation for one step before moving to the next.

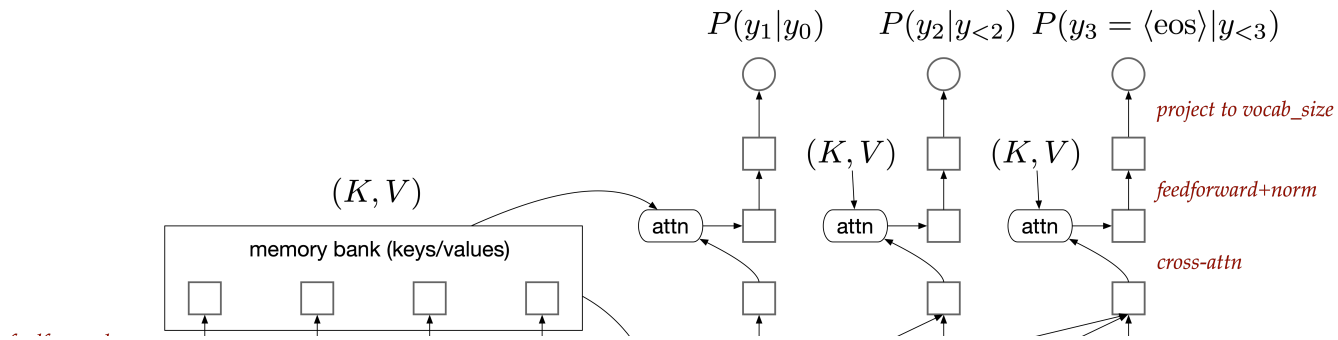
However, in a transformer decoder, self-attention allows the model to process all tokens at once. Since all target tokens are available during training, the model can attend to the entire sequence in a **single computation step**. This means the logits for (y_{t1}) and (y_{t2}) can be computed in parallel, significantly improving efficiency.

In conclusion, by replacing recurrence with self-attention, transformers eliminate the sequential bottleneck of RNNs, making training much faster and more scalable.

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

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

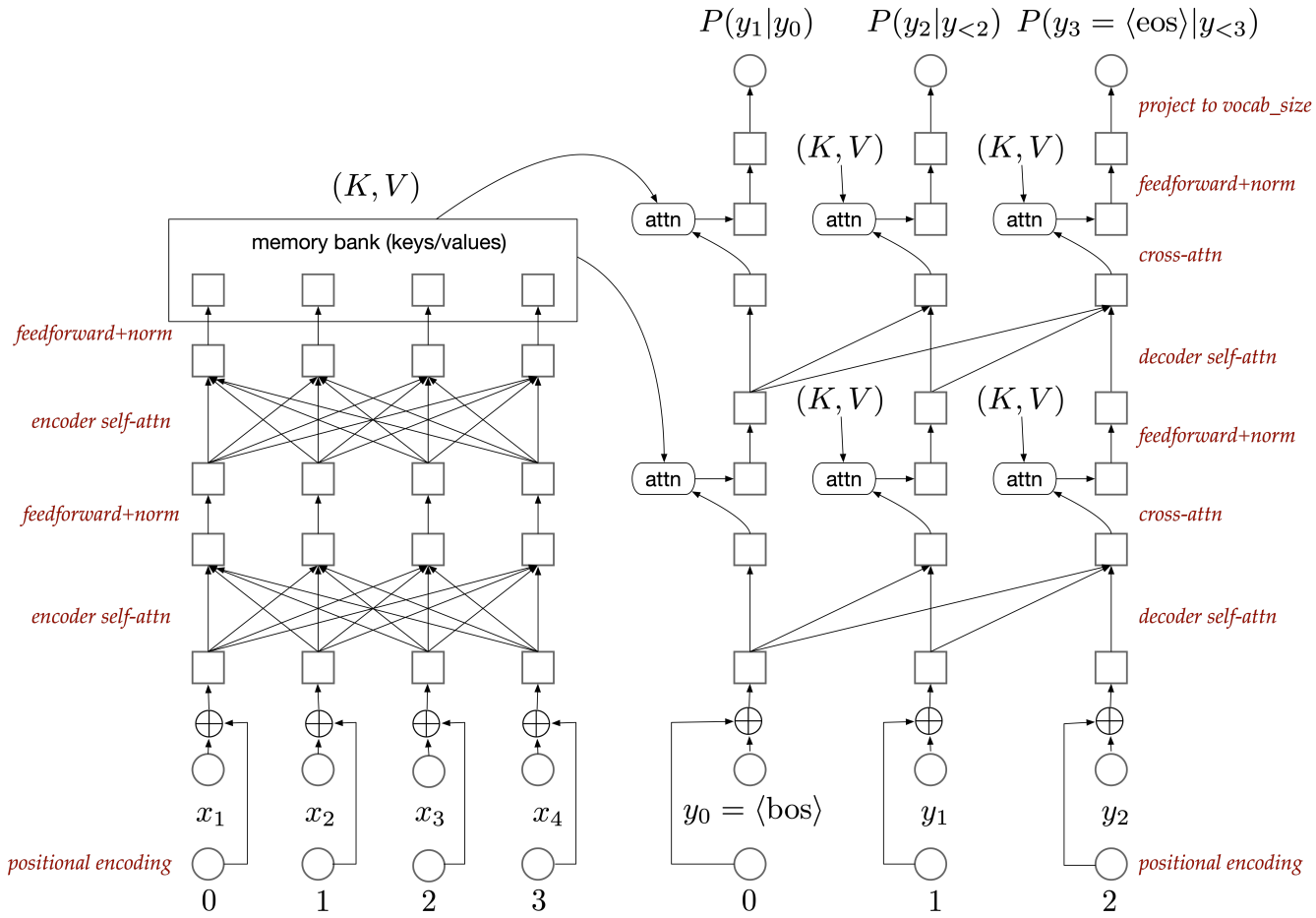
The process we describe above is only a single layer of attention. In practice, transformers stack multiple layers of attention and feedforward layers, using the outputs from the layer below as the inputs to the layer above, as shown in the illustration below.



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

Type your answer here, replacing this text.

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



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

Causal attention mask

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

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

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

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

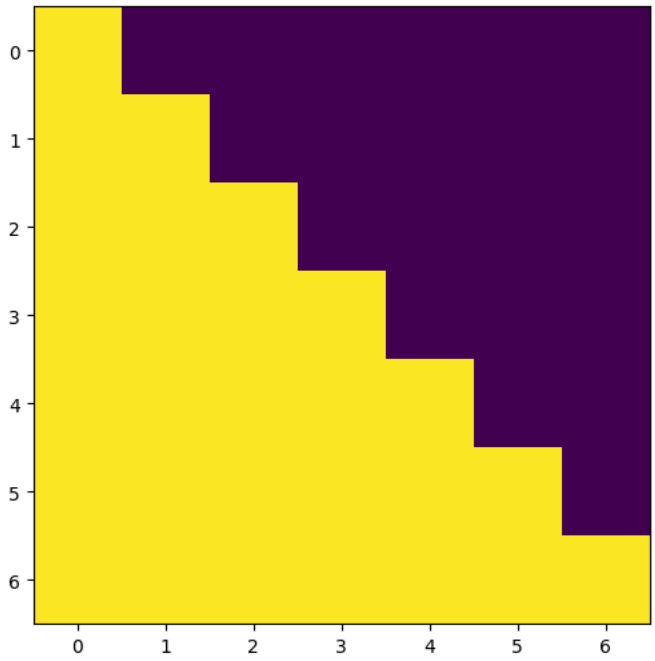
```
#TODO - implement this function, which returns a causal attention mask
def causal_mask(T):
  """
  Generate a causal mask.
  Arguments:
      T: the length of target sequence
  Returns:
      mask: a T x T tensor, where `mask[i, j]` should be `True`
      if y_i can attend to y_{j-1} (there's a "-1"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


grader.check("causal_attention_mask")
```

```
All tests passed!
```

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

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

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

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

```
<matplotlib.image.AxesImage at 0x7a81bef4b9d0>
```



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

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

> Hint: Completing this transformer implementation should require very little code, just a few lines.

> Hint: The causal mask is a 2-D matrix, but we want to add a batch dimension, and expand it to be of the desired size. For this purpose, you can use `torch.repeat`.

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

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

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

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

  def forward_encoder(self, src, src_lengths):
    """
    Encodes source words `src`.
    Arguments:
        src: src batch of size (bsz, max_src_len)
        src_lengths: src lengths (bsz)
    Returns:
        memory_bank: a tensor of size (bsz, src_len, hidden_size)
    """
    # The reason we don't directly pass in src_mask as in `forward_decoder` is to
    # enable us to reuse beam search implemented for RNN-based encoder-decoder
    src_len = src.size(1)
    #TODO - compute `encoder_self_attn_mask`
    encoder_self_attn_mask = src.ne(self.padding_id_src).unsqueeze(1).repeat(1, src_len, 1)
    memory_bank = self.encoder(src, encoder_self_attn_mask)
    return memory_bank, None

  def forward_decoder(self, tgt_in, memory_bank, src_mask):
    """
    Decodes based on memory bank, and ground truth target words.
    Arguments:
        tgt_in: a tensor of size (bsz, tgt_len)
        memory_bank: a tensor of size (bsz, src_len, hidden_size), encoder outputs
                     at every position
        src_mask: a tensor of size (bsz, src_len) which is `False` for source paddings
    Returns:
        Logits of size (bsz, tgt_len, V_tgt) (before the softmax operation)
    """
    tgt_len = tgt_in.size(1)
    bsz = tgt_in.size(0)
    #TODO - compute `cross_attn_mask` and `decoder_self_attn_mask`
    cross_attn_mask = src_mask.unsqueeze(1).repeat(1, tgt_len, 1)
    decoder_self_attn_mask = causal_mask(tgt_len).unsqueeze(0).repeat(bsz, 1, 1)
```

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

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

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask, normalize=True):
        """
        Forward the decoder at `decoder_state` for a single step with token `tgt_in_onestep`.
        This function will be used in beam search. Note that the implementation here is
        very inefficient, since we do not cache any decoder state, but instead we only
        cache previously generated tokens in `prev_decoder_states`, and do a fresh
        `forward_decoder`.
        Arguments:
            prev_decoder_states: previous tgt words. None for the first step.
            tgt_in_onestep: a tensor of size (bsz), tokens at one step
            memory_bank: a tensor of size (bsz, src_len, hidden_size), src hidden states
                         at every position
            src_mask: a tensor of size (bsz, src_len): a boolean tensor, `False` where
                      src is padding.
            normalize: use log_softmax to normalize or not. Beam search needs to normalize,
                       while `forward_decoder` does not
        Returns:
            logits: Log probabilities for `tgt_in_token` of size (bsz, V_tgt)
            decoder_states: we use tgt words up to now as states, a tensor of size (bsz, len)
            None: to keep output format the same as AttnEncoderDecoder, such that we can
                  reuse beam search code

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

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


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

    def __init__(self, vocab_size, hidden_size, layers):
        super().__init__()
        self.embed = PositionalEmbedding(vocab_size, hidden_size)
        encoder_layer = TransformerEncoderLayer(hidden_size)
        self.layers = _get_clones(encoder_layer, layers)
        self.norm = nn.LayerNorm(hidden_size)

    def forward(self, src, encoder_self_attn_mask):
        r"""Pass the input through the word embedding layer, followed by
        the encoder layers in turn.
        Arguments:
            src: src batch of size (bsz, max_src_len)
            encoder_self_attn_mask: the mask for encoder self-attention, it's of size
                                    (bsz, max_src_len, max_src_len)
        Returns:
```

```
        a tensor of size (bsz, max_src_len, hidden_size)
    """
    output = self.embed(src)
    for mod in self.layers:
      output = mod(output, encoder_self_attn_mask=encoder_self_attn_mask)
    output = self.norm(output)
    return output


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

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

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


  def forward(self, src, encoder_self_attn_mask):
    r"""Pass the input through the encoder layer.
    Arguments:
        src: an input tensor of size (bsz, max_src_len, hidden_size).
        encoder_self_attn_mask: attention mask of size (bsz, max_src_len, max_src_len),
                                it's `False` where the corresponding attention is disabled
    Returns:
        a tensor of size (bsz, max_src_len, hidden_size).
    """
    # Attend
    q = self.q_proj(src) / math.sqrt(self.hidden_size) # a trick needed to make transformer work
    k = self.k_proj(src)
    v = self.v_proj(src)
    #TODO - compute `context`
    attn_scores = torch.bmm(q, k.transpose(1, 2))  # (bsz, max_src_len, max_src_len)
    attn_scores = attn_scores.masked_fill(~encoder_self_attn_mask, float('-inf'))
    attn_weights = torch.softmax(attn_scores, dim=-1)
    context = torch.bmm(attn_weights, v)  # (bsz, max_src_len, hidden_size)
    src2 = self.context_proj(context)
    # Residual connection
    src = src + src2
    src = self.norm1(src)
    # Feedforward for each position
    src2 = self.linear2(self.activation(self.linear1(src)))
    src = src + src2
    src = self.norm2(src)
    return src


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

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

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

        output = self.norm(output)
        return output


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

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

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

        self.activation = nn.ReLU()

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

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

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

      def forward(self, tgt, memory, cross_attn_mask, decoder_self_attn_mask):
        r"""Pass the inputs (and mask) through the decoder layer.
        Arguments:
            tgt: an input tensor of size (bsz, max_tgt_len, hidden_size).
            memory: encoder outputs of size (bsz, max_src_len, hidden_size).
            cross_attn_mask: attention mask of size (bsz, max_tgt_len, max_src_len),
                             it's `False` where the cross-attention is disallowed.
            decoder_self_attn_mask: attention mask of size (bsz, max_tgt_len, max_tgt_len),
                                    it's `False` where the self-attention is disallowed.
        Returns:
            a tensor of size (bsz, max_tgt_len, hidden_size)
        """
        # Self attention (decoder-side)
        q = self.q_proj_self(tgt) / math.sqrt(self.hidden_size)
        k = self.k_proj_self(tgt)
        v = self.v_proj_self(tgt)
        #TODO - compute `context`
        attn_scores = torch.bmm(q, k.transpose(1, 2))
        attn_scores = attn_scores.masked_fill(~decoder_self_attn_mask, float('-inf'))
        attn_weights = torch.softmax(attn_scores, dim=-1)
        context = torch.bmm(attn_weights, v)
        tgt2 = self.context_proj_self(context)
        tgt = tgt + tgt2
        tgt = self.norm1(tgt)
        # Cross attention (decoder attends to encoder)
        q = self.q_proj_cross(tgt) / math.sqrt(self.hidden_size)
        k = self.k_proj_cross(memory)
        v = self.v_proj_cross(memory)
        #TODO - compute `context`
        attn_scores = torch.bmm(q, k.transpose(1, 2))  # (bsz, max_tgt_len, max_src_len)
        attn_scores = attn_scores.masked_fill(~cross_attn_mask, float('-inf'))
        attn_weights = torch.softmax(attn_scores, dim=-1)
        context = torch.bmm(attn_weights, v)  # (bsz, max_tgt_len, hidden_size)
```

```
    tgt2 = self.context_proj_cross(context)
    tgt = tgt + tgt2
    tgt = self.norm2(tgt)
    tgt2 = self.linear2(self.activation(self.linear1(tgt)))
    tgt = tgt + tgt2
    tgt = self.norm3(tgt)
    return tgt

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

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

  def forward(self, batch):
    x = self.embed(batch) * math.sqrt(self.embedding_size) # type embedding
    # Add positional encoding to type embedding
    x = x + self.pe[:, :x.size(1)].detach()
    return x


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


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

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

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

```
100%|██████████| 2032/2032 [02:51<00:00, 11.88it/s]
Epoch: 0 Training Perplexity: 1.7230 Validation Perplexity: 1.1467
100%|██████████| 2032/2032 [02:42<00:00, 12.49it/s]
Epoch: 1 Training Perplexity: 1.1286 Validation Perplexity: 1.0453
<All keys matched successfully>
```

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

Question: We argued above that *training* transformers can be much faster than training RNNs. What about *generation* using transformers? Would there be any speed advantage of decoding (generation) using transformers compared to RNNs? Why or why not?

During training, transformers are significantly faster than RNNs because they process entire sequences in parallel. This allows them to utilize GPUs efficiently and handle larger datasets within the same computational budget. However, when it comes to generation (decoding), the speed advantage is not as significant.

In text generation, models like GPT use auto-regressive decoding, meaning they generate one token at a time. This process is sequential since each new token depends on the previously generated tokens. Unlike training, where transformers can process all tokens simultaneously, generation cannot be fully parallelized. As a result, transformers do not have a major speed advantage over RNNs during decoding.

In conclusion, while transformers are much faster for training, their speed benefit during generation is limited due to the sequential nature of the decoding process.

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

```
Test perplexity: 1.049
```

```
grader.check("transformer_ppl")
```

```
All tests passed!
```

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

```
grader.check("transformer_beam_search")
```

```
All tests passed!
```

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

correct = 0
total = 0

# create beam searcher
beam_searcher = BeamSearcher(model_transformer)

for index, batch in enumerate(test_iter, start=1):
  # Input and output
  src = batch['src_ids']
  src_lengths = batch['src_lengths']
  # Predict
  model.all_attns = []
  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)
  if DEBUG_FIRST > index:
    src = hf_src_tokenizer.decode(src[0], skip_special_tokens=True)
    print (f'Source: {src}')
    print (f'Prediction:   {prediction}')
    print (f'Ground truth: {ground_truth}')
  if ground_truth == prediction:
    correct += 1
  total += 1

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

```
Source: sixteen thousand eight hundred and thirty two
Prediction:   1 6 8 3 2
Ground truth: 1 6 8 3 2
Source: 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: six thousand two hundred and twelve
Prediction:   6 2 1 2
Ground truth: 6 2 1 2
Source: seven hundred and ninety eight million three hundred and thirty one thousand eight hundred and eighteen
Prediction:   7 9 8 3 3 1 8 1 8
Ground truth: 7 9 8 3 3 1 8 1 8
Source: 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: 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: 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: ninety seven thousand seven hundred and sixty two
Prediction:   9 7 7 6 2
Ground truth: 9 7 7 6 2
Source: four hundred and ten thousand two hundred and three
Prediction:   4 1 0 2 0 3
Ground truth: 4 1 0 2 0 3
Accuracy: 0.89
```

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

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

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

In a multi-layer transformer, attention scores can still be interpreted, but their meaning is more complex than in an RNN with attention.

In an RNN with attention, the scores α i α iclearly indicate which input tokens are most relevant to the current decoding step. Since attention is applied directly to the input sequence, we can interpret high attention weights as a strong alignment between specific input words and output words.

In a transformer, attention is applied at multiple layers and heads, meaning that each head in each layer can focus on different aspects of the input. Some heads might capture syntactic relationships (e.g., subject-verb connections), while others might capture semantic similarities. Because of this complexity, attention scores in transformers do not always provide a clear, single alignment between input and output words.

However, we can still analyze attention maps to gain insights into how the model processes information. For example, in early layers, attention scores may correspond to local word dependencies, while in later layers, they might capture broader sentence-level meanings.

In conclusion, while attention scores in transformers can still be analyzed, they do not have the same direct interpretability as in an RNN with attention, because they operate across multiple layers and attention heads, capturing different types of relationships in the data.

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

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

In real-world applications, many state-of-the-art NLP approaches are based on transformers. For further (relatively old, but still very relevant) readings if you are interested, we recommend BERT and GPT-3.

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? <!-- BEGIN QUESTION name: open_response_debrief manual: true
- ->

This lab was a valuable learning experience, allowing me to explore key concepts related to transformers and attention mechanisms. The exercises provided a good balance between theory and practice, helping to reinforce my understanding through hands-on implementation.

One of the strongest aspects of the lab was how it demonstrated the differences between RNN-based attention and transformer attention. It made clear why transformers are more efficient for certain tasks and how attention mechanisms evolve across multiple layers. The structured exercises guided me well through the concepts, making them more concrete.

However, there were some areas that could be improved. The lab was a bit long, and some explanations could have been more concise to improve efficiency. A few additional clarifications, especially on how to interpret multi-head attention weights in deep transformers, would have been helpful. An example with a more intuitive breakdown of attention scores across layers could make the learning process even smoother.

Overall, the lab was well-designed and provided a solid foundation for understanding attention mechanisms in modern deep learning architectures.

End of Lab 4-4 {-}

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

```
grader.check_all()
```

```
beam_search:
All tests passed!
causal_attention_mask:
All tests passed!
encoder_decoder_ppl:
All tests passed!
transformer_beam_search:
All tests passed!
transformer_ppl:
All tests passed!
```