```
# Downloads required packages and files
required_files = "https://github.com/jhu-intro-hlt/jhu-intro
! wget $required_files && unzip -o required_files.zip
! pip install -r requirements.txt
```

```
--2025-10-29 01:50:39--  https://github.com/jhu-intro-hlt/jhu-intro-hlt.githu
Resolving github.com (github.com)... 140.82.121.3
Connecting to github.com (github.com)|140.82.121.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/jhu-intro-hlt/jhu-intro-hlt.githu
--2025-10-29 01:50:39--  https://raw.githubusercontent.com/jhu-intro-hlt/jhu-
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.10
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.1
HTTP request sent, awaiting response... 200 OK
Length: 5675 (5.5K) [application/zip]
Saving to: ‘required_files.zip’

required_files.zip  100%[===================>]   5.54K  --.-KB/s    in 0s

2025-10-29 01:50:39 (75.2 MB/s) - ‘required_files.zip’ saved [5675/5675]

Archive:  required_files.zip
   creating: __MACOSX/
  inflating: __MACOSX/._requirements.txt
  inflating: __MACOSX/._tests
   creating: __MACOSX/tests/
  inflating: __MACOSX/tests/._metric-cer-impl.py
  inflating: __MACOSX/tests/._metric-torchmetric-impl.py
  inflating: __MACOSX/tests/._model-generate-checks.py
  inflating: __MACOSX/tests/._model-generate-test.py
  inflating: requirements.txt
   creating: tests/
  inflating: tests/metric-cer-impl.py
  inflating: tests/metric-torchmetric-impl.py
  inflating: tests/model-generate-checks.py
  inflating: tests/model-generate-test.py
Collecting datascience (from -r requirements.txt (line 1))
  Downloading datascience-0.18.0-py3-none-any.whl.metadata (910 bytes)
Requirement already satisfied: jupyter_client in /usr/local/lib/python3.12/di
Requirement already satisfied: ipykernel in /usr/local/lib/python3.12/dist-pa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-p
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packa
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.12/dist-p
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packag
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-pack
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packa
Requirement already satisfied: nbconvert in /usr/local/lib/python3.12/dist-pa
Requirement already satisfied: nbformat in /usr/local/lib/python3.12/dist-pac
Requirement already satisfied: dill in /usr/local/lib/python3.12/dist-package
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packag
Collecting otter-grader==4.0.1 (from -r requirements.txt (line 14))
  Downloading otter_grader-4.0.1-py3-none-any.whl.metadata (2.8 kB)
Collecting pdfkit (from -r requirements.txt (line 16))
```

```
Collecting pdfkit (from -r requirements.txt (line 16))
  Downloading pdfkit-1.0.0-py3-none-any.whl.metadata (9.3 kB)
Collecting PyPDF2 (from -r requirements.txt (line 17))
  Downloading pypdf2-3.0.1-py3-none-any.whl.metadata (6.8 kB)
Collecting wkhtmltopdf (from -r requirements.txt (line 18))
  Downloading wkhtmltopdf-0.2.tar.gz (9.7 kB)
  Preparing metadata (setup.py) ... done
Collecting overrides===6.2.0 (from -r requirements.txt (line 21))
  Downloading overrides-6.2.0-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packag
Requirement already satisfied: torchvision in /usr/local/lib/python3.12/dist-
```

```
# Initialize Otter
import otter

grader = otter.Notebook(colab=True)
```

Assignment 3

You have now learnt about sequence-to-sequence models in the context of machine translation. In the upcoming lectures, you will see that these models are useful for a wide variety of tasks, wherever the source sequence and the target sequence have different lengths. You have also been introduced to phonemes, which are the building blocks of speech.

In this assignment, you will build sequence-to-sequence models for pronunciation prediction of English words, which simply means that given a word (sequence of characters), the model should predict its pronunciation (sequence of phonemes). You should be able to see why this is a straightforward application of sequence-to-sequence models.

The input is a sequence of characters making up an English word e.g. `a l g e b r a i c a l l y`. The output should be a sequence of phonemes that describe the pronunciation. For the example above, the desired output should be `AE2 L JH AH0 B R EY1 IH0 K L IY0`. The data for this task was obtained from the CMU Pronunciation Dictionary. We will use a small subset of the CMU dict data.

Setup

For this assignment, as in the previous one, we will be using Google Colab, for both code as well as descriptive questions. Your task is to finish all the questions in the Colab notebook and then upload a PDF version of the notebook, and a viewable link on Gradescope.

Google colaboratory

Before getting started, get familiar with google colaboratory: https://colab.research.google.com/notebooks/welcome.ipynb

This is a neat python environment that works in the cloud and does not require you to set up anything on your personal machine (it also has some built-in IDE features that make writing code easier). Moreover, it allows you to copy any existing collaboratory file, alter it and share with other people.

Note:

1. You may need to change your Runtime setting to GPU in order to run the following code blocks.
2. On changing the Runtime setting, you would be required to run the previous code-blocks again.

Submission

Before you start working on this homework do the following steps:

1. Press **File > Save a copy in Drive...** tab. This will allow you to have your own copy and change it.
2. Follow all the steps in this collaboratory file and write / change / uncomment code as necessary.
3. Do not forget to occasionally press **File > Save** tab to save your progress.
4. After all the changes are done and progress is saved press **Share** button (top right corner of the page), press **get shareable link** and make sure you have the option **Anyone with the link can view** selected. Copy the link and paste it in the box below.
5. After completing the notebook, press **File > Download .ipynb** to download a local copy on your computer, and then upload the file to Gradescope.
6. Please export the notebook to PDF and upload the PDF to the writing part.

Special handling for model checkpoints.

6. As the homework requires training neural models, such trained model checkpoints should also be submitted together with the notebook, hence avoiding re-training during the grading phase. For such model checkpoints, they would be stored at `./lightning_logs` directory. You have to first locate the directory from the left side panel (`Files`) on Colab.
7. Enter `./lightning_logs` and find the training label that you would like to

submit. The versions are labelled with respect to the training calls.

8. Download the `.ckpt` file from `./lightning_logs/<your_version>/checkpoints/<name>.ckpt`.
9. Rename the downloaded checkpoint and re-name it as the corresponding name shown in the question, say `vanilla_rnn_model.ckpt`.
10. Submit checkpoint file(s) together with your notebook to the autograder. Please make sure that checkpoint files should be put at the same directory level as the notebook (content root).
11. Please enter your leaderboard name the same as how it is written on your gradescope account.

Paste your notebook link in the box below. *(0 points)*

https://colab.research.google.com/drive/1BkuXg2IoTrMYeCASA0fNEllvazWPRirD?usp=sharing

```
import os
import urllib
import enum
import json
from itertools import zip_longest
from dataclasses import dataclass
from typing import Optional, List, Dict, Tuple, Any, Union

import torch
import torchmetrics
import pytorch_lightning as pl
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from transformers import PreTrainedModel, PretrainedConfig,
from transformers.modeling_outputs import BaseModelOutput,
```

```
# Checks whether it is in the autograder grading mode
# Checks whether GPU accelerators are available
is_autograder = os.path.exists('is_autograder.py')
if torch.cuda.is_available() and not is_autograder:
    accelerator = 'gpu'
else:
    accelerator = 'cpu'
print(f'The notebook is running for "{"autograder" if is_au
```

```
print('Students should make sure you are running under the
print(f'You are using "{accelerator}".')
```

```
The notebook is running for "student".
Students should make sure you are running under the "student" mode.
You are using "gpu".
```

```
# Seed everything to make sure all experiments are reproduc
pl.seed_everything(seed=777)
```

```
INFO:lightning_fabric.utilities.seed:Seed set to 777
777
```

```
# Defines constants
HOMEWORK_DATA_URL = "https://github.com/jhu-intro-hlt/jhu-i

SRC_SUFFIX = '.src'
TGT_SUFFIX = '.tgt'

CMUDICT_BASE = 'cmudict'
CMUDICT_TRAIN_SMALL = CMUDICT_BASE + '.small.train'
CMUDICT_TRAIN = CMUDICT_BASE + '.train'
CMUDICT_DEV = CMUDICT_BASE + '.dev'

CMUDICT_SRC_VOCAB = 'cmudict.src.vocab.json'
CMUDICT_TGT_VOCAB = 'cmudict.tgt.vocab.json'

# Special tokens
class SpecialToken(enum.Enum):
    BOS = '<BOS>'
    EOS = '<EOS>'
    UNK = '<UNK>'
    PAD = '<PAD>'
```

```
def download_data():
    def _download(url: str, filename: str) -> str:
        txt = urllib.request.urlopen(url)
        with open(filename, 'w') as f:
            f.write(txt.read().decode('utf-8'))

    for suffix in (SRC_SUFFIX, TGT_SUFFIX):
        _download(f'{HOMEWORK_DATA_URL}/{CMUDICT_TRAIN_SMAL
        _download(f'{HOMEWORK_DATA_URL}/{CMUDICT_TRAIN}{suf
```

```
            _download(f'{HOMEWORK_DATA_URL}/{CMUDICT_DEV}{suffi
            _download(f'{HOMEWORK_DATA_URL}/{CMUDICT_SRC_VOCAB}
            _download(f'{HOMEWORK_DATA_URL}/{CMUDICT_TGT_VOCAB}

    download_data()
```

Vocabulary

In the previous homework, we did not specifically define a class to serve as `Vocabulary`. In this homework, as we have to deal with both the source side vocabulary and target side vocabulary, which are different, it is easier to have a class to perform the index-string and string-index mappings.

Although there is nothing for you to implement, we suggest you to walk through the whole implementation to understand each function and think about what they might be used. If you are not familiar with any concepts here, you should either figure it out from previous homework or post questions on Piazza.

```
    class Vocabulary(object):
        def __init__(self,
                     tokens: Optional[List[str]] = None):
            # Registers `SpecialToken`
            self.special_token_enum = SpecialToken
            self._special_tokens = set([s.value for s in Specia
            if tokens is not None:
                tokens = set(tokens) | set(self._special_tokens
            else:
                tokens = set(self._special_tokens)
            self._idx2token = list(tokens)
            self._token2idx = {t: i for i, t in enumerate(self.

        def add_token(self, token: str) -> int:
            if token not in self._idx2token:
                self._token2idx[token] = len(self._idx2token)
                self._idx2token.append(token)
            return self.token2idx(token)

        def to_file(self, file_path: str):
            with open(file_path, 'w') as f:
                json.dump({
```

```
            json.dump({
                'idx2token': self._idx2token,
                'special_tokens': list(self._special_tokens
                'token2idx': self._token2idx
            }, f, indent=2)

    @classmethod
    def from_file(cls, file_path: str) -> 'Vocabulary':
        with open(file_path) as f:
            read_data = json.load(f)
        vocab = Vocabulary()
        vocab._special_tokens = set(read_data['special_toke
        vocab._idx2token = read_data['idx2token']
        vocab._token2idx = read_data['token2idx']
        return vocab

    def is_special(self, token: str) -> bool:
        return token in self._special_tokens

    def token2idx(self, token: str) -> int:
        return self._token2idx.get(token, self._token2idx[s

    def idx2token(self, index: int) -> str:
        return self._idx2token[index]

    def bos(self) -> SpecialToken:
        return self.special_token_enum.BOS

    def eos(self) -> SpecialToken:
        return self.special_token_enum.EOS

    def unk(self) -> SpecialToken:
        return self.special_token_enum.UNK

    def pad(self) -> SpecialToken:
        return self.special_token_enum.PAD

    def special_to_id(self, st: SpecialToken) -> int:
        return self.token2idx(token=str(st.value))

    def bos_id(self) -> int:
        return self.special_to_id(st=self.bos())
```

```
    def eos_id(self) -> int:
        return self.special_to_id(st=self.eos())

    def unk_id(self) -> int:
        return self.special_to_id(st=self.unk())

    def pad_id(self) -> int:
        return self.special_to_id(st=self.pad())

    def __len__(self):
        return len(self._idx2token)
```

```
def decode_as_str(tgt_vocab: Vocabulary, output_tensor: tor
    """Decodes generation outputs to a list of strings."""
    outputs: List[List[int]] = output_tensor.detach().tolis
    ignore_token_ids = (tgt_vocab.pad_id(), tgt_vocab.bos_i
    decoded_strs: List[str] = [
        ' '.join([
            tgt_vocab.idx2token(index=tid)  # Converts toke
            for tid in b
            if tid not in ignore_token_ids  # Filters speci
        ])  # Creates a decoded str
        for b in outputs  # Iterates over the batch
    ]
    return decoded_strs


def encode_as_tensor(src_vocab: Vocabulary, sentence: str)
    """Encodes a sentence to a tensor via the source vocabu
    return torch.tensor(
        [
            [src_vocab.bos_id()]
            + [
                src_vocab.token2idx(token)
                for token in sentence.strip().split()
            ]
            + [src_vocab.eos_id()]
        ],
        dtype=torch.long
    )
```

```
    # We have pre-built vocabs to ensure deterministic mappings
    # So we can directly load these vocabs from files
    cmudict_src_vocab = Vocabulary.from_file(CMUDICT_SRC_VOCAB)
    cmudict_tgt_vocab = Vocabulary.from_file(CMUDICT_TGT_VOCAB)
```

Dataset

For your convenience, we have preselected a subset of the CMU pronunciation dictionary (input-output pairs) and split the subset into training, validation (dev) and test sets.

Data Reader

We will use the `ParallelDataset` class below to load, process and iterate through the data. Since you worked quite a bit on data loading implementation in the previous assignment, we will provide you the loader for this one, so you can get on with the more interesting parts of this assignment. Why is it parallel? Each source sequence is paired with a target sequence. The task is to generate a target sequence given the source sequence.

Note the use of special symbols `<BOS>`, `<EOS>`, `<UNK>`, and `<PAD>`. All the sequences (both input and output) are made to begin with `<BOS>` (Begin of sequence) and end with `<EOS>` (End of sequence). The `<UNK>` symbol is used if we encouter any new symbol that we have not seen (unknown symbol).

```
    @dataclass
    class ParallelInstance:
        src_seq: torch.Tensor  # Shape: (seq_len)
        src_tokens: List[str]
        tgt_seq: Optional[torch.Tensor]  # Shape: (seq_len)
        tgt_tokens: Optional[List[str]]


    @dataclass
    class ParallelBatch:
        src_seqs: torch.Tensor  # Shape: (batch_size, src_seq_l
        src_attention_mask: torch.Tensor  # Shape: (batch_size,
        src_tokens: List[List[str]]
```

```
        src_tokens: List[List[str]]
        tgt_seqs: Optional[torch.Tensor]  # Shape: (batch_size,
        tgt_attention_mask: Optional[torch.Tensor]  # Shape: (b
        tgt_tokens: Optional[List[List[str]]]


    class ParallelDataset(Dataset):
        """A dataset class that reads the parallel data.
        """

        def __init__(self,
                     src_file: str,
                     tgt_file: Optional[str] = None,
                     src_vocab: Optional[Vocabulary] = None,
                     tgt_vocab: Optional[Vocabulary] = None):
            """
            Parameters
            ----------
            src_file : str
                Path to the source side file.
            tgt_file : str, optional
                Path to the target side file. During the testin
            src_vocab : Vocabulary, optional
                An existing `Vocabulary` object for the source
                one will be created by iterating over source da
            tgt_vocab : Vocabulary, optional
                An existing `Vocabulary` object for the target
                one will be created by iterating over target da
            """
            src_token_sequences = self.read_data(src_file)
            tgt_token_sequences = self.read_data(tgt_file) if t
            self.src_vocab = src_vocab if src_vocab is not None
            self.tgt_vocab = tgt_vocab if tgt_vocab is not None
            src_tensors = self.tensorize(vocab=self.src_vocab,
            tgt_tensors = self.tensorize(vocab=self.tgt_vocab,
                                         sequences=tgt_token_se

            self.instances: List[ParallelInstance] = [
                ParallelInstance(s, st, t, tt)
                for s, t, st, tt in zip_longest(
                    src_tensors, tgt_tensors, src_token_sequenc
                    fillvalue=None
```

```
            )
        ]

    def tensorize(self, vocab: Vocabulary, sequences: List[
        indexed_tensors: List[torch.Tensor] = []
        for seq in sequences:
            # Shape: (1, seq_len)
            new_tensor = torch.tensor(
                [vocab.token2idx(t) for t in self.add_speci
                dtype=torch.long
            )
            indexed_tensors.append(new_tensor)
        return indexed_tensors

    @staticmethod
    def read_data(filepath: str) -> List[List[str]]:
        data: List[List[str]] = []
        with open(filepath, 'r', encoding='utf8') as f:
            for l in f:
                d = [tok for tok in l.strip().split()]
                data.append(d)
        return data

    @staticmethod
    def add_special_tokens(vocab: Vocabulary, seq: List[str
        return [str(vocab.bos().value)] + seq + [str(vocab.

    @staticmethod
    def build_vocab(sequences: List[List[str]]) -> Vocabula
        return Vocabulary(tokens=[t for s in sequences for

    def __len__(self) -> int:
        """Returns the number of instances read in the data
        return len(self.instances)

    def __getitem__(self, index: int) -> ParallelInstance:
        return self.instances[index]
```

```
class ParallelDataModule(pl.LightningDataModule):
    """Wraps PyTorch dataset as a lightning data module."""

    def __init__(self
```

```
def __init__(self,
             dataset_paths: Dict[str, str],
             batch_size: int = 32,
             shuffle: bool = True,
             src_vocab: Optional[Vocabulary] = None,
             tgt_vocab: Optional[Vocabulary] = None):
    super(ParallelDataModule, self).__init__()

    self.datasets: Dict[str, Dataset] = {}
    self.src_vocab = src_vocab
    self.tgt_vocab = tgt_vocab
    for split in ('train', 'val', 'test'):
        if split not in dataset_paths:
            continue
        split_path = dataset_paths[split]
        new_dataset = ParallelDataset(src_file=split_pa
                                      tgt_file=split_pa
                                      src_vocab=self.sr
                                      tgt_vocab=self.tg
        if self.src_vocab is None:
            self.src_vocab = new_dataset.src_vocab
        if self.tgt_vocab is None:
            self.tgt_vocab = new_dataset.tgt_vocab
        self.datasets[split] = new_dataset

    self.batch_size = batch_size
    self.shuffle = shuffle

@staticmethod
def _pad_sequence(seq: torch.Tensor,
                  max_length: int,
                  padding_value:
                  Union[int, float]
                  ) -> torch.Tensor:
    seq_len = seq.shape[-1]
    if seq_len < max_length:
        return torch.cat(
            [seq, torch.tensor([padding_value] * (max_l
            dim=-1
        )
    else:
        return seq
```

```
    def collate_fn(self, instances: List[ParallelInstance])
        """Collates a list of instances and composes a batc

        Parameters
        ----------
        instances : List[ParallelInstance]
            A list of `ParallelInstance` to comprise.

        Returns
        -------
        batch : ParallelBatch
            A single `ParallelBatch` where tensors are batc
        """
        max_src_seq_len = max([x.src_seq.shape[0] for x in
        has_tgt = instances[0].tgt_seq is not None
        max_tgt_seq_len = max([x.tgt_seq.shape[0] for x in
        return ParallelBatch(
            src_seqs=torch.stack(
                [
                    self._pad_sequence(x.src_seq, max_lengt
                                       padding_value=self.s
                    for x in instances
                ],
                dim=0
            ),
            src_attention_mask=torch.stack(
                [
                    self._pad_sequence(
                        seq=torch.ones_like(x.src_seq, dtyp
                        max_length=max_src_seq_len,
                        padding_value=0
                    )
                    for x in instances
                ],
                dim=0
            ),
            tgt_seqs=torch.stack(
                [
                    self._pad_sequence(x.tgt_seq, max_lengt
                                       padding_value=self.t
                    for x in instances
```

```
                ],
                dim=0
            ) if has_tgt else None,
            tgt_attention_mask=torch.stack(
                [
                    self._pad_sequence(
                        seq=torch.ones_like(x.tgt_seq, dtyp
                        max_length=max_tgt_seq_len,
                        padding_value=0
                    )
                    for x in instances
                ],
                dim=0
            ) if has_tgt else None,
            src_tokens=[x.src_tokens for x in instances],
            tgt_tokens=[x.tgt_tokens for x in instances] if
        )

    def train_dataloader(self):
        return DataLoader(self.datasets['train'],
                          batch_size=self.batch_size,
                          shuffle=self.shuffle,
                          collate_fn=lambda x: self.collate

    def val_dataloader(self):
        return DataLoader(self.datasets['val'],
                          batch_size=self.batch_size,
                          shuffle=False,
                          collate_fn=lambda x: self.collate

    def test_dataloader(self):
        return DataLoader(self.datasets['test'],
                          batch_size=self.batch_size,
                          shuffle=False,
                          collate_fn=lambda x: self.collate
```

```
cmudict_corpus = ParallelDataModule(
    dataset_paths={'train': CMUDICT_TRAIN, 'val': CMUDICT_D
    src_vocab=cmudict_src_vocab,
    tgt_vocab=cmudict_tgt_vocab
)
```

```
# Please take a look at the output to get the sense of what
for i, x in enumerate(cmudict_corpus.train_dataloader()):
    if i > 0:
        break
    print(x)
```

```
ParallelBatch(src_seqs=tensor([[18,  9, 26, 10, 10, 22,  7, 30, 11, 23,  6, 1
       [18, 28, 23, 14, 26, 11, 21, 12, 22, 26,  7, 22, 13, 22,  7, 30,  6],
       [18, 21,  7, 19, 21,  5, 11, 22,  2, 25, 23,  4,  6, 15, 15, 15, 15],
       [18, 32,  1,  7,  4,  1, 25,  1, 28,  6, 15, 15, 15, 15, 15, 15, 15],
       [18, 30, 28,  1,  7,  4, 22, 11, 11, 26,  6, 15, 15, 15, 15, 15, 15],
       [18, 19, 28, 26, 12, 26, 12, 20, 19, 22,  9,  1, 11,  6, 15, 15, 15],
       [18,  5, 28, 23,  1, 12, 25, 12,  1, 32, 22,  7, 30,  6, 15, 15, 15],
       [18,  5, 23,  7, 23,  4, 22,  9, 12,  1,  6, 15, 15, 15, 15, 15, 15],
       [18, 21,  7, 26, 29, 29, 22,  9, 22,  1, 11,  6, 15, 15, 15, 15, 15],
       [18,  2, 12, 21,  7,  7, 22,  7, 30, 11, 20,  6, 15, 15, 15, 15, 15],
       [18,  4, 22,  2,  9, 21,  2,  2, 23,  2,  6, 15, 15, 15, 15, 15, 15],
       [18, 28, 23,  9, 26,  7, 29, 22, 30, 21, 28, 23,  4,  6, 15, 15, 15],
       [18, 29, 28, 21,  2, 12, 28,  1, 12, 23,  6, 15, 15, 15, 15, 15, 15],
       [18, 11, 22, 30,  1, 10, 23,  7, 12,  2,  6, 15, 15, 15, 15, 15, 15],
       [18, 12, 28,  1,  7,  2,  2, 23, 31, 21,  1, 11,  2,  6, 15, 15, 15],
       [18, 25, 21, 10,  1,  7, 22, 12, 20,  3,  2,  6, 15, 15, 15, 15, 15],
       [18,  9, 26, 11, 26,  7, 22,  1, 11, 22,  2, 12,  2,  6, 15, 15, 15],
       [18, 21,  7,  2,  1, 12, 22,  2, 29,  1,  9, 12, 26, 28, 20,  6, 15],
       [18, 26, 19, 25, 12, 25,  1, 11, 10, 26, 11, 26, 30, 20,  6, 15, 15],
       [18, 10, 26,  7, 12, 29, 26, 28,  4,  6, 15, 15, 15, 15, 15, 15, 15],
       [18,  5, 22, 26, 30, 28,  1, 19, 25, 23, 28,  6, 15, 15, 15, 15, 15],
       [18, 23, 11, 23,  9, 12, 26, 28,  1, 11,  6, 15, 15, 15, 15, 15, 15],
       [18, 12,  1, 32, 23,  2, 25, 22, 12,  1,  6, 15, 15, 15, 15, 15, 15],
       [18, 10,  1, 11, 29, 26, 28, 10,  1, 12, 22, 26,  7,  6, 15, 15, 15],
       [18, 19,  2, 20,  9, 25, 26,  1,  7,  1, 11, 20,  2, 22,  2,  6, 15],
       [18, 19, 23,  7,  1, 11, 22, 13, 23,  2,  6, 15, 15, 15, 15, 15, 15],
       [18,  2,  1,  7,  8,  2,  1, 11, 14,  1,  4, 26, 28,  6, 15, 15, 15],
       [18,  2, 26,  9, 22, 26, 23,  9, 26,  7, 26, 10, 22,  9,  6, 15, 15],
       [18, 23, 28, 20, 12, 25, 28, 26, 19, 26, 22, 23, 12, 22,  7,  6, 15],
       [18, 10,  1,  9, 25, 22,  7,  1, 12, 22, 26,  7,  6, 15, 15, 15, 15],
       [18, 30, 11, 26, 30, 26,  0,  2, 32, 22,  6, 15, 15, 15, 15, 15, 15],
       [18, 19, 28, 26, 10, 22,  7, 23,  7, 12, 11, 20,  6, 15, 15, 15, 15]]
         True, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
         True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
         True,  True,  True, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
        False, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
         True, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
         True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
         True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
```

```
          True, False, False, False, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
          True,  True, False, False, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
          True,  True, False, False, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
          True, False, False, False, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
          True,  True,  True,  True, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
          True, False, False, False, False, False, False],
        [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True
```

Evaluation Routine

The evaluation routine is to take model predictions and compare them with gold labels. The measurements for the quality of predictions is called metric. The routine also includes the mechanism to integrate such metrics to the training loop so that model selection techniques such as early stopping can be employed.

In this section, we will walk you through how to implement a metric, say Character Error Rate, and how to integrate it into the existing Lightning training loop.

Metric: Character Error Rate (CER)

We are going to evaluate our model's predictions using Character Error Rate (CER). This measures the number of edits (insertions, deletions and substitutions) needed to convert our model's prediction to the correct output sequence. Edit distance computation is one of the popular applications of dynamic programming, and is used for measuring character/phone/word error rates in speech recognition.

Complete the following function which takes two sequences (list of characters) and computes the edit distance between them. Another function computes statistics (errors and totals) that will later be used to get CER. These are what we called functional primitives, which themselves can be used to compute edit distance and CER without getting involved in any Lightning contexts. *(12 points)*

```
    def compute_edit_distance(prediction_tokens: List[str], ref
        """Computes edit distance for two sequences using dynam
        This is actually a LeetCode problem :-)

        Parameters
```

```
        ----------
        prediction_tokens : List[str]
            A tokenized predicted sentence.
        reference_tokens : List[str]
            A tokenized reference sentence.

        Returns
        -------
        distance : int
            Edit distance between the predicted sentence and th
        """
        # TODO: Your implementation here
        m, n = len(prediction_tokens), len(reference_tokens)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(m + 1):
            dp[i][0] = i
        for j in range(n + 1):
            dp[0][j] = j

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if prediction_tokens[i-1] == reference_tokens[j
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp
        distance = dp[m][n]
        return distance


    def update_cer(
            preds: Union[str, List[str]],
            targets: Union[str, List[str]]
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        """Updates the CER score with the current set of refere

        Parameters
        ----------
            preds: Transcription(s) to score as a string or lis
            targets: Reference(s) for each speech input as a st

        Returns
        -------
```

```
        Number of edit operations to get from the reference
        Number of character overall references
    """
    if isinstance(preds, str):
        preds = [preds]
    if isinstance(targets, str):
        targets = [targets]
    errors = torch.tensor(0, dtype=torch.float)
    total = torch.tensor(0, dtype=torch.float)

    # TODO: you have to compute edit distance for each pair
    # Then update total errors and total number of referenc
    for pred, target in zip(preds, targets):

        pred_tokens = list(pred.strip())
        target_tokens = list(target.strip())


        distance = compute_edit_distance(pred_tokens, targe

        errors += distance
        total += len(target_tokens)

    return errors, total


def compute_cer(errors: torch.Tensor, total: torch.Tensor)
    """Computes the CER."""
    ...
    if total == 0:
        return torch.tensor(float('inf'))
    return errors / total


def character_error_rate(preds: Union[str, List[str]], targ
    """character error rate is a common metric of the perfo
    value indicates the percentage of characters that were
    the performance of the ASR system with a CER of 0 being

    Parameters
    ----------
        preds: Transcription(s) to score as a string or lis
```

```
        preds: Transcription(s) to score as a string or lis
        targets: Reference(s) for each speech input as a st

    Returns
    -------
        Character error rate score

    Examples
    --------
        preds = ["this is the prediction", "there is an oth
        target = ["this is the reference", "there is anothe
        character_error_rate(preds=preds, targets=targets)
        tensor(0.3415)
    """
    errors, total = update_cer(preds, targets)
    return compute_cer(errors, total)
```

```
grader.check("metric-cer-impl")
```

metric-cer-impl passed!

With the above functional primitives implemented, we will then wrap them to be a `TorchMetric`, which can be integrated into Lightning. *(5 points)*

```
class CharErrorRate(torchmetrics.Metric):
    """ Character Error Rate metric wrapper.

    Parameters
    ----------
        kwargs: Additional keyword arguments, see :ref:`Met

    Returns
    -------
        Character error rate score
    """
    is_differentiable: bool = False
    higher_is_better: bool = False
    full_state_update: bool = False

    error: torch.Tensor
    total: torch.Tensor
```

```
    def __init__(
            self,
            **kwargs: Any,
    ):
        super().__init__(**kwargs)
        self.add_state("errors", torch.tensor(0, dtype=torc
        self.add_state("total", torch.tensor(0, dtype=torch

    def update(self, preds: Union[str, List[str]], targets:
        """Stores references/predictions for computing Char

        Parameters
        ----------
            preds : Union[str, List[str]]
                Transcription(s) to score as a string or li
            targets: Union[str, List[str]]
                Reference(s) for each speech input as a str
        """
        errors, total = update_cer(preds, targets)

        self.errors += errors
        self.total += total

    def compute(self) -> torch.Tensor:
        """Calculates the character error rate.

        Returns
        -------
           Character error rate score
        """
        return compute_cer(self.errors, self.total)
```

```
grader.check("metric-torchmetric-impl")
```

metric-torchmetric-impl passed!

Sequence-to-sequence model with encoder-decoder architecture

In this homework, we require you to implement a sequence-to-sequence model that uses encoder-decoder architecture. You are free to implement the model with either RNN or Transformer. However, you have to bear in mind that the computing resource provided by Colab is limited, and the autograder is configured to run with CPU-only for up to 40 mins with 6GB memory. The autograder runtime is for inference only, so if your training takes relatively longer, it might still be fine with the autograder. To emphasize, you have to test your submission runs with the autograder as in previous homework.

For people that are not familiar with the concept of encoder-decoder model, we refer you to read the Chapter 10 in Speech and Language Processing. You have at least understand what is an encoder-decoder architecture, what is sequence-to-sequence model, how the model is going to be trained, and how decoding works. Some important connections among the described task in the book chapter and this homework, and previous homework:

- Machine Translation task is to translate from one language to the other. In this homework, you can imagine that the task is translating from English to phonemes. One obvious difference is that the target vocabulary is no longer another language, but phonemes.
- In the previous homework, we played with language models, which is very similar to a decoder. For each step, we feed the decoder with what has been generated and ask it to predict a next token. This is the behavior happened at the inference time (using the model to do something). However, we also noticed that the behavior for training is different as we offset the sequence by 1. This is actually exactly the same as the inference, in which given a previous one you have to predict the next one; the only difference is that it happens in parallel as we assume a gold previous one is fed to the model. This is what we called teacher forcing. Please make sure you have understood these concepts before proceeding.
- With the trained model, you have again to use decoding methods learned from the previous homework to decode output sequences.

Note:

- The homework is open to different architecture designs as long as your model takes in one English sentence and spits one phoneme sequence.

Grading

- Simple checks of model function total 7 points.

- There is a baseline on Gradescope using the similar architecture as in the previous homework. Any submission reaches a comparable test score would be given **30 points**. This baseline achieves a value of 0.6834.
- We set a cutoff above the baseline. This cutoff is 0.617. Any submission passes the cutoff would be given another **10 points**. This problem is considered to have **40 points** in total.
- There will also be a leaderboard. The top 15% performers would be given **10 points**, and top 30% performers would be given **5 points**. **These are considered as extra credits.**

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_pa
# This cell provides an example skeleton that you can imple
# You are free to change the skeleton as long as your final
# To use the provided skeleton, it would provide you with a

def shift_tokens_right(input_ids: torch.Tensor, pad_token_i
    """Shifts input ids one token to the right."""
    shifted_input_ids = input_ids.new_zeros(input_ids.shape
    shifted_input_ids[:, 1:] = input_ids[:, :-1].clone()
    shifted_input_ids[:, 0] = decoder_start_token_id

    if pad_token_id is None:
        raise ValueError("self.model.config.pad_token_id ha
    # replace possible -100 values in labels by `pad_token_
    shifted_input_ids.masked_fill_(shifted_input_ids == -10

    return shifted_input_ids


class Encoder(PreTrainedModel):
    def __init__(self,
                 vocab_size: int,
                 embedding_size: int,
                 hidden_size: int,
                 num_layers: int,
                 dropout: float,
                 pad_token_id: int,
                 bos_token_id: int,
                 eos_token_id: int):
        super(Encoder, self).__init__(config=PretrainedConf
            vocab_size=vocab_size,
```

```
            pad_token_id=pad_token_id,
            bos_token_id=bos_token_id,
            eos_token_id=eos_token_id
        ))

        # Configures the embedding - source side
        # We name the embedding as `self.embed_tokens`
        # which is aligned with the following get and set e

        self.embed_tokens = nn.Embedding(vocab_size, embedd

        # Configures the network
        # We assume using RNN here - this aligns with the o
        self.rnn = nn.LSTM(
            input_size=embedding_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            dropout=dropout if num_layers > 1 else 0.0,
            batch_first=True,
            bidirectional=False
        )
        self.dropout = nn.Dropout(dropout)

    def get_input_embeddings(self) -> nn.Module:
        return self.embed_tokens

    def set_input_embeddings(self, value: nn.Module):
        return self.embed_tokens

    def forward(self,
                input_ids: torch.LongTensor = None,
                attention_mask: Optional[torch.Tensor] = No
                **kwargs) -> BaseModelOutput:
        """Encodes `input_ids` using some encoding networks

        Parameters
        ----------
        input_ids : torch.Tensor
            `src_seqs` in the shape of (batch_size, src_seq
        attention_mask : torch.Tensor
            `1` means valid tokens, and `0` means invalid t
            `attention_mask.cpu().sum(-1)` can result in co
```

```
     attention_mask.cpu().sum(-1)` can result in co
    inputs.
kwargs

Returns
-------
`BaseModelOutput` that describes the encoder output
"""
# Shape: (batch_size, sequence_length, embedding_si
# Embeds the `input_ids` to embeddings
embeddings = self.embed_tokens(input_ids)
embeddings = self.dropout(embeddings)

if attention_mask is not None:
    lengths = attention_mask.sum(dim=1).cpu()
else:
    lengths = torch.full((input_ids.size(0),), inpu

packed_embeddings = pack_padded_sequence(
    embeddings,
    lengths,
    batch_first=True,
    enforce_sorted=False
)




# It would output a tuple consists:
#   `outputs` shape: (batch_size, sequence_length,
#   `updated_hidden` shape: (num_layers, batch_size
# Feed input embeddings to the RNN

packed_outputs, (updated_hidden, updated_cell) = se

# Unpacks (back to padded)
# Unpack the `outputs` to:
#   `unpacked_outputs` shape: (batch_size, sequence
#   `output_lengths` shape: (batch_size)
# Using `torch.nn.utils.rnn.pad_packed_sequence`
outputs, output_lengths = pad_packed_sequence(
    packed_outputs,
```

```
            batch_first=True,
            padding_value=0.0
        )

        # You are free to change the outputs
        return BaseModelOutput(
            last_hidden_state=outputs,
            hidden_states=(updated_hidden, updated_cell),
            attentions=None  # Change this if you implement
        )


class Decoder(PreTrainedModel):
    def __init__(self,
                 vocab_size: int,
                 embedding_size: int,
                 hidden_size: int,
                 num_layers: int,
                 dropout: float,
                 pad_token_id: int,
                 bos_token_id: int,
                 eos_token_id: int):
        super(Decoder, self).__init__(config=PretrainedConf
            vocab_size=vocab_size,
            pad_token_id=pad_token_id,
            bos_token_id=bos_token_id,
            eos_token_id=eos_token_id,
            is_decoder=True,
            num_hidden_layers=num_layers,
        ))

        # Similar configurations as in the encoder
        # But this time they are configured to use the targ
        self.embed_tokens = nn.Embedding(vocab_size, embedd

        self.rnn = nn.LSTM(
            input_size=embedding_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            dropout=dropout if num_layers > 1 else 0.0,
            batch_first=True,
            bidirectional=False
```

```
        )
        self.dropout = nn.Dropout(dropout)

    def get_input_embeddings(self) -> nn.Module:
        return self.embed_tokens

    def set_input_embeddings(self, value: nn.Module):
        return self.embed_tokens

    def forward(self,
                input_ids: torch.LongTensor = None,
                encoder_hidden_states: Optional[torch.Float
                decoder_attention_mask: Optional[torch.Tens
                **kwargs) -> BaseModelOutputWithPastAndCros
        """Decodes

        Parameters
        ----------
        input_ids : torch.Tensor, optional
            Input sequence ids on the decoder-side. This is
            fed on the encoder-side. Please make sure you u
            different from encoder-decoder slides.
        encoder_hidden_states : torch.Tensor, optional
            Hidden states from the encoder. This enables th
            the information from the encoder.
        decoder_attention_mask : torch.Tensor, optional
            This is similar to the encoder attention mask.
        kwargs

        Returns
        -------
        `BaseModelOutputWithPastAndCrossAttentions` similar
        but this also leaves the space for implementing att
        """
        # If the decoder mask is not provided, we create on
        if decoder_attention_mask is None:
            decoder_attention_mask = input_ids != self.conf

        # Shape: (batch_size, sequence_length, hidden_size)
        # This is to give a hint on how encoder hidden stat
        # You are free to change it for your implementation
        hidden = encoder_hidden_states
```

```
hidden = encoder_hidden_states

# Shape: (batch_size, sequence_length, embedding_si
# Similar thing for embeddings
if encoder_hidden_states is not None and isinstance
    hidden = encoder_hidden_states[0]  # hidden sta
    cell = encoder_hidden_states[1] if len(encoder_
else:
    hidden = None
    cell = None

embeddings = self.embed_tokens(input_ids)
embeddings = self.dropout(embeddings)

if decoder_attention_mask is not None:
    lengths = decoder_attention_mask.sum(dim=1).cpu
else:
    lengths = torch.full((input_ids.size(0),), inpu

packed_embeddings = pack_padded_sequence(
    embeddings,
    lengths,
    batch_first=True,
    enforce_sorted=False
)


# It would output a tuple consists:
#   `outputs` shape: (batch_size, sequence_length,
#   `updated_hidden` shape: (num_layers, batch_size
# Similar to the encoder, but we can now use the hi
if hidden is not None and cell is not None:
    packed_outputs, (updated_hidden, updated_cell)
else:
    packed_outputs, (updated_hidden, updated_cell)


# Unpacks (back to padded)
# Unpack the `outputs` to:
#   `unpacked_outputs` shape: (batch_size, sequence
#   `output_lengths` shape: (batch_size)
# Using `torch.nn.utils.rnn.pad_packed_sequence`
outputs, output_lengths = pad_packed_sequence(
```

```
        outputs, output_lengths = pad_packed_sequence(
            packed_outputs,
            batch_first=True,
            padding_value=0.0
        )


        # You are free to change the outputs
        return BaseModelOutputWithPastAndCrossAttentions(
            last_hidden_state=outputs,
            hidden_states=(outputs,),
            attentions=None  # Change this if you implement
        )


class EncoderDecoder(PreTrainedModel, GenerationMixin):
    def __init__(self,
                 src_vocab: Vocabulary,
                 tgt_vocab: Vocabulary,
                 embedding_size: int,
                 hidden_size: int,
                 num_layers: int,
                 dropout: float, ):
        self.src_vocab = src_vocab
        self.tgt_vocab = tgt_vocab
        self.src_vocab_size = len(src_vocab)
        self.tgt_vocab_size = len(tgt_vocab)

        # You are free to modify the `__init__` to accommod
        super(EncoderDecoder, self).__init__(config=Pretrai
            decoder_start_token_id=self.tgt_vocab.bos_id(),
            pad_token_id=self.tgt_vocab.pad_id(),
            eos_token_id=self.tgt_vocab.eos_id(),
            is_encoder_decoder=True,  # Tells the generatio
            num_hidden_layers=num_layers,
        ))

        self.encoder = Encoder(
            vocab_size=self.src_vocab_size,
            embedding_size=embedding_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
```

```
            dropout=dropout,
            pad_token_id=self.src_vocab.pad_id(),
            bos_token_id=self.src_vocab.bos_id(),
            eos_token_id=self.src_vocab.eos_id()
        )
        self.decoder = Decoder(
            vocab_size=self.tgt_vocab_size,
            embedding_size=embedding_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            dropout=dropout,
            pad_token_id=self.tgt_vocab.pad_id(),
            bos_token_id=self.tgt_vocab.bos_id(),
            eos_token_id=self.tgt_vocab.eos_id()
        )

        # Classification head for generating tokens for the
        self.lm_head = nn.Linear(hidden_size, self.tgt_voca

    def get_output_embeddings(self) -> nn.Module:
        return self.lm_head

    def set_input_embeddings(self, value: nn.Module):
        self.lm_head = value

    def get_encoder(self):
        return self.encoder

    def get_decoder(self):
        return self.decoder

    # The `generate()` method is commented out
    # It should take `input_ids`, which are batched `src_se
    # corresponding outputs using some decoding algorithms;
    # what we learned from the previous homework.
    # If you decided to base your submission on the current
    # be able to directly use the huggingface generation AP
    #
    # def generate(self, input_ids, attention_mask, **kwarg
    #     raise NotImplementedError

    def forward(
```

```
            self,
            input_ids: torch.LongTensor = None,
            attention_mask: Optional[torch.Tensor] = None,
            decoder_input_ids: Optional[torch.LongTensor] =
            decoder_attention_mask: Optional[torch.Tensor]
            encoder_outputs: Optional[BaseModelOutput] = No
            labels: Optional[torch.LongTensor] = None,
            **kwargs
        ) -> Seq2SeqLMOutput:
            """Runs the encoder-decdoer model.

            Parameters
            ----------
            input_ids : torch.LongTensor, optional
                `src_seqs` to the encoder.
            attention_mask : torch.Tensor, optional
                Encoder side mask for indicating valid tokens (
            decoder_input_ids : torch.LongTensor, optional
                Decoder side input tokens.
            decoder_attention_mask : torch.Tensor, optional
                Decoder side mask for indivating valid tokens (
            encoder_outputs : BaseModelOutput, optional
                The output from encoder. This is to avoid re-ru
                during the decoding as the encoded information
            labels : torch.LongTensor, optional
                This is used to indicate output labels. Recall
                modelling in hw2, where the labels are input to
                In this homework, as we have output different f
                the target sequence as the label.
            kwargs

            Returns
            -------
            `Seq2SeqLMOutput` see corresponding definitions.
            """
            # Automatically creates decoder_input_ids from
            # input_ids if no decoder_input_ids are provided
            if labels is not None:
                if decoder_input_ids is None:
                    decoder_input_ids = shift_tokens_right(
                        labels, self.decoder.config.pad_token_i
                    )
```

```
            ,

        if encoder_outputs is None:
            # Run encoder
            encoder_outputs = self.encoder(
                input_ids=input_ids,
                attention_mask=attention_mask,
            )

        # RUN decoder
        decoder_outputs = self.decoder(
            input_ids=decoder_input_ids,
            encoder_hidden_states=encoder_outputs.hidden_st
            decoder_attention_mask=decoder_attention_mask,
        )

        # Passes the decoder output to LM head to get a dis
        # over the target vocabulary
        # This part is a hint for how to use the decoder ou
        lm_logits = self.lm_head(decoder_outputs.last_hidde

        lm_loss = None
        if labels is not None:
            # Compute the loss function - similar to hw2
            loss_fct = nn.CrossEntropyLoss(ignore_index=sel
            lm_loss = loss_fct(
                lm_logits.view(-1, self.tgt_vocab_size),
                labels.view(-1)
            )

        return Seq2SeqLMOutput(
            loss=lm_loss,
            logits=lm_logits,
            decoder_hidden_states=decoder_outputs.hidden_st
            encoder_last_hidden_state=encoder_outputs.last_
            encoder_hidden_states=encoder_outputs.hidden_st
        )

    def prepare_inputs_for_generation(
            self,
            decoder_input_ids: torch.Tensor,
            # attention_mask: Optional[torch.Tensor]=None,
            encoder_outputs: Optional[BaseModelOutput] = No
```

```
            encoder_outputs: Optional[BaseModelOutput] = No
            **kwargs
    ):
        """Prepares the inputs for the generation mixin fro
        This is the key interface to make sure that you cou
        implemented by huggingface.

        You can imagine that:
        - At the beginning, the generation module calls to
        then be used across all decoding steps.
        - This method prepares input for each decoding step
        decoded token would be added to update `decoder_inp
        - As we implemented `decoder_attention_mask` genera
        need to process attention mask in this method.

        Parameters
        ----------
        decoder_input_ids : torch.Tensor
            Input ids on the decoder side (not the encoder
        encoder_outputs : BaseModelOutput, optional
            The encoded information once the encoding is do
            have the encoder outputs reused across the deco
        kwargs

        Returns
        -------
        Inputs to the `Decoder`. This is literally what bei
        """
        return {
            "input_ids": None,  # encoder_outputs is define
            "encoder_outputs": encoder_outputs,
            "decoder_input_ids": decoder_input_ids,
        }

    def prepare_decoder_input_ids_from_labels(self, labels:
        return shift_tokens_right(labels, self.config.pad_t
```

```
# Please instantiate your model here.
# This will be used to do the sanity check for test cases.

src_vocab = cmudict_src_vocab
tgt_vocab = cmudict_tgt_vocab
```

```
embedding_size = 256
hidden_size = 512
num_layers = 2
dropout = 0.3

test_model = EncoderDecoder(
    src_vocab=src_vocab,
    tgt_vocab=tgt_vocab,
    embedding_size=embedding_size,
    hidden_size=hidden_size,
    num_layers=num_layers,
    dropout=dropout
)
```

```
grader.check("model-generate-checks")
```

model-generate-checks passed!

```
def wrapped_generate(model_to_wrap: nn.Module, **kwargs) ->
    """Wraps the generate method. This function is what wil
    called by the evaluation routine for the leaderboard.

    In this function, you can wrap your `generate()` method
    different generation configurations to be used at the t

    Parameters
    ----------
    model_to_wrap : nn.Module
        Your encoder-decoder model.
    kwargs
        Argument dict that passes all arguments to the gene

    Returns
    -------
        Generated phoneme sequences in the form of torch.Te
    """
    # This is a compatible version with the above `EncoderD
    # If you would like to enable SamplingDecoding here, yo
    #       return model.generate(do_sample=True, **kwargs)
```

```
        return model_to_wrap.generate(**kwargs)
```

```
    class PhonemeGenerationTask(pl.LightningModule):
        """Wraps a PyTorch module as a Lightning Module for the

        def __init__(self,
                     model: nn.Module,
                     src_vocab: Vocabulary,
                     tgt_vocab: Vocabulary,
                     learning_rate: float = 0.001):
            super(PhonemeGenerationTask, self).__init__()
            self.model = model
            self.learning_rate = learning_rate
            self.src_vocab = src_vocab
            self.tgt_vocab = tgt_vocab

            self.cer = CharErrorRate()

        def training_step(self, batch: ParallelBatch) -> torch.
            """Defines the training step.

            Parameters
            ----------
            batch : ParallelBatch
                The batched training instances.

            Returns
            -------
            loss : torch.Tensor
                The loss computed from the seq-to-seq model.
            """
            # Please make necessary modifications to accommodat
            outputs = self.model(
                input_ids=batch.src_seqs,
                attention_mask=batch.src_attention_mask,
                labels=batch.tgt_seqs
            )
            return outputs.loss

        def validation_step(self, batch: ParallelBatch, batch_i
            """Defines the validation step - for this module, w
```

```
            training and validation behaviors. Usually, we woul
            used to select the best performing model checkpoint

            Parameters
            ----------
            batch : ParallelBatch
                The batched training instances.
            batch_idx: int
                The index of the batch.

            Returns
            -------
            loss : torch.Tensor
                The loss computed using CrossEntropyLoss.
            """
            # You are free to modify this function to ensure th
            outputs = self.model.generate(
                input_ids=batch.src_seqs,
                attention_mask=batch.src_attention_mask,
            )
            decoded_outputs = decode_as_str(self.tgt_vocab, out
            curr_cer = self.cer(decoded_outputs, [' '.join(s) f
            self.log('val_cer', self.cer)

        def configure_optimizers(self):
            """Configures optimizers for the training."""
            optimizer = torch.optim.Adam(self.parameters(), lr=
            return optimizer
```

```
    # Please modify the model and trainer configurations to acc
    encoder_decoder_model = EncoderDecoder(
        src_vocab=cmudict_src_vocab,
        tgt_vocab=cmudict_tgt_vocab,
        embedding_size=256,
        hidden_size=512,
        num_layers=2,
        dropout=0.3,
    )
    if is_autograder:  # You have to make sure that you checkpo
        CHECKPOINT_TO_LOAD = "test_model.ckpt" # Please replace
        phoneme_gen_pl_module = PhonemeGenerationTask.load_from
```

```
        checkpoint_path=CHECKPOINT_TO_LOAD,
        src_vocab=cmudict_corpus.src_vocab,
        tgt_vocab=cmudict_corpus.tgt_vocab,
        model=encoder_decoder_model
    )
else:
    # In the student mode, a new model would be trained
    # You are allowed to change training hyperparameters
    # But you are not allowed to create a new task
    phoneme_gen_pl_module = PhonemeGenerationTask(
        model=encoder_decoder_model,
        src_vocab=cmudict_corpus.src_vocab,
        tgt_vocab=cmudict_corpus.tgt_vocab,
        learning_rate=0.001
    )
    phoneme_gen_trainer = pl.Trainer(
        accelerator=accelerator,
        max_epochs=16,
        callbacks=[pl.callbacks.EarlyStopping(monitor='val_
    )
    phoneme_gen_trainer.fit(model=phoneme_gen_pl_module,
                            datamodule=cmudict_corpus)
```

```
INFO:pytorch_lightning.utilities.rank_zero:💡 Tip: For seamless cloud uploads
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used:
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPU
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type           | Params | Mode
-------------------------------------------------
0 | model | EncoderDecoder | 7.4 M  | train
1 | cer   | CharErrorRate  | 0      | train
-------------------------------------------------
7.4 M     Trainable params
0         Non-trainable params
7.4 M     Total params
29.693    Total estimated model params size (MB)
11        Modules in train mode
0         Modules in eval mode
```

Epoch 15: 100%  803/803 [00:14<00:00, 53.86it/s, v_num=0]

```
# You can run this cell to test whether you have get your m
# You are expecting to see a phoneme sequence
```

```
decode_as_str(
    phoneme_gen_pl_module.model.tgt_vocab,
    wrapped_generate(
        model_to_wrap=phoneme_gen_pl_module.model,
        input_ids=encode_as_tensor(phoneme_gen_pl_module.mo
                        'v a n l a n i n g h a m ').to(pho
    )
)
```

```
['V AE0 N L AE1 N IH0 NG HH AE0 M']
```

```
grader.check("model-generate-test")
```

model-generate-test passed!

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs
```