CS 505 Homework 05: Recurrent Neural Networks

Due Monday 11/27 at midnight (1 minute after 11:59 pm) in Gradescope (with a grace period of 6 hours)

You may submit the homework up to 24 hours late (with the same grace period) for a penalty of 10%.

All homeworks will be scored with a maximum of 100 points; point values are given for individual problems, and if parts of problems do not have point values given, they will be counted equally toward the total for that problem.

Note: This homework is a bit different from the first four in this class in that in some parts we are specified **what** you need to do for your solutions, but much less of the **how** you write the details of the code. There are three reasons for this:

- In a graduate level CS class, after four homeworks and two months of lectures, you should be well-equipped to work out the coding issues for yourself, and in general, going forward, this is how you will solve the kinds of problems presented here;
- Suggestions for resources (mostly ML blogs) will be suggested; there are many resources, but these are from bloggers that I trust and have used in the past;
- I am expecting that you will make good use of chatGPT for help with the details of syntax and low-level organization of your code. There is often nothing very stimulating or informative about precisely what is the syntax needed for a particular kind of layer in a network, and rather than poke around on StackOverflow, chatGPT is particularly good at summarizing existing approaches to ML coding tasks.

Submission Instructions

You must complete the homework by editing **this notebook** and submitting the following two files in Gradescope by the due date and time:

- A file `HW05.ipynb` (be sure to select `Kernel -> Restart and Run All` before you submit, to make sure everything works); and
- A file `HW05.pdf` created from the previous.

For best results obtaining a clean PDF file on the Mac, select `File -> Print Review` from the Jupyter window, then choose `File-> Print` in your browser and then `Save as PDF` . Something similar should be possible on a Windows machine -- just make sure it is readable and no cell contents have been cut off. Make it easy to grade!

The date and time of your submission is the last file you submitted, so if your IPYNB file is submitted on time, but your PDF is late, then your submission is late.

Collaborators (5 pts)

Describe briefly but precisely

1. Any persons you discussed this homework with and the nature of the discussion;
2. Any online resources you consulted and what information you got from those resources; and
3. Any AI agents (such as chatGPT or CoPilot) or other applications you used to complete the homework, and the nature of the help you received.

A few brief sentences is all that I am looking for here.

I collaborated with Phillip Tran and Dominic Maglione. I occasionally used chatGPT for minor debugging and checking if the code I wrote is correct by writing tests for it.

In [ ]:
```
import math
```

```python
import numpy as np
from numpy.random import shuffle, seed, choice
from lightning import seed_everything

seed_everything(seed=42)
from tqdm import tqdm
from collections import defaultdict, Counter
import pandas as pd
import re

from pathlib import Path
import matplotlib.pyplot as plt
import matplotlib_inline

# get higher quality plots
matplotlib_inline.backend_inline.set_matplotlib_formats("retina")
from typing import List, Tuple, Any, Dict, Optional, Union
from tqdm import tqdm
import numpy.typing as npt

import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
from torch.utils.data import random_split, Dataset, DataLoader
from torchvision import datasets, transforms
from torch import nn, optim

import torchvision.transforms as T

from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
```

```
Seed set to 42
```

Problem One: Character-Level Generative Model (20 pts)

A basic character-level model has been provided on the class web site in the row for Lecture 14: IPYNB. Your first step is to download this and run it in Colab (or download the data file, which is in the CS 505 Data Directory and also linked on the web site, and run it on your local machine) and understand all its various features. Most of it is straight-forward at this point in the course, but the definition of the model is a bit messy, and you will need to read about LSTM layers in the Pytorch documents to really understand what it is doing and what the hyperparameters mean.

Also take a look at the article "The Unreasonable Effectiveness of Recurrent Neural Networks" linked with lecture 14.

For this problem, you will run this code on a dataset consisting of Java code files, which has been uploaded to the CS 505 Data Directory and also to the class web site: DIR Select some number of these files and concatenate them into one long text file, such that you have approximately 10-20K characters (if you have trouble running out of RAM you can use fewer, but try to get at least 10K).

You will run the character-level model on this dataset. You may either cut and paste code into this notebook, or submit the file with your changes and output along with this notebook to Gradescope.

Your task is to get a character-level model that has not simply memorized the Java text file by overfitting, and does not do much other than spit out random characters (underfitting). You will get the former if you simply run it for many epochs without any changes to the hyperparameters; you will get the latter if you run it only a few epochs.

You should experiment with different hyperparameters, which in the notebook are indicated by

```
        <== something to play with
```

and try to get a model that seems to recognize typical Java syntax such as comments, matching parentheses, expressions, assignments, and formatting, but is not just repeating exact text from the data file. Clearly, the number of epochs plays a crucial role, but I also want you to experiment with the various hyperparameters to try to avoid overfitting. See my lectures on T 10/31 and Th 11/2 (recorded and on my YT channel) for the background to this.

Note that the code you will work from does not use validation and testing sets, nor does it calculate the accuracy, but only tracks the loss. The nature of the data sets for character-level models does not seem to lend itself to accuracy metrics, but you may wish to try this -- I have not found it to be useful, but have simply focussed on the output and "eyeballed" the results to determine how much they have generalized from the data.

Submit your notebook(s) to Gradescope as usual, and also provide a summary of your results in the next cell.

Problem One Solution

In [ ]:
```
DATA_DIR = Path("/data")
MAX_CHARS = 20000
```

In [ ]:
```
java_corpus_path = DATA_DIR.joinpath("java_corpus", "corpus.txt")
with open(java_corpus_path) as f:
    corpus = f.read()[:MAX_CHARS]
```

In [ ]:
```
def create_sequences(
    corpus: str, sequence_len: int = 100
) -> Tuple[List[str], List[str]]:
    input_sequences = []
    target_sequences = []

    for i in range(len(corpus) - sequence_len - 1):
        input_sequences.append(corpus[i : i + sequence_len])
        target_sequences.append(corpus[i + 1 : i + 1 + sequence_len])

    return input_sequences, target_sequences
```

In [ ]:
```
input_sequences, target_sequences = create_sequences(corpus, 100)

print("Input sequence:\n", input_sequences[0])
print("Target sequence:\n", target_sequences[0])
```

```
Input sequence:
 /* File: BSTExperiment.java
 * Authors: Brian Borucki and Wayne Snyder
 * Date: 9/10/13
 * Purpose:
Target sequence:
 * File: BSTExperiment.java
 * Authors: Brian Borucki and Wayne Snyder
 * Date: 9/10/13
 * Purpose: T
```

In [ ]:
```
vocab = sorted(list(set(corpus)))
```

In [ ]:
```
class Tokenizer:
    def __init__(
        self, vocabulary: List[str], dtype: npt.DTypeLike = np.float32
    ) -> None:
        self.vocabulary = vocabulary
        self.embedding_table = dict(zip(self.vocabulary, range(len(self.vocabulary))))
        self.reverse_embedding_table = {
            value: key for key, value in self.embedding_table.items()
        }
        self.embedding_dim = len(self.embedding_table)
        self._identity = np.eye(self.embedding_dim, dtype=dtype)

    def encode(self, sequence: str) -> np.ndarray:
        sequence_ids = [self.embedding_table[s] for s in sequence]
        return self._identity[sequence_ids][np.newaxis, :, :]

    def decode(self, embedding: np.ndarray) -> str:
        sequence_ids = np.argmax(embedding, axis=-1).squeeze()

        text = [self.reverse_embedding_table[s] for s in sequence_ids]

        return "".join(text)
```

In [ ]:
```
tok = Tokenizer(vocabulary=vocab)
```

In [ ]:
```
print(input_sequences[0])
print(tok.decode(tok.encode(sequence=input_sequences[0])))
```

```
/* File: BSTExperiment.java
 * Authors: Brian Borucki and Wayne Snyder
 * Date: 9/10/13
 * Purpose:
/* File: BSTExperiment.java
 * Authors: Brian Borucki and Wayne Snyder
 * Date: 9/10/13
 * Purpose:
```

In [ ]:
```
sequence_len = 100
input_sequences, target_sequences = create_sequences(
    corpus=corpus, sequence_len=sequence_len
)

input_sequences = torch.concatenate(
    [
        torch.tensor(tok.encode(sequence=input_sequence), dtype=torch.float32)
        for input_sequence in input_sequences
    ],
    dim=0,
)

target_sequences = torch.concatenate(
    [
        torch.tensor(tok.encode(sequence=target_sequence), dtype=torch.float32)
        for target_sequence in target_sequences
    ],
    dim=0,
)
```

In [ ]:
```
class CharacterDataset(Dataset):
    def __init__(
        self, input_sequences: torch.Tensor, target_sequences: torch.Tensor
    ) -> None:
        self.input_sequences = input_sequences
        self.target_sequences = target_sequences

    def __len__(self) -> int:
        return len(self.input_sequences)

    def __getitem__(self, index: int) -> Any:
        return (self.input_sequences[index], self.target_sequences[index])
```

In [ ]:
```
# set device
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is being used.")
else:
    device = torch.device("cpu")
    print("CPU is being used.")
```

```
GPU is being used.
```

In [ ]:
```
class Model(nn.Module):
    def __init__(
        self,
        input_size: int,
        hidden_size: int,
        num_layers: int,
        dropout: float,
        sequence_len: int,
        device: torch.device,
    ) -> None:
        super().__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.dropout = dropout
        self.sequence_len = sequence_len
        self.device = device

        self.lstm = nn.LSTM(
            input_size=self.input_size,
            hidden_size=self.hidden_size,
            num_layers=self.num_layers,
            dropout=self.dropout,
            batch_first=True,
        )

        self.fcn = nn.ModuleList(
            [
                nn.Linear(in_features=self.hidden_size, out_features=256),
                nn.GELU(),
                nn.Linear(in_features=256, out_features=self.input_size),
            ]
        )

        self.to(self.device)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        output, (h_n, c_n) = self.lstm(x)
        output = output.contiguous().view(-1, self.hidden_size)
        for layer in self.fcn:
            output = layer(output)
        return output

    def softmax(self, proba: torch.Tensor, temperature: float = 1.0) -> torch.Tensor:
        numerator = torch.exp(proba / temperature)
        denominator = numerator.sum()
        return numerator / denominator

    def predict(self, sequence: str, temperature: float, tokenizer: Tokenizer) -> str:
        assert temperature > 0, "invalid temperature"
        if self.training:
            self.eval()

        input_sequence = torch.from_numpy(tokenizer.encode(sequence)).to(self.device)

        with torch.no_grad():
            output_sequence = self.forward(input_sequence).cpu()

        proba = self.softmax(output_sequence[-1]).numpy()
        return np.random.choice(tokenizer.vocabulary, p=proba)

    def generate(
        self,
        sequence: str,
        tokenizer: Tokenizer,
        max_tokens: int = 512,
        temperature: float = 1.0,
    ) -> str:
        generation_size = max_tokens - len(sequence)

        assert (
            generation_size >= 0
        ), "To generate new tokens you must increase `max tokens`"
        generated_chars = sequence

        for _ in range(generation_size):
            new_char = self.predict(
                generated_chars, temperature=temperature, tokenizer=tokenizer
            )
            generated_chars += new_char

        return generated_chars
```

In [ ]:
```
input_size = len(tok.vocabulary)
model = Model(
    input_size=input_size,
    hidden_size=512,
    num_layers=2,
    dropout=0.0,
    sequence_len=100,
    device=device,
)
```

In [ ]:
```
BS = 64
dataset = CharacterDataset(
    input_sequences=input_sequences, target_sequences=target_sequences
)
dataloader = DataLoader(dataset=dataset, batch_size=BS, shuffle=True, num_workers=6)
```

In [ ]:
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)


def train(
    num_epochs: int,
    save_model_dir: Union[Path, str],
    early_stopping: bool = True,
    patience: int = 3,
) -> Dict[str, List[float]]:
    model.train()

    UPDATE_INTERVAL = 8
    best_epoch_idx = 0
    p = 0

    save_model_dir = Path(save_model_dir)
    if not save_model_dir.exists():
        save_model_dir.mkdir()

    history = dict()
    history["loss"] = []

    for epoch in range(num_epochs):
        with tqdm(total=len(dataloader), unit="batch", desc=f"Epoch {epoch+1}") as pbar:
            N = 0
            running_loss = 0.0

            for idx, (input_sequence, target_sequence) in enumerate(dataloader):
                input_sequence, target_sequence = (
                    input_sequence.to(device),
                    target_sequence.to(device),
                )

                optimizer.zero_grad()

                pred_sequence = model(input_sequence)

                loss = loss_fn(
                    pred_sequence, target_sequence.contiguous().view(-1, input_size)
                )
                running_loss += loss.cpu().item() * len(target_sequence)
                loss.backward()

                nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                optimizer.step()

                N += len(target_sequence)

                if idx % UPDATE_INTERVAL == 0 or idx == len(dataloader) - 1:
                    pbar.update(min(UPDATE_INTERVAL, len(dataloader) - pbar.n))
                    pbar.set_postfix(loss=loss.cpu().item())

            history["loss"].append(running_loss / N)
            loss = history["loss"][-1]
            pbar.set_postfix(loss=loss)

            if early_stopping:
                if p == patience:
                    break

                if loss < history["loss"][best_epoch_idx]:
                    p = 0
                    best_epoch_idx = epoch
                    torch.save(
                        model.state_dict(),
                        save_model_dir.joinpath(
                            f"epoch_{best_epoch_idx}_val_loss_{loss}.pth"
                        ),
                    )
                else:
                    p += 1

    return history
```

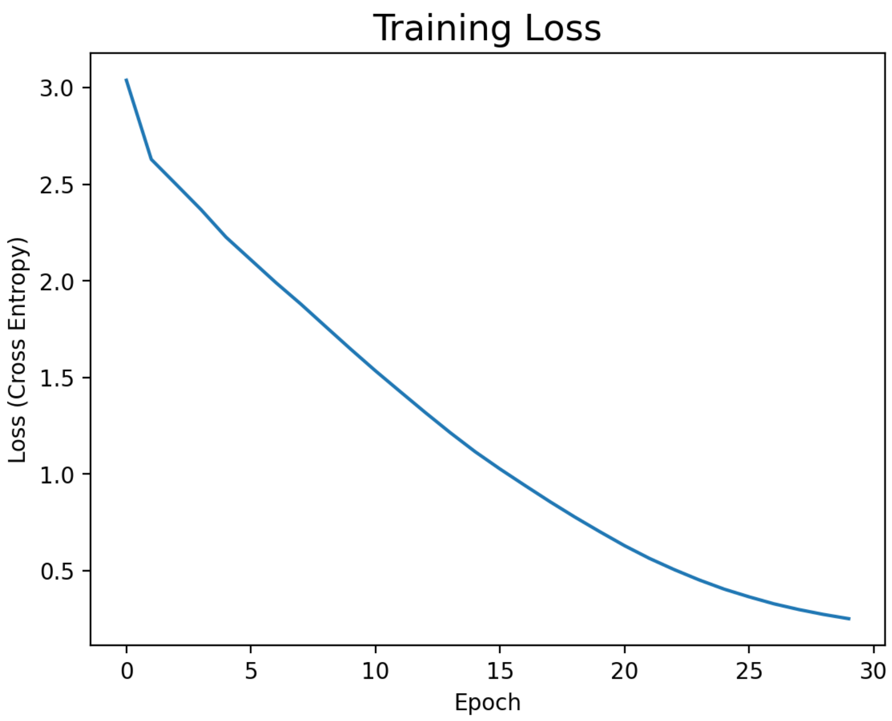
In [ ]:
```
history = train(num_epochs=30,
                early_stopping=True,
                save_model_dir="saved_models",
                patience=3)
```

```
Epoch 1: 100%|██████████| 311/311 [00:09<00:00, 33.16batch/s, loss=3.04]
Epoch 2: 100%|██████████| 311/311 [00:09<00:00, 33.96batch/s, loss=2.63]
Epoch 3: 100%|██████████| 311/311 [00:09<00:00, 34.37batch/s, loss=2.5]
Epoch 4: 100%|██████████| 311/311 [00:09<00:00, 34.15batch/s, loss=2.37]
Epoch 5: 100%|██████████| 311/311 [00:09<00:00, 34.41batch/s, loss=2.23]
Epoch 6: 100%|██████████| 311/311 [00:09<00:00, 34.43batch/s, loss=2.11]
Epoch 7: 100%|██████████| 311/311 [00:09<00:00, 33.99batch/s, loss=1.99]
Epoch 8: 100%|██████████| 311/311 [00:09<00:00, 34.36batch/s, loss=1.88]
Epoch 9: 100%|██████████| 311/311 [00:09<00:00, 34.26batch/s, loss=1.76]
Epoch 10: 100%|██████████| 311/311 [00:09<00:00, 33.20batch/s, loss=1.65]
Epoch 11: 100%|██████████| 311/311 [00:09<00:00, 34.10batch/s, loss=1.53]
Epoch 12: 100%|██████████| 311/311 [00:09<00:00, 34.18batch/s, loss=1.43]
Epoch 13: 100%|██████████| 311/311 [00:09<00:00, 34.23batch/s, loss=1.32]
Epoch 14: 100%|██████████| 311/311 [00:09<00:00, 34.07batch/s, loss=1.21]
Epoch 15: 100%|██████████| 311/311 [00:09<00:00, 34.17batch/s, loss=1.12]
Epoch 16: 100%|██████████| 311/311 [00:09<00:00, 33.98batch/s, loss=1.03]
Epoch 17: 100%|██████████| 311/311 [00:09<00:00, 33.58batch/s, loss=0.941]
Epoch 18: 100%|██████████| 311/311 [00:09<00:00, 33.38batch/s, loss=0.857]
Epoch 19: 100%|██████████| 311/311 [00:09<00:00, 33.30batch/s, loss=0.777]
Epoch 20: 100%|██████████| 311/311 [00:09<00:00, 33.31batch/s, loss=0.702]
Epoch 21: 100%|██████████| 311/311 [00:09<00:00, 33.06batch/s, loss=0.629]
Epoch 22: 100%|██████████| 311/311 [00:09<00:00, 33.19batch/s, loss=0.563]
Epoch 23: 100%|██████████| 311/311 [00:09<00:00, 33.23batch/s, loss=0.505]
Epoch 24: 100%|██████████| 311/311 [00:09<00:00, 33.03batch/s, loss=0.451]
Epoch 25: 100%|██████████| 311/311 [00:09<00:00, 32.98batch/s, loss=0.404]
Epoch 26: 100%|██████████| 311/311 [00:09<00:00, 32.96batch/s, loss=0.364]
Epoch 27: 100%|██████████| 311/311 [00:09<00:00, 32.94batch/s, loss=0.328]
Epoch 28: 100%|██████████| 311/311 [00:09<00:00, 32.93batch/s, loss=0.298]
Epoch 29: 100%|██████████| 311/311 [00:09<00:00, 33.10batch/s, loss=0.273]
Epoch 30: 100%|██████████| 311/311 [00:09<00:00, 32.95batch/s, loss=0.251]
```

In [ ]:
```
fig = plt.figure()
ax = fig.add_subplot(111)

ax.plot(history["loss"])
ax.set_title("Training Loss", fontsize=16)
ax.set_xlabel("Epoch")
ax.set_ylabel("Loss (Cross Entropy)")
```

Out[ ]:
```
Text(0, 0.5, 'Loss (Cross Entropy)')
```



Your analysis

Please describe your experiments and cut and paste various outputs to show how the model performed at various numbers of epochs and with various hyperparameters. What characteristics of Java was it able to learn? What did it not learn? The article "The Unreasonable ..." does a nice job of showing this kind of behavior as the number of epochs increases, and you might look at it before writing your answer here.

Testing different model checkpoints

In [ ]:
```
input_size = len(tok.vocabulary)
model = Model(
    input_size=input_size,
    hidden_size=512,
    num_layers=2,
    dropout=0.0,
    sequence_len=100,
    device=device,
)
```

In [ ]:
```
# start of training
model.load_state_dict(torch.load("saved_models/epoch_1_val_loss_2.6285207074361505.pth"))
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
text = "System."

print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "for (int = 0"
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "if ("
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))
```

```
System.euol(
N
uH
for (int = 0
if (    < al f  tvra
```

In [ ]:
```
# middle of training
model.load_state_dict(
    torch.load("saved_models/epoch_15_val_loss_1.0257065503541785.pth")
)
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
text = "System."

print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "for (int = 0"
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "if ("
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))
```

```
System.<.m.l);

for (int = 0inproy(b
if ((1, ingemt ula))
```

In [ ]:
```
# end of training
model.load_state_dict(
    torch.load("saved_models/epoch_29_val_loss_0.2508196767547441.pth")
)
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
text = "System."

print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "for (int = 0"
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))

text = "if ("
print(model.generate(text, tokenizer=tok, max_tokens=20, temperature=1.0))
```

```
System.out.print(",
for (int = 0; i < a.
if (ins a borraac bo
```

Analysis

I experimented with different model hyperparameters such as `num_layers` (number of stacked LSTMs), played around with hidden unit sizes: 128, 256, and 512, finding 512 to work the best. Experimenting with different number of epochs lead me to find 30 epochs converged to a lower loss without drastically overfitting. In addition, I played around with calculating the gradients by only calculating the gradient of the last character and the gradients of the entire sequence. Calculating the gradients of the entire sequence seemed to yield better performance, however, it is not certain if that result is conclusive. I also experimented with a sequence length of 100 and 128, finding that 100 performs better since the LSTM does not have to worry about long term dependencies. I also played around with the configuration of the fully connected portion of the network, testing two hidden layers and the GELU activation function. These seemed to qualitatively improve the performance, so I kept those additions. From my experimentation, the trained model started to be able to learn the basic syntax of Java like `for` , `if` , and `System.out.println` , but is still very lacking. The model was not able to nail the syntax of the language and seems to have memorized the dataset. It also lacks the ability to write functional code. Furthermore, playing around with the temperature did not yield any more convincing results.

Problem Two: Word-Level Generative Model (40 pts)

In this problem you will write another generative model, as you did in HW 03, but this time you will use an LSTM network, GloVe word embeddings, and beam search.

Before you start, read the following blog post to see the core ideas involved in creating a generative model using word embeddings:

https://machinelearningmastery.com/how-to-develop-a-word-level-neural-language-model-in-keras/

You may also wish to consult with chatGPT about how to develop this kind of model in Pytorch.

The requirements for this problem are as follows (they mostly consist of the extensions proposed at the end of the blog post linked above):

- Develop your code in Pytorch, not Keras
- Use the novel *Persuation* by Jane Austen as your training data (available through the NLTK, you can just grab the sentences using `nltk.corpus.gutenberg.sents('austen-persuasion.txt')`); if you have trouble with RAM you will need to cut down the number of sentences (perhaps by eliminating the longest sentences as well, see next point).
- Develop a sentence-level model by padding sentences to the maximum sentence length in the novel (if this seems extreme, you may wish to delete a small number of the longest sentences to reduce the maximum length). Surround your data sentences with `<s>` and `</s>` and your model should generate one sentence at a time (as you did in HW 03), i.e., it should stop if it generates the `</s>` token.
- Use pretrained GLoVe embeddings with dimension 200, and update them (refine by training further) on the sentences in the novel; if you have trouble with RAM you may use a smaller dimension.
- Experiment with the hyperparameters (sample length, number of layers, uni- or bi-directional, weight_decay, dropout, number of epochs, temperature of the softmax, etc.) as you did in Problem One to find the "sweet spot" where you are generating interesting-looking sentences but not simply repeating sentences from the data. You may want to try adding more linear layers on top to pick the most likely next word.
- Generate sentences using Beam Search, which we describe below.

Your solution should be the code, samples of sentences generated with their score (described below), and your description of the investigation of various hyperparameters, and what strategy ended up seeming to generate the most realistic sentences that were not simply a repeat of sentences in the data.

Beam Search

Beam search was described, and example shown, in Lecture 14. Here is a brief pseudo-code explaination of what you need to do:

1. Develop your code as described above so that it can generate single sentences;
2. Copy enough of your code over from HW 03 so that you can calculate the perplexity of sentences (using the entire novel, or perhaps even a number of Jane Austen's novels as the data source). As an alternative, you may wish to do this separately, store the nested dictionary using Pickle, and load it here.
3. Calculate the probability distribution of sentences in your data source that you used in the previous step, similar to what you did at the end of HW 01.
4. Create a "goodness function" which estimates the quality of a sentence as the perplexity times the probability of its length. This will be applied to all sequences of words, and not just sentences, but as a first approximation this is a way to attempt to make the distribution of sentence lengths similar to that in the novel.
5. Follow the description in slide 7 of Lecture 14 to generate until you have 10 finished sentences. Print these out with their perplexity, probability of their length, and the combined goodness metric.

Problem Two Solution

In [ ]:
```
import nltk
from nltk.corpus import gutenberg
from nltk.tokenize import NLTKWordTokenizer
from tqdm import tqdm
import numpy as np
from typing import List, Tuple, Any
import torch
import heapq
from collections import Counter
```

In [ ]:
```
nltk.download("gutenberg")
nltk.download("punkt")
```

```
[nltk_data] Downloading package gutenberg to
[nltk_data]     /home/alilavaee/nltk_data...
[nltk_data]   Package gutenberg is already up-to-date!
[nltk_data] Downloading package punkt to /home/alilavaee/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[ ]:
```
True
```

In [ ]:
```
sentences = list(gutenberg.sents("austen-persuasion.txt"))
```

In [ ]:
```
len(sentences)
```

Out[ ]:
```
3747
```

In [ ]:
```
SPECIAL_TOKENS = ["[PAD]", "<s>", "</s>"]
EMBEDDING_DIM = 200
BASE_EMBEDDING_SIZE = 400000
PADDING_IDX = 0
```

In [ ]:
```
def preprocess_sentences(sentences: List[List[str]]) -> List[List[str]]:
    processed_sentences = []
    for sentence in sentences:
        s = ["<s>"]
        for word in sentence:
            s.append(word)
        s.append("</s>")
        processed_sentences.append(s)
    return processed_sentences
```

In [ ]:
```
processed_sentences = preprocess_sentences(sentences=sentences)
words = [w for s in processed_sentences for w in s]
```

In [ ]:
```
def create_sequences(
    words: List[str], sequence_len: int = 50
) -> Tuple[List[List[str]], List[List[str]]]:
    input_sequences = []
    target_sequences = []

    for i in range(len(words) - sequence_len - 1):
        input_sequences.append(words[i : i + sequence_len])
        target_sequences.append(words[i + 1 : i + 1 + sequence_len])

    return input_sequences, target_sequences
```

In [ ]:
```
vocab = []
embeddings = []
unique_words = set(words)

for t in SPECIAL_TOKENS:
    vocab.append(t)
    embeddings.append(
        torch.normal(mean=0, std=1, size=(EMBEDDING_DIM,), dtype=torch.float32)
    )

with open("/data/glove_6B/glove.6B.200d.txt") as f:
    for line in tqdm(f.readlines(), total=BASE_EMBEDDING_SIZE):
        content = line.split()
        word, vector = content[0], content[1:]
        if word in unique_words:
            vocab.append(word)
            vector = np.array(vector, dtype=np.float32)
            embeddings.append(torch.from_numpy(vector))
```

```
100%|██████████| 400000/400000 [00:02<00:00, 142029.01it/s]
```

In [ ]:
```
for word in tqdm(unique_words):
    if word not in set(vocab):
        vocab.append(word)
        embeddings.append(
            torch.normal(mean=0, std=1, size=(EMBEDDING_DIM,), dtype=torch.float32)
        )
```

```
100%|██████████| 6133/6133 [00:01<00:00, 5454.19it/s]
```

In [ ]:
```
len(vocab)
```

Out[ ]:
```
6134
```

In [ ]:
```
embeddings = torch.stack(embeddings, dim=0)
```

In [ ]:
```
embeddings.size()
```

Out[ ]:
```
torch.Size([6134, 200])
```

In [ ]:
```
sentence_lengths = [len(s) for s in sentences]
total_sentence_lengths = sum(sentence_lengths)
sentence_lengths = Counter(sentence_lengths)
sentence_length_probas = {
    length: freq / total_sentence_lengths for length, freq in sentence_lengths.items()
}
```

In [ ]:
```
class Tokenizer:
    def __init__(
        self,
        vocabulary: List[str],
        special_tokens: List[str],
        dtype: torch.dtype = torch.int64,
    ) -> None:
        self.vocabulary = vocabulary
        self.special_tokens = special_tokens
        self.embedding_table = dict(zip(self.vocabulary, range(len(self.vocabulary))))
        self.reverse_embedding_table = {
            value: key for key, value in self.embedding_table.items()
        }
        self.embedding_dim = len(self.embedding_table)
        self.dtype = dtype

    def encode(self, sequence: List[str]) -> torch.Tensor:
        sequence_ids = torch.tensor(
            [[self.embedding_table[s] for s in sequence]], dtype=self.dtype
        )
        return sequence_ids

    def decode(
        self, sequence_ids: torch.Tensor, skip_special_tokens: bool = True
    ) -> List[str]:
        tokens = [self.reverse_embedding_table[s] for s in sequence_ids.numpy()]
        if skip_special_tokens:
            tokens = list(filter(lambda t: t not in set(self.special_tokens), tokens))
        return tokens
```

In [ ]:
```python
class GutenBergDataset(Dataset):
    def __init__(
        self,
        vocab_size: int,
        input_sequences: torch.Tensor,
        target_sequences: torch.Tensor,
        dtype: torch.dtype = torch.float32,
    ) -> None:
        self.vocab_size = vocab_size
        self.input_sequences = input_sequences
        self.target_sequences = target_sequences
        self.dtype = dtype
        self._one_hot_matrix = torch.eye(self.vocab_size, dtype=self.dtype)

    def __len__(self) -> int:
        return len(self.input_sequences)

    def __getitem__(self, index: int) -> Tuple[torch.Tensor, torch.Tensor]:
        input_sequence = self.input_sequences[index]
        target_sequence = self.target_sequences[index]
        one_hot_target = self._one_hot_matrix[target_sequence]
        return input_sequence, one_hot_target
```

In [ ]:
```python
class Model(nn.Module):
    def __init__(
        self,
        embed_dim: int,
        embeddings: torch.Tensor,
        hidden_size: int,
        num_layers: int,
        dropout: float,
        device: torch.device,
        freeze_embedding_weights: bool = True,
        padding_idx: int = 0,
    ) -> None:
        super().__init__()

        self.embed_dim = embed_dim
        self.num_embeddings = embeddings.size(0)
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.dropout = dropout
        self.device = device
        self.padding_idx = padding_idx
        self.word_tokenizer = NLTKWordTokenizer()

        self.embed = nn.Embedding.from_pretrained(
            embeddings=embeddings,
            padding_idx=self.padding_idx,
            freeze=freeze_embedding_weights,
        )

        self.lstm = nn.LSTM(
            input_size=self.embed_dim,
            hidden_size=self.hidden_size,
            num_layers=self.num_layers,
        )

        self.fcn = nn.ModuleList(
            [
                nn.Linear(in_features=self.hidden_size, out_features=256),
                nn.BatchNorm1d(num_features=256),
                nn.GELU(),
                nn.Linear(in_features=256, out_features=self.num_embeddings),
            ]
        )

        self.to(self.device)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        sequence_len = x.size(1)
        output = self.embed(x)
        output, (h_n, c_n) = self.lstm(output)
        output = output.contiguous().view(-1, self.hidden_size)
        for layer in self.fcn:
            output = layer(output)
        output = output.view(-1, sequence_len, self.num_embeddings)
        return output

    def softmax(self, proba: torch.Tensor, temperature: float = 1.0) -> torch.Tensor:
        numerator = torch.exp(proba / temperature)
        denominator = numerator.sum()
        return numerator / denominator

    def calculate_perplexity(self, sum_log_proba: float, N: int) -> float:
        perplexity = math.exp((-1 / N) * sum_log_proba)
        return perplexity

    def calculate_goodness(
        self, perplexity: float, sentence_length_proba: float
    ) -> float:
        return math.exp(math.log(perplexity) + math.log(sentence_length_proba))

    def select_idx(
        self,
        proba: torch.Tensor,
        proba_indices: torch.Tensor,
        num_samples: int,
        strategy: str,
    ) -> torch.Tensor:
        assert strategy in ["max", "sample"], "strategy must be `max` or `sample`"
        if strategy == "max":
            idx = proba_indices[:num_samples]
        else:
            idx = torch.multinomial(
                input=proba, num_samples=num_samples, replacement=True
            )
        return idx

    def greedy(
        self,
        sequence: List[str],
        temperature: float,
        tokenizer: Tokenizer,
        max_generation_size: int,
        sentence_length_probas: Dict[int, float],
        top_k: Optional[int],
        top_p: Optional[float],
    ) -> List[Dict[str, Any]]:
        seq = []
        seq.extend(sequence)
        sum_log_proba = 0
        for _ in range(max_generation_size):
            proba, proba_indices = self.predict(
                sequence=seq,
                tokenizer=tokenizer,
                temperature=temperature,
                top_k=top_k,
                top_p=top_p,
            )
            idx = self.select_idx(
                proba=proba,
                proba_indices=proba_indices,
                num_samples=1,
                strategy="sample" if top_k or top_p else "max",
            )
            next_seq = tokenizer.decode(idx, skip_special_tokens=True)

            if next_seq == "</s>":
                break
            seq.extend(next_seq)
            sum_log_proba += math.log(proba.index_select(dim=0, index=idx))

        seq_len = len(seq) - 1
        perplexity = self.calculate_perplexity(sum_log_proba=sum_log_proba, N=seq_len)
        length_proba = sentence_length_probas[seq_len]
        goodness = self.calculate_goodness(
            perplexity=perplexity, sentence_length_proba=length_proba
        )

        return [
            {
                "generation": seq[1:],
                "perplexity": perplexity,
                "sentence_length_proba": length_proba,
                "goodness": goodness,
            }
        ]

    def beam_search(
        self,
        sequence: List[str],
        temperature: float,
        tokenizer: Tokenizer,
        max_generation_size: int,
        sentence_length_probas: Dict[int, float],
        top_k: Optional[int],
        top_p: Optional[float],
        max_generate_sentences: int,
        beam_width: int,
    ) -> List[Dict[str, Any]]:
        priority_queue = [(0.0, 0.0, sequence)]
        heapq.heapify(priority_queue)
        generated_sentences = []
        while (
            len(generated_sentences) < max_generate_sentences
            and len(priority_queue) > 0
        ):
            perplexity, cum_log_proba, seq = heapq.heappop(priority_queue)
            proba, proba_indices = self.predict(
                sequence=seq,
                tokenizer=tokenizer,
                temperature=temperature,
                top_k=top_k,
                top_p=top_p,
            )
            candidates = self.select_idx(
                proba=proba,
                proba_indices=proba_indices,
                num_samples=beam_width,
                strategy="sample" if top_k or top_p else "max",
            )
            next_sequences = tokenizer.decode(candidates, skip_special_tokens=False)
            # print(next_sequences)
            for idx, next_seq in zip(candidates, next_sequences):
                if next_seq != "</s>":
                    sum_log_proba = cum_log_proba + math.log(
                        proba.index_select(dim=0, index=idx)
                    )
                    new_seq = seq + [next_seq]
                else:
                    sum_log_proba = cum_log_proba
                    new_seq = seq

                # len(new_seq) - 1 because we are excluding <s> special token in calculation
                seq_len = len(new_seq) - 1

                perplexity = self.calculate_perplexity(
                    sum_log_proba=sum_log_proba, N=seq_len
                )

                if next_seq == "</s>" or seq_len == max_generation_size:
                    length_proba = sentence_length_probas[seq_len]
                    goodness = self.calculate_goodness(
                        perplexity=perplexity, sentence_length_proba=length_proba
                    )
                    generated_sentences.append(
                        {
                            "generation": seq[1:],
                            "perplexity": perplexity,
                            "sentence_length_proba": length_proba,
                            "goodness": goodness,
                        }
                    )
                    continue

                heapq.heappush(priority_queue, (perplexity, sum_log_proba, new_seq))
        return generated_sentences

    def predict(
        self,
        sequence: List[str],
        temperature: float,
        tokenizer: Tokenizer,
        top_k: Optional[int],
        top_p: Optional[float],
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        # checks
        assert temperature > 0, "invalid temperature"

        if self.training:
            self.eval()

        input_sequence = tokenizer.encode(sequence).to(self.device)

        with torch.no_grad():
            output_sequence = self.forward(input_sequence).cpu()

        proba = self.softmax(output_sequence[0, -1, :])
        proba_indices = torch.argsort(proba, descending=True).numpy()

        cutoff_idx = len(proba_indices) - 1

        if top_k:
            cutoff_idx = min(cutoff_idx, top_k)

        if top_p:
            cum_proba = 0.0
            idx = 0
            while cum_proba < top_p and idx <= cutoff_idx:
                cum_proba += float(proba[proba_indices[idx]].item())
                idx += 1

            cutoff_idx = idx

        if cutoff_idx < len(proba_indices) - 1:
            proba[proba_indices[cutoff_idx:]] = 0.0
            # re-normalize probabilities to sum to 1
            proba /= proba.sum()

        proba_indices = torch.from_numpy(proba_indices)

        return proba, proba_indices

    def generate(
        self,
        text: str,
        tokenizer: Tokenizer,
        sentence_length_probas: Dict[int, float],
        max_tokens: int = 512,
        temperature: float = 1.0,
        decoding_method: str = "greedy",
        top_k: Optional[int] = None,
        top_p: Optional[float] = None,
        max_generation_sentences: int = 10,
        beam_width: int = 2,
    ) -> List[Dict[str, Any]]:
        assert decoding_method in [
            "greedy",
            "beam_search",
        ], "choose either greedy or beam_search decoding method"

        sequence = ["<s>"]
        sequence.extend(self.word_tokenizer.tokenize(text))

        max_generation_size = max_tokens - len(sequence)

        assert (
            max_generation_size >= 0
        ), "To generate new tokens you must increase `max tokens`"

        if decoding_method == "greedy":
            result = self.greedy(
                sequence=sequence,
                temperature=temperature,
                tokenizer=tokenizer,
                max_generation_size=max_generation_size,
                sentence_length_probas=sentence_length_probas,
                top_p=top_p,
                top_k=top_k,
            )
        else:
            result = self.beam_search(
                sequence=sequence,
                temperature=temperature,
                tokenizer=tokenizer,
                max_generation_size=max_generation_size,
                sentence_length_probas=sentence_length_probas,
                top_p=top_p,
                top_k=top_k,
                max_generate_sentences=max_generation_sentences,
                beam_width=beam_width,
            )
        return result
```

In [ ]:
```
# set device
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is being used.")
else:
    device = torch.device("cpu")
    print("CPU is being used.")
```

```
GPU is being used.
```

In [ ]:
```
model = Model(
    embed_dim=EMBEDDING_DIM,
    embeddings=embeddings,
    hidden_size=512,
    num_layers=2,
    dropout=0.5,
    device=device,
)
```

In [ ]:
```
BS = 256
tok = Tokenizer(vocabulary=vocab, special_tokens=SPECIAL_TOKENS)

sequence_len = 100
input_sequences, target_sequences = create_sequences(
    words=words, sequence_len=sequence_len
)
input_sequences = torch.concatenate([tok.encode(s) for s in input_sequences], dim=0)
target_sequences = torch.concatenate([tok.encode(s) for s in target_sequences], dim=0)
```

In [ ]:
```
dataset = GutenBergDataset(
    vocab_size=len(vocab),
    input_sequences=input_sequences,
    target_sequences=target_sequences,
)
dataloader = DataLoader(dataset=dataset, batch_size=BS, shuffle=True, num_workers=6)
```

In [ ]:
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


def train(
    num_epochs: int,
    save_model_dir: Union[Path, str],
    history: Dict[str, Any],
    early_stopping: bool = True,
    patience: int = 3,
) -> Dict[str, List[float]]:
    model.train()

    UPDATE_INTERVAL = 8
    best_epoch_idx = 0
    p = 0

    save_model_dir = Path(save_model_dir)
    if not save_model_dir.exists():
        save_model_dir.mkdir()

    if "loss" not in history:
        history["loss"] = []

    for epoch in range(num_epochs):
        with tqdm(total=len(dataloader), unit="batch", desc=f"Epoch {epoch+1}") as pbar:
            N = 0
            running_loss = 0.0

            for idx, (input_sequence, target_sequence) in enumerate(dataloader):
                input_sequence, target_sequence = (
                    input_sequence.cuda(),
                    target_sequence.cuda(),
                )

                optimizer.zero_grad()

                pred_sequence = model(input_sequence)

                loss = loss_fn(
                    pred_sequence.contiguous().view(-1, embeddings.size(0)),
                    target_sequence.contiguous().view(-1, embeddings.size(0)),
                )

                running_loss += loss.cpu().item() * len(target_sequence)
                loss.backward()

                nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                optimizer.step()

                N += len(target_sequence)

                if idx % UPDATE_INTERVAL == 0 or idx == len(dataloader) - 1:
                    pbar.update(min(UPDATE_INTERVAL, len(dataloader) - pbar.n))
                    pbar.set_postfix(loss=loss.cpu().item())

            history["loss"].append(running_loss / N)
            loss = history["loss"][-1]
            pbar.set_postfix(loss=loss)

            if early_stopping:
                if p == patience:
                    break

                if loss < history["loss"][best_epoch_idx]:
                    p = 0
                    best_epoch_idx = epoch
                    torch.save(
                        model.state_dict(),
                        save_model_dir.joinpath(
                            f"epoch_{best_epoch_idx}_val_loss_{loss}.pth"
                        ),
                    )
                else:
                    p += 1

    return history
```

In [ ]:
```
history = dict()
```

In [ ]:
```
history = train(num_epochs=10,
                history=history,
                early_stopping=True,
                save_model_dir="saved_models_2_100_words",
                patience=3)
```

```
Epoch 1: 100%|██████████| 413/413 [01:39<00:00,  4.14batch/s, loss=4.24]
Epoch 2: 100%|██████████| 413/413 [01:40<00:00,  4.13batch/s, loss=3.46]
Epoch 3: 100%|██████████| 413/413 [01:39<00:00,  4.14batch/s, loss=3.36]
Epoch 4: 100%|██████████| 413/413 [01:39<00:00,  4.15batch/s, loss=3.33]
Epoch 5: 100%|██████████| 413/413 [01:39<00:00,  4.15batch/s, loss=3.32]
Epoch 6: 100%|██████████| 413/413 [01:39<00:00,  4.15batch/s, loss=3.32]
Epoch 7: 100%|██████████| 413/413 [01:39<00:00,  4.14batch/s, loss=3.31]
Epoch 8: 100%|██████████| 413/413 [01:39<00:00,  4.13batch/s, loss=3.31]
Epoch 9: 100%|██████████| 413/413 [01:40<00:00,  4.11batch/s, loss=3.31]
Epoch 10: 100%|██████████| 413/413 [01:40<00:00,  4.13batch/s, loss=3.3]
```

In [ ]:
```
# unfreeze embedding weight
model.embed.weight.requires_grad = True
```

In [ ]:
```
history = train(num_epochs=10,
                history=history,
                early_stopping=True,
                save_model_dir="saved_models_2_100_words",
                patience=3)
```

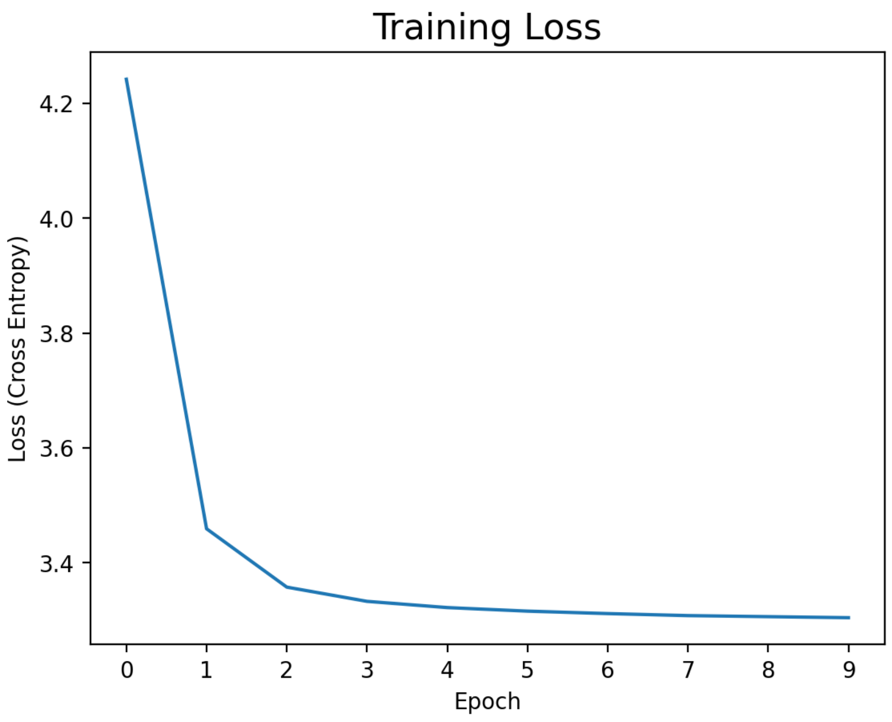
```
Epoch 1: 100%|██████████| 413/413 [00:53<00:00,  7.79batch/s, loss=3.32]
Epoch 2: 100%|██████████| 413/413 [00:52<00:00,  7.83batch/s, loss=3.31]
Epoch 3: 100%|██████████| 413/413 [00:52<00:00,  7.81batch/s, loss=3.31]
Epoch 4: 100%|██████████| 413/413 [00:52<00:00,  7.79batch/s, loss=3.31]
Epoch 5: 100%|██████████| 413/413 [00:53<00:00,  7.79batch/s, loss=3.31]
Epoch 6: 100%|██████████| 413/413 [00:52<00:00,  7.80batch/s, loss=3.31]
Epoch 7: 100%|██████████| 413/413 [00:53<00:00,  7.79batch/s, loss=3.31]
Epoch 8: 100%|██████████| 413/413 [00:52<00:00,  7.81batch/s, loss=3.3]
Epoch 9: 100%|██████████| 413/413 [00:53<00:00,  7.77batch/s, loss=3.3]
Epoch 10: 100%|██████████| 413/413 [00:53<00:00,  7.79batch/s, loss=3.3]
```

```
In [ ]: from matplotlib.ticker import MaxNLocator

        fig = plt.figure()
        ax = fig.add_subplot(111)

        ax.plot(history["loss"])
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.set_title("Training Loss", fontsize=16)
        ax.set_xlabel("Epoch")
        ax.set_ylabel("Loss (Cross Entropy)")
```

```
Out[ ]: Text(0, 0.5, 'Loss (Cross Entropy)')
```



```
In [ ]: model = Model(
            embed_dim=EMBEDDING_DIM,
            embeddings=embeddings,
            hidden_size=512,
            num_layers=2,
            dropout=0.5,
            device=device,
        )

        model.load_state_dict(
            torch.load("saved_models_2_100_words/epoch_9_val_loss_3.3042992417253254.pth")
        )
```

```
Out[ ]: <All keys matched successfully>
```

```
In [ ]: model.generate(
            text="Sir Walter Elliot, of Kellynch Hall was",
            tokenizer=tok,
            sentence_length_probas=sentence_length_probas,
            max_tokens=32,
            temperature=1.6,
            decoding_method="greedy",
            top_k=None,
            top_p=0.9,
            max_generation_sentences=10,
            beam_width=2,
        )
```

```
Out[ ]: [{'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'evidently',
           'delighted',
           ',',
           'was',
           ',',
           'and',
           'you',
           'do',
           'without',
           'her',
           'own',
           'chamber',
           'to',
           'Bath',
           'the',
           'possibility',
           'of',
           'her',
           'family',
           '?'],
          'perplexity': 13.564376134099195,
          'sentence_length_proba': 0.00057042741311116816,
          'goodness': 0.007737491988648034}]
```

```
In [ ]: model.generate(
            text="Sir Walter Elliot, of Kellynch Hall was",
            tokenizer=tok,
            sentence_length_probas=sentence_length_probas,
            max_tokens=64,
            temperature=1.3,
            decoding_method="beam_search",
            top_k=None,
            top_p=0.9,
            max_generation_sentences=10,
            beam_width=2,
        )
```

```
Out[ ]: [{'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'import',
           '.'],
          'perplexity': 2.535503149069148,
          'sentence_length_proba': 0.0013343926985291122,
          'goodness': 0.003383356889215443},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'import',
           '.'],
          'perplexity': 2.535503149069148,
          'sentence_length_proba': 0.0013343926985291122,
          'goodness': 0.003383356889215443},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'import',
           '.'],
          'perplexity': 2.535503149069148,
          'sentence_length_proba': 0.0013343926985291122,
          'goodness': 0.003383356889215443},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'import',
           '.'],
          'perplexity': 2.535503149069148,
          'sentence_length_proba': 0.0013343926985291122,
          'goodness': 0.003383356889215443},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'a',
           'gnawing',
           'solicitude',
           'in',
           'the',
           'evening',
           '.'],
          'perplexity': 3.7593476204285117,
          'sentence_length_proba': 0.0011204824186122316,
          'goodness': 0.004212282914141875},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'a',
           'gnawing',
           'solicitude',
           'in',
           'the',
           'evening',
           '.'],
          'perplexity': 3.7593476204285117,
          'sentence_length_proba': 0.0011204824186122316,
          'goodness': 0.004212282914141875},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'a',
           'gnawing',
           'solicitude',
           'for',
           'them',
           '!'],
          'perplexity': 3.8216472812929907,
          'sentence_length_proba': 0.0010899238071955344,
          'goodness': 0.004165304354585322},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'a',
           'gnawing',
           'solicitude',
           'for',
           'them',
           '!'],
          'perplexity': 3.8216472812929907,
          'sentence_length_proba': 0.0010899238071955344,
          'goodness': 0.004165304354585322},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'to',
           'Mr',
           'Elliot',
           "'",
           's',
           'acquaintance',
           '.'],
          'perplexity': 4.1043864407100985,
          'sentence_length_proba': 0.000865827323473088,
          'goodness': 0.00355368992645926},
         {'generation': ['Sir',
           'Walter',
           'Elliot',
           ',',
           'of',
           'Kellynch',
           'Hall',
           'was',
           'too',
           'painful',
           'to',
           'Mr',
           'Elliot',
           "'",
           's',
           'acquaintance',
           '.'],
          'perplexity': 4.1043864407100985,
          'sentence_length_proba': 0.000865827323473088,
          'goodness': 0.00355368992645926}]
```

Analysis

Describe what experiments you did with various alternatives as described above, and cut and paste examples illustrating your results.

I tested different model hyperparameters such as `num_layers` (number of stacked LSTMs), played around with hidden unit sizes: 128, 256, and 512, finding 512 to work the best. Experimenting with different number of epochs lead me to find 20 epochs converged to a lower loss without drastically overfitting. In addition, I played around with calculating the gradients by only calculating the gradient of the last word and the gradients of the entire sequence of words. Calculating the gradients of the entire sequence seemed to yield more stable gradients during training, however, the performance of both were lackluster. I also played around with the GELU activation function and batch normalization in the fully connected layers for regularization. Another thing I tried was initially freezing the embedding weights and then in later stages of training unfreezing and updating the embedding weights. This slightly improves performance of the model. Additionally, I experimented with a sequence length of 50 and 100, finding that 100 performs slightly better (I hypothesize it provides a reasonable sequence length to learn). In the generation stage I played around with different parameters and found `max_tokens=64`, `temperature=1.3`, `top_k=None`, `top_p=0.9`, and `beam_width=2` for beam search to get reasonable performance. Interestingly, `top_p` was one of the more important parameters in generation along with `max_tokens`. They slightly improved the models performance, but I did clearly recognize the repetition problem in language models. Maybe in the future adding some sort of repetition penalty would help. In summary, the model seems to be developing some ability, however, it is nowhere near fluent or interpretable.

Problem Three: Part-of-Speech Tagging (40 pts)

In this problem, we will experiment with three different approaches to the POS tagging problem, using the Brown Corpus as our data set.

Before starting this problem, please review Lecture 13 and download the file Viterbi_Algorithm.ipynb from the class web site.

There are four parts to this problem:

- Part A: You will establish a baseline accuracy for the task.
- Part B: Using the implementation of the Viterbi algorithm for Hidden Markov Models you downloaded, you will determine how much better than the baseline you can do with this very standard method.
- Part C: You will repeat the exercise of Part B, but using an LSTM implementation, exploring several options for the implementation of the LSTM layer.
- Part D: You will evaluate your results, comparing the various methods in the context of the baseline method from Part A.
- Optional: You may wish to try the same task with a transformer such as Bert.

Recall that the Brown Corpus has a list of all sentences tagged with parts of speech. The tags are a bit odd, and not generally used any more, so we will use a much simpler set of tags the `universal_tagset` .

If you run the following cells, you will see that there are 57,340 sentences, tagged with 12 different tags.

In [ ]:
```python
import numpy as np
from numba import njit
from tqdm import trange
import nltk
import math
from numpy.random import shuffle, seed, choice
from tqdm import tqdm
from sklearn.model_selection import train_test_split

# The first time you will need to download the corpus:

from nltk.corpus import brown

nltk.download("brown")
nltk.download("universal_tagset")

tagged_sentences = brown.tagged_sents(tagset="universal")

print(
    f"There are {len(tagged_sentences)} sentences tagged with universal POS tags in the Brown Corpu
)
print("\nHere is the first sentence with universal tags:", tagged_sentences[0])
```

```
[nltk_data] Downloading package brown to /home/alilavaee/nltk_data...
[nltk_data]   Package brown is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     /home/alilavaee/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
There are 57340 sentences tagged with universal POS tags in the Brown Corpus.

Here is the first sentence with universal tags: [('The', 'DET'), ('Fulton', 'NOUN'), ('County', 'NO
UN'), ('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ('Friday', 'NOUN'), ('an', 'DET'), ('in
vestigation', 'NOUN'), ('of', 'ADP'), ("Atlanta's", 'NOUN'), ('recent', 'ADJ'), ('primary', 'NOUN
'), ('election', 'NOUN'), ('produced', 'VERB'), ('``', '.'), ('no', 'DET'), ('evidence', 'NOUN'),
("''", '.'), ('that', 'ADP'), ('any', 'DET'), ('irregularities', 'NOUN'), ('took', 'VERB'), ('place
', 'NOUN'), ('.', '.')]
```

In [ ]:
```python
# Uncomment to see the complete list of tags.

all_tagged_words = np.concatenate(tagged_sentences)
all_tags = sorted(set([pos for (w, pos) in all_tagged_words]))
print(f"There are {len(all_tags)} universal tags in the Brown Corpus.")
print(all_tags)
print()
```

```
There are 12 universal tags in the Brown Corpus.
['.', 'ADJ', 'ADP', 'ADV', 'CONJ', 'DET', 'NOUN', 'NUM', 'PRON', 'PRT', 'VERB', 'X']
```

Part A

In this part, you will establish a baseline for the task, using the naive method suggested on slide 35 of Lecture 13:

- Tag every word with its most frequent POS tag (for example, if 'recent' is most frequently tagged as 'ADJ', then assume that every time 'recent' appears in a sentence, it should be tagged with 'ADJ');
- If a word has two or more most frequent tags, choose the one that appears first in the list of sorted tags above.

Note that there will not be any "unknown words."

Use this method to determine your baseline accuracy (it may not be 92% as reported on slide 35!):

- Build a dictionary mapping every word to its most frequent tag;
- Go through the entire tagged corpus, and report the accuracy (percentage of correct tags) of this baseline method.

Do not tokenize or lower-case the words. Use the words and tags exactly as they are in the tagged sentences.

In [ ]:
```
tagged_word_frequencies = dict()

for w, pos in tqdm(all_tagged_words):
    if w not in tagged_word_frequencies:
        tagged_word_frequencies[w] = defaultdict(int)
    tagged_word_frequencies[w][pos] += 1
```

```
  6%|▌         | 72918/1161192 [00:00<00:01, 729151.14it/s]100%|██████████| 1161192/1161192 [00:01
<00:00, 744601.50it/s]
```

In [ ]:
```
tagged_word_predictions = defaultdict(int)
for w in tqdm(tagged_word_frequencies.keys()):
    tagged_word_predictions[w] = max(
        tagged_word_frequencies[w], key=lambda pos: tagged_word_frequencies[w][pos]
    )
```

```
100%|██████████| 56057/56057 [00:00<00:00, 1655297.41it/s]
```

In [ ]:
```
baseline_accuracy = sum(
    [int(tagged_word_predictions[w] == y_test) for w, y_test in all_tagged_words]
) / len(all_tagged_words)
```

In [ ]:
```
baseline_accuracy
```

Out[ ]:
```
0.9570777270253326
```

Part B:

Now, review the `Viterbi.ipynb` notebook and read through Section 8.4 in Jurafsky & Martin to understand the basic approach that is used in the "Janet will back the bill" example. In detail:

- Cut and paste the code from the Viterby notebook below and run your experiments in this notebook.
- You need to calculate from the Brown Corpus tagged sentences the probabilities for the various matrices used as input to the method:
    - `start_p` : This is the probability that a sentence starts with a given POS (in Figure 8.12 in J & M, this is given as the first line, in the row for <s> ; simply collect the statistics for the first word in each sentence; it will be of size 1 x 12.
    - `trans_p` : This is the matrix of probabilities that one POS follows another in a sentence; build a 12 x 12 matrix of frequencies for whether the column POS follows the row POS in a sentence and then normalize each row so that it is a probability distribution (each row should add to 1.0)
    - `emit_p` : This is a matrix of size 12 x N, where N is the number of unique words in the corpus, which for each POS (the row) gives the probability that this POS in the output sequence corresponds to a specific word (the column) in the input sequence; again, you should collect frequency statistics about the relationship between POS and words, and normalize so that every row sums to 1.0.

Then run the algorithm on all the sentences in the tagged corpus, and determine the accuracy of the Viterbi algorithm. Again, the accuracy is calculated on each word, not on sentences as a whole.

Report your results as a raw accuracy score, and in the two ways that were suggested on slide 12 of Lecture 11: percentage above the baseline established in Part A, and Cohen's Kappa.

In [ ]:
```python
@njit
def viterbi(
    obs_sequence: np.ndarray,
    num_states: int,
    start_p: np.ndarray,
    trans_p: np.ndarray,
    emit_p: np.ndarray,
):
    T = len(obs_sequence)
    V = np.zeros((T, num_states))
    path = np.zeros((T, num_states), dtype=np.int32)

    # Initialize base cases
    V[0, :] = start_p + emit_p[:, obs_sequence[0]]

    # Run Viterbi for t > 0
    for t in range(1, T):
        for st in range(num_states):
            prob = V[t - 1] + trans_p[:, st]
            best_prev_state = np.argmax(prob)
            V[t, st] = prob[best_prev_state] + emit_p[st, obs_sequence[t]]
            path[t, st] = best_prev_state

    # The final probability and the path
    opt = []
    max_prob = np.max(V[-1, :])
    previous = np.argmax(V[-1, :])

    # Follow the path back to the first observation
    for t in range(T - 1, -1, -1):
        opt.insert(0, previous)
        previous = path[t, previous]

    return opt, max_prob
```

In [ ]:
```python
pos_mapping = dict(zip(all_tags, range(len(all_tags))))
num_states = 12
start_p = np.zeros(shape=(1, num_states))
trans_p = np.zeros(shape=(num_states, num_states))
```

In [ ]:
```python
for sent in tagged_sentences:
    first_word = sent[0]
    word, pos = first_word
    start_p[0][pos_mapping[pos]] += 1
    for a in range(len(sent) - 1):
        b = a + 1
        _, pos_i = sent[a]
        _, pos_j = sent[b]
        i = pos_mapping[pos_i]
        j = pos_mapping[pos_j]
        trans_p[i][j] += 1

start_p = np.log(start_p / start_p.sum(axis=1, keepdims=True))
trans_p = np.log(trans_p / trans_p.sum(axis=1, keepdims=True))
```

In [ ]:
```python
N = len(tagged_word_frequencies)
emit_p = np.zeros(shape=(num_states, N))
all_words = list(set([w for (w, _) in all_tagged_words]))
word_mapping = dict(zip(all_words, range(len(all_words))))
```

In [ ]:
```python
for word, pos_freqs in tagged_word_frequencies.items():
    for pos, freq in pos_freqs.items():
        i = pos_mapping[pos]
        j = word_mapping[word]
        emit_p[i][j] = freq

emit_p = np.log(emit_p / emit_p.sum(axis=1, keepdims=True))
```

```
/tmp/ipykernel_135724/1933126962.py:7: RuntimeWarning: divide by zero encountered in log
  emit_p = np.log(emit_p / emit_p.sum(axis=1, keepdims=True))
```

In [ ]:
```
obs_sequences = []
y_test = []
for sent in tagged_sentences:
    obs_sequence = []
    pos_sequence = []
    for word, pos in sent:
        obs_sequence.append(word_mapping[word])
        pos_sequence.append(pos_mapping[pos])
    obs_sequences.append(obs_sequence)
    y_test.append(pos_sequence)
```

In [ ]:
```
num_correct = 0
total = 0
for idx in trange(len(y_test)):
    y_pred, _ = viterbi(
        np.array(obs_sequences[idx]), num_states, start_p, trans_p, emit_p
    )
    num_correct += (np.asarray(y_test[idx]) == y_pred).sum()
    total += len(y_pred)

viterbi_accuracy = num_correct / total
```

```
100%|██████████| 57340/57340 [00:02<00:00, 25900.00it/s]
```

In [ ]:
```
def cohens_kappa(accuracy: float, baseline_accuracy: float) -> float:
    numerator = accuracy - baseline_accuracy
    denominator = 1 - baseline_accuracy
    k = numerator / denominator
    return k
```

In [ ]:
```
results = {"baseline": baseline_accuracy, "viterbi": viterbi_accuracy}
results = {
    "method": ["baseline", "viterbi"],
    "accuracy": [baseline_accuracy, viterbi_accuracy],
    "pct_above_baseline": np.round([0.0, viterbi_accuracy - baseline_accuracy], 3),
    "cohens_kappa": np.round(
        [0.0, cohens_kappa(viterbi_accuracy, baseline_accuracy)], 3
    ),
}
results_df = pd.DataFrame(results)
results_df
```

Out[ ]:

	method	accuracy	pct_above_baseline	cohens_kappa
0	baseline	0.957078	0.000	0.000
1	viterbi	0.975385	0.018	0.427

Part C:

Next, you will need to develop an LSTM model to solve this problem. You may find it useful to refer to the following, which presents an approach in Keras.

https://www.kaggle.com/code/tanyadayanand/pos-tagging-using-rnn/notebook

You must do the following for this part:

- Develop your code in Pytorch (of course!);
- Use pretrained GloVe embeddings of dimension 200 and update them with the brown sentences; if you run

into problems with RAM, you may use a smaller embedding dimension;
- Truncate all sentences to a maximum of length 100 tokens, and pad shorter sentences (as in the reference above);
- Use an LSTM model and try several different choices for the parameters to the layer:
    - `hidden_size` : Try several different widths for the layer
    - `bidirectional` : Try unidirectional (False) and bidirectional (True)
    - `num_layers` : Try 1 layer and 2 layers
    - `dropout` : In the case of 2 layers, try several different dropouts, including 0.
- Use early stopping with `patience = 50` ;
  You do not have to try every possible combination of these parameter choices; a good strategy is to try them separately, and then try a couple of combinations of the best choices of each.

It is your choice about the other hyperparameters.

Provide a brief discussion of what you discovered, your best loss and accuracy measures for validation, and three versions of your testing accuracy, as in Part B.

In [ ]:
```
SPECIAL_TOKENS = ["[PAD]"]
EMBEDDING_DIM = 200
BASE_EMBEDDING_SIZE = 400000
PADDING_IDX = 0
```

In [ ]:
```
vocab = []
embeddings = []
unique_words = set(all_words)

for t in SPECIAL_TOKENS:
    vocab.append(t)
    embeddings.append(
        torch.normal(mean=0, std=1, size=(EMBEDDING_DIM,), dtype=torch.float32)
    )

with open("/data/glove_6B/glove.6B.200d.txt") as f:
    for line in tqdm(f.readlines(), total=BASE_EMBEDDING_SIZE):
        content = line.split()
        word, vector = content[0], content[1:]
        if word in unique_words:
            vocab.append(word)
            vector = np.array(vector, dtype=np.float32)
            embeddings.append(torch.from_numpy(vector))
```

```
100%|██████████| 400000/400000 [00:03<00:00, 120466.51it/s]
```

In [ ]:
```
vocab_set = set(vocab)
```

In [ ]:
```
for word in tqdm(unique_words):
    if word not in vocab_set:
        vocab.append(word)
        embeddings.append(
            torch.normal(mean=0, std=1, size=(EMBEDDING_DIM,), dtype=torch.float32)
        )
```

```
100%|██████████| 56057/56057 [00:00<00:00, 600329.12it/s]
```

In [ ]:
```
embeddings = torch.stack(embeddings, dim=0)
```

In [ ]:
```
embeddings.size()
```

Out[ ]:
```
torch.Size([56058, 200])
```

In [ ]:
```
class Tokenizer:
    def __init__(
        self,
        vocabulary: List[str],
        special_tokens: List[str],
        dtype: torch.dtype = torch.int64,
    ) -> None:
        self.vocabulary = vocabulary
        self.special_tokens = special_tokens
        self.embedding_table = dict(zip(self.vocabulary, range(len(self.vocabulary))))
        self.reverse_embedding_table = {
            value: key for key, value in self.embedding_table.items()
        }
        self.embedding_dim = len(self.embedding_table)
        self.dtype = dtype

    def encode(self, sequence: List[str]) -> torch.Tensor:
        sequence_ids = torch.tensor(
            [[self.embedding_table[s] for s in sequence]], dtype=self.dtype
        )
        return sequence_ids

    def decode(
        self, sequence_ids: torch.Tensor, skip_special_tokens: bool = True
    ) -> List[str]:
        tokens = [self.reverse_embedding_table[s] for s in sequence_ids.numpy()]
        if skip_special_tokens:
            tokens = list(filter(lambda t: t not in set(self.special_tokens), tokens))
        return tokens
```

In [ ]:
```
class NER_Dataset(Dataset):
    def __init__(
        self,
        num_tags: int,
        vocab_size: int,
        X: List[torch.Tensor],
        y: List[torch.Tensor],
        one_hot_targets: bool = False,
        dtype: torch.dtype = torch.float32,
    ) -> None:
        self.num_tags = num_tags
        self.vocab_size = vocab_size
        self.X = X
        self.y = y
        self.one_hot_targets = one_hot_targets
        self.dtype = dtype
        self._one_hot_matrix = torch.eye(self.num_tags, dtype=self.dtype)

    def __len__(self) -> int:
        return len(self.X)

    def __getitem__(self, index: int) -> Tuple[torch.Tensor, torch.Tensor]:
        X = self.X[index]
        y = self.y[index]
        if self.one_hot_targets:
            y = self._one_hot_matrix[y]
        return X, y
```

In [ ]:
```
def collate_fn(batch: List[Tuple[torch.Tensor, torch.Tensor]]) -> Tuple[torch.Tensor, torch.Tensor,
    # Find the maximum sequence length in this batch
    max_seq_len = max([b[0].size(1) for b in batch])
    one_hot_targets = len(batch[0][1].size()) + 1 == 3

    padded_sequences = []
    padded_target_sequences = []
    mask = torch.zeros((len(batch), max_seq_len), dtype=bool)
    non_pad_idx_count = 0

    for idx, (input_sequence, target_sequence) in enumerate(batch):

        seq_padding = ((0, max_seq_len - input_sequence.size(1), 0, 0)
            if one_hot_targets else (0, max_seq_len - input_sequence.size(1)))

        mask[idx][:input_sequence.size(1)] = 1
        non_pad_idx_count += input_sequence.size(1)
        target_padding = ((0, 0, 0, max_seq_len - input_sequence.size(1))
            if one_hot_targets else (0, max_seq_len - input_sequence.size(1)))

        padded_sequences.append(
            F.pad(input_sequence, seq_padding)
        )

        # Pad each target to have the same length and stack all
        padded_target_sequences.append(
            F.pad(target_sequence, target_padding)
        )

    padded_sequences = torch.cat(padded_sequences, dim=0)
    padded_target_sequences = torch.stack(padded_target_sequences, dim=0)

    return padded_sequences, padded_target_sequences, mask, non_pad_idx_count
```

```
In [ ]: class Model(nn.Module):
    def __init__(
        self,
        num_tags: int,
        embed_dim: int,
        embeddings: torch.Tensor,
        hidden_size: int,
        bidirectional: bool,
        num_layers: int,
        dropout: float,
        device: torch.device,
        freeze_embedding_weights: bool = True,
        padding_idx: int = PADDING_IDX,
    ) -> None:
        super().__init__()

        self.embed_dim = embed_dim
        self.num_tags = num_tags
        self.hidden_size = hidden_size
        self.bidirectional = bidirectional
        self.D = 2 if self.bidirectional else 1
        self.num_layers = num_layers
        self.dropout = dropout
        self.device = device
        self.padding_idx = padding_idx

        self.embed = nn.Embedding.from_pretrained(
            embeddings=embeddings,
            padding_idx=self.padding_idx,
            freeze=freeze_embedding_weights,
        )

        self.lstm = nn.LSTM(
            input_size=self.embed_dim,
            hidden_size=self.hidden_size,
            num_layers=self.num_layers,
            bidirectional=self.bidirectional,
        )

        self.fcn = nn.ModuleList(
            [
                nn.Linear(in_features=self.D * self.hidden_size, out_features=self.num_tags)
            ]
        )

        self.to(self.device)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        sequence_len = x.size(1)
        output = self.embed(x)
        output, (h_n, c_n) = self.lstm(output)
        output = output.contiguous().view(-1, self.D * self.hidden_size)
        for layer in self.fcn:
            output = layer(output)
        output = output.view(-1, sequence_len, self.num_tags)
        return output

    def predict(self, input_sequences: torch.Tensor) -> torch.Tensor:
        if self.training:
            self.eval()

        with torch.no_grad():
            proba = self.forward(input_sequences.to(self.device)).cpu()

        proba = F.softmax(proba, dim=-1)

        return proba
```

In [ ]:
```
# set device
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is being used.")
else:
    device = torch.device("cpu")
    print("CPU is being used.")
```

```
GPU is being used.
```

In [ ]:
```
def create_sequences(
    tagged_sentences: List[List[Tuple[str, str]]],
    max_sequence_len: Optional[int] = 100,
) -> Tuple[List[List[str]], List[List[str]]]:
    input_sequences = []
    target_sequences = []
    for sent in tagged_sentences:
        input_sequence = []
        target_sequence = []
        num_iters = len(sent)
        if max_sequence_len and max_sequence_len < num_iters:
            num_iters = max_sequence_len
        for idx in range(num_iters):
            input_sequence.append(sent[idx][0])
            target_sequence.append(sent[idx][1])
        input_sequences.append(input_sequence)
        target_sequences.append(target_sequence)
    return input_sequences, target_sequences
```

In [ ]:
```
input_sequences, target_sequences = create_sequences(tagged_sentences=tagged_sentences,
                                                     max_sequence_len=100)
```

In [ ]:
```
tok = Tokenizer(vocabulary=vocab, special_tokens=SPECIAL_TOKENS)

X = [tok.encode(sequence=input_sequence) for input_sequence in input_sequences]

# +1 because we ignore PADDING_IDX and offset by 1
y = [
    torch.tensor([pos_mapping[pos] + 1 for pos in target_sequence], dtype=torch.int64)
    for target_sequence in target_sequences
]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2)
```

In [ ]:
```
BS = 128
# +1 since we ignore PADDING_IDX and need to add offset
NUM_CLASSES = len(pos_mapping) + 1
VOCAB_SIZE = len(vocab)
ONE_HOT_TARGETS = False
NUM_WORKERS = 6

train_dataset = NER_Dataset(
    num_tags=NUM_CLASSES,
    vocab_size=VOCAB_SIZE,
    X=X_train,
    y=y_train,
    one_hot_targets=ONE_HOT_TARGETS,
    dtype=torch.float32 if ONE_HOT_TARGETS else torch.int64
)

val_dataset = NER_Dataset(
    num_tags=NUM_CLASSES,
    vocab_size=VOCAB_SIZE,
    X=X_val,
    y=y_val,
    one_hot_targets=ONE_HOT_TARGETS,
    dtype=torch.float32 if ONE_HOT_TARGETS else torch.int64
)

test_dataset = NER_Dataset(
    num_tags=NUM_CLASSES,
    vocab_size=VOCAB_SIZE,
    X=X_test,
    y=y_test,
    one_hot_targets=ONE_HOT_TARGETS,
    dtype=torch.float32 if ONE_HOT_TARGETS else torch.int64
)

train_dataloader = DataLoader(
    dataset=train_dataset,
    batch_size=BS,
    shuffle=True,
    num_workers=NUM_WORKERS,
    collate_fn=collate_fn
)

val_dataloader = DataLoader(
    dataset=val_dataset, batch_size=BS, num_workers=NUM_WORKERS, collate_fn=collate_fn
)

test_dataloader = DataLoader(
    dataset=test_dataset, batch_size=BS, num_workers=NUM_WORKERS, collate_fn=collate_fn
)
```

In [ ]:
```
model = Model(
    num_tags=NUM_CLASSES,
    embed_dim=EMBEDDING_DIM,
    embeddings=embeddings,
    hidden_size=512,
    bidirectional=True,
    num_layers=2,
    dropout=0.5,
    freeze_embedding_weights=False,
    device=device,
)
```

In [ ]:
```
loss_fn = nn.CrossEntropyLoss(ignore_index=PADDING_IDX)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)

def train(
    num_epochs: int,
    save_model_dir: Union[Path, str],
    history: Dict[str, Any],
    early_stopping: bool = True,
    patience: int = 3,
) -> Dict[str, List[float]]:
    model.train()

    UPDATE_INTERVAL = 8
    best_epoch_idx = 0
    p = 0

    save_model_dir = Path(save_model_dir)
```

```
  if not save_model_dir.exists():
      save_model_dir.mkdir()

  if "loss" not in history:
      history["train_accuracy"] = []
      history["train_loss"] = []
      history["val_accuracy"] = []
      history["val_loss"] = []

  for epoch in range(num_epochs):

    with tqdm(total=len(train_dataloader), unit="batch",
              desc=f"Epoch {epoch+1}") as pbar:

      all_correct = 0
      N = 0
      running_loss = 0.0

      for idx, (X, y, mask, non_pad_idx_count) in enumerate(train_dataloader):
        X, y = X.cuda(), y.cuda()

        optimizer.zero_grad()

        logits = model(X)

        loss = loss_fn(logits.contiguous().view(-1, NUM_CLASSES),
                       y.contiguous().view((-1, NUM_CLASSES) if
                                           ONE_HOT_TARGETS else -1))

        loss.backward()

        running_loss += (loss.cpu().item() * non_pad_idx_count)

        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()

        proba = F.softmax(logits[mask], dim=-1)
        y_pred = torch.argmax(proba, dim=-1)

        num_correct = (y_pred == y[mask]).sum().cpu().item()

        train_accuracy = num_correct / non_pad_idx_count

        all_correct += num_correct
        N += non_pad_idx_count

        if idx % UPDATE_INTERVAL == 0 or idx == len(train_dataloader) - 1:
          pbar.update(min(UPDATE_INTERVAL, len(train_dataloader) - pbar.n))
          pbar.set_postfix(loss=loss.cpu().item(), train_accuracy=train_accuracy)

      train_loss = running_loss / N
      train_accuracy = all_correct / N

      history["train_loss"].append(train_loss)
      history["train_accuracy"].append(train_accuracy)

      pbar.set_postfix(train_loss=train_loss, train_accuracy=train_accuracy)

      all_correct = 0
      N = 0
      running_loss = 0.0
      with torch.no_grad():
        for (X, y, mask, non_pad_idx_count) in val_dataloader:
          X, y = X.cuda(), y.cuda()

          logits = model(X)

          val_loss = loss_fn(logits.contiguous().view(-1, NUM_CLASSES),
                       y.contiguous().view((-1, NUM_CLASSES) if
                                           ONE_HOT_TARGETS else -1))

          running_loss += (val_loss.cpu().item() * non_pad_idx_count)

          proba = F.softmax(logits[mask], dim=-1)
          y_pred = torch.argmax(proba, dim=-1)

          num_correct = (y_pred == y[mask]).sum().cpu().item()

          all_correct += num_correct
          N += non_pad_idx_count

      val_loss = running_loss / N
      val_accuracy = all_correct / N

      history["val_accuracy"].append(val_accuracy)
      history["val_loss"].append(val_loss)

      pbar.set_postfix(train_loss=train_loss,
                      train_accuracy=train_accuracy,
                      val_loss=val_loss,
                      val_accuracy=val_accuracy)

      if early_stopping:
        if p == patience:
          break

        if val_accuracy > history["val_accuracy"][best_epoch_idx]:
          p = 0
          best_epoch_idx = epoch
          torch.save(model.state_dict(),
                    save_model_dir.joinpath(
                        f"epoch_{best_epoch_idx}_val_acc_{val_accuracy}.pth"))
        else:
          p += 1

  return history
```

In [ ]:
```
history = dict()
```

In [ ]:
```
history = train(num_epochs=30,
                history=history,
                early_stopping=True,
                save_model_dir="saved_models_3_dropout_0.5_layers_2_bidirectional",
                patience=50)
```

```
Epoch 1: 100%|██████████| 287/287 [00:32<00:00,  8.80batch/s, train_accuracy=0.812, train_loss=0.5
6, val_accuracy=0.913, val_loss=0.261]
Epoch 2: 100%|██████████| 287/287 [00:32<00:00,  8.79batch/s, train_accuracy=0.921, train_loss=0.23
5, val_accuracy=0.927, val_loss=0.216]
Epoch 3: 100%|██████████| 287/287 [00:33<00:00,  8.67batch/s, train_accuracy=0.931, train_loss=0.20
4, val_accuracy=0.933, val_loss=0.196]
Epoch 4: 100%|██████████| 287/287 [00:33<00:00,  8.67batch/s, train_accuracy=0.936, train_loss=0.18
6, val_accuracy=0.934, val_loss=0.186]
Epoch 5: 100%|██████████| 287/287 [00:33<00:00,  8.65batch/s, train_accuracy=0.94, train_loss=0.17
2, val_accuracy=0.933, val_loss=0.193]
Epoch 6: 100%|██████████| 287/287 [00:33<00:00,  8.56batch/s, train_accuracy=0.942, train_loss=0.16
5, val_accuracy=0.936, val_loss=0.181]
Epoch 7: 100%|██████████| 287/287 [00:34<00:00,  8.43batch/s, train_accuracy=0.944, train_loss=0.15
6, val_accuracy=0.936, val_loss=0.179]
Epoch 8: 100%|██████████| 287/287 [00:33<00:00,  8.47batch/s, train_accuracy=0.944, train_loss=0.15
1, val_accuracy=0.934, val_loss=0.183]
Epoch 9: 100%|██████████| 287/287 [00:33<00:00,  8.51batch/s, train_accuracy=0.945, train_loss=0.14
6, val_accuracy=0.933, val_loss=0.19]
Epoch 10: 100%|██████████| 287/287 [00:33<00:00,  8.44batch/s, train_accuracy=0.946, train_loss=0.1
41, val_accuracy=0.937, val_loss=0.18]
Epoch 11: 100%|██████████| 287/287 [00:34<00:00,  8.44batch/s, train_accuracy=0.947, train_loss=0.1
37, val_accuracy=0.938, val_loss=0.176]
Epoch 12: 100%|██████████| 287/287 [00:33<00:00,  8.50batch/s, train_accuracy=0.949, train_loss=0.1
32, val_accuracy=0.939, val_loss=0.177]
Epoch 13: 100%|██████████| 287/287 [00:34<00:00,  8.43batch/s, train_accuracy=0.95, train_loss=0.12
8, val_accuracy=0.936, val_loss=0.173]
Epoch 14: 100%|██████████| 287/287 [00:34<00:00,  8.42batch/s, train_accuracy=0.951, train_loss=0.1
25, val_accuracy=0.937, val_loss=0.172]
Epoch 15: 100%|██████████| 287/287 [00:34<00:00,  8.38batch/s, train_accuracy=0.952, train_loss=0.1
23, val_accuracy=0.94, val_loss=0.177]
Epoch 16: 100%|██████████| 287/287 [00:34<00:00,  8.40batch/s, train_accuracy=0.952, train_loss=0.1
2, val_accuracy=0.942, val_loss=0.159]
Epoch 17: 100%|██████████| 287/287 [00:34<00:00,  8.39batch/s, train_accuracy=0.952, train_loss=0.1
2, val_accuracy=0.942, val_loss=0.18]
Epoch 18: 100%|██████████| 287/287 [00:33<00:00,  8.47batch/s, train_accuracy=0.953, train_loss=0.1
18, val_accuracy=0.937, val_loss=0.169]
Epoch 19: 100%|██████████| 287/287 [00:34<00:00,  8.38batch/s, train_accuracy=0.953, train_loss=0.1
17, val_accuracy=0.943, val_loss=0.164]
Epoch 20: 100%|██████████| 287/287 [00:33<00:00,  8.45batch/s, train_accuracy=0.953, train_loss=0.1
16, val_accuracy=0.942, val_loss=0.17]
Epoch 21: 100%|██████████| 287/287 [00:34<00:00,  8.43batch/s, train_accuracy=0.953, train_loss=0.1
15, val_accuracy=0.943, val_loss=0.157]
Epoch 22: 100%|██████████| 287/287 [00:33<00:00,  8.47batch/s, train_accuracy=0.954, train_loss=0.1
15, val_accuracy=0.942, val_loss=0.169]
Epoch 23: 100%|██████████| 287/287 [00:33<00:00,  8.44batch/s, train_accuracy=0.954, train_loss=0.1
13, val_accuracy=0.942, val_loss=0.168]
Epoch 24: 100%|██████████| 287/287 [00:34<00:00,  8.43batch/s, train_accuracy=0.954, train_loss=0.1
14, val_accuracy=0.943, val_loss=0.164]
Epoch 25: 100%|██████████| 287/287 [00:34<00:00,  8.42batch/s, train_accuracy=0.954, train_loss=0.1
13, val_accuracy=0.942, val_loss=0.158]
Epoch 26: 100%|██████████| 287/287 [00:33<00:00,  8.46batch/s, train_accuracy=0.954, train_loss=0.1
13, val_accuracy=0.943, val_loss=0.162]
Epoch 27: 100%|██████████| 287/287 [00:33<00:00,  8.50batch/s, train_accuracy=0.954, train_loss=0.1
12, val_accuracy=0.943, val_loss=0.166]
Epoch 28: 100%|██████████| 287/287 [00:34<00:00,  8.36batch/s, train_accuracy=0.954, train_loss=0.1
12, val_accuracy=0.942, val_loss=0.168]
Epoch 29: 100%|██████████| 287/287 [00:33<00:00,  8.46batch/s, train_accuracy=0.954, train_loss=0.1
12, val_accuracy=0.943, val_loss=0.16]
Epoch 30: 100%|██████████| 287/287 [00:33<00:00,  8.49batch/s, train_accuracy=0.954, train_loss=0.1
12, val_accuracy=0.941, val_loss=0.167]
```

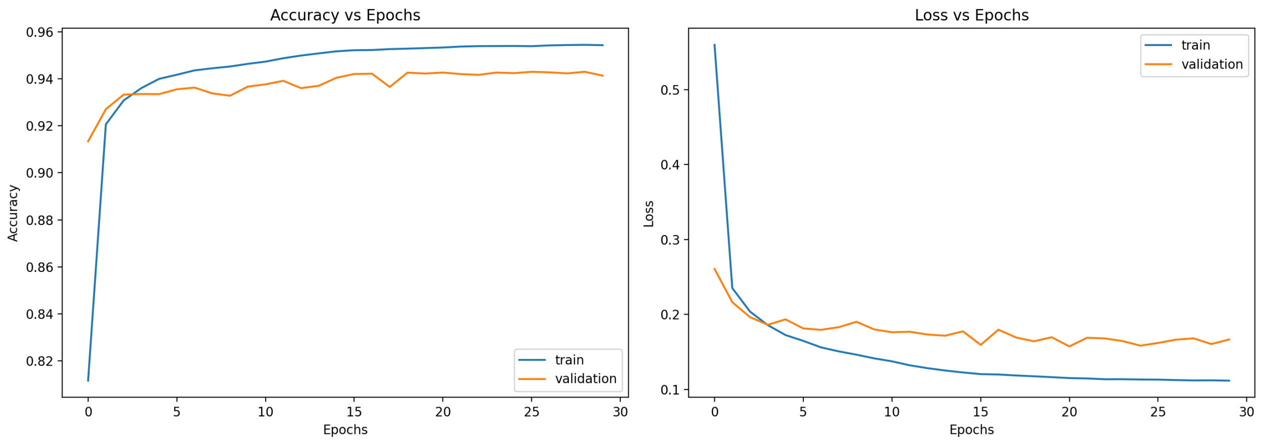
In [ ]:

```python
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(14, 5),
                        tight_layout=True)

axs[0].plot(history["train_accuracy"], label="train")
axs[0].plot(history["val_accuracy"], label="validation")
axs[0].set_xlabel("Epochs")
axs[0].set_ylabel("Accuracy")
axs[0].set_title("Accuracy vs Epochs")
axs[0].legend()
```

```
axs[1].plot(history["train_loss"], label="train")
axs[1].plot(history["val_loss"], label="validation")
axs[1].set_xlabel("Epochs")
axs[1].set_ylabel("Loss")
axs[1].set_title("Loss vs Epochs")
axs[1].legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f4415948b20>
```



In [ ]:
```
model = Model(
    num_tags=NUM_CLASSES,
    embed_dim=EMBEDDING_DIM,
    embeddings=embeddings,
    hidden_size=512,
    bidirectional=True,
    num_layers=2,
    dropout=0.5,
    freeze_embedding_weights=False,
    device=device,
)

model.load_state_dict(
    torch.load("saved_models_3_dropout_0.5_layers_2_bidirectional/epoch_25_val_acc_0.94297654500793
)
```

Out[ ]:
```
<All keys matched successfully>
```

In [ ]:
```
N = 0
num_correct = 0

print("Evaluating test")
for (X, y, mask, non_pad_idx_count) in tqdm(test_dataloader):
    logits = model.predict(X)
    y_pred = torch.argmax(logits[mask], dim=-1)
    num_correct += (y_pred == y[mask]).sum().item()
    N += non_pad_idx_count
```

```
Evaluating test
100%|██████████| 90/90 [00:03<00:00, 24.19it/s]
```

In [ ]:
```
lstm_accuracy = num_correct / N
```

In [ ]:
```
lstm_accuracy
```

Out[ ]:
```
0.941993046323755
```

In [ ]:
```
results_df.loc[2] = ["LSTM", lstm_accuracy,
                     np.round(lstm_accuracy - baseline_accuracy, 3),
                     cohens_kappa(lstm_accuracy, baseline_accuracy)]
results_df
```

Out[ ]:

	method	accuracy	pct_above_baseline	cohens_kappa
0	baseline	0.957078	0.000	0.000000
1	viterbi	0.975385	0.018	0.427000
2	LSTM	0.941993	-0.015	-0.351442

Part D:

Provide an analysis of what experiments you conducted with hyperparameters, what your results were, and in particular comment on how the two methods compare, especially given that one has *no* choice of hyperparameters, and one has *many* choices of parameters. How useful was the flexibility of choice in hyperparameters in Part C?

I tried many different parameters in constructing the LSTM including the number of hidden units, number of stacked layers, deciding whether to use a bidirectional LSTM, dropout, and fully connected layer configurations. Specifically I tried playing around with hidden units of 128, 256, and 512, finding 512 to perform the best. I compared 1 and 2 stacked layers in the LSTM, finding that 2 stacked layers performs better. I also tested various settings of dropout including 0.0 and 0.5, discovering that the added regularization of `dropout=0.5` lead to slightly better performance. Adding weight decay to the optimizer also helped very slightly. Additionally, a bidirectional LSTM leads to a very slight improvement compared to an unidirectional LSTM. My finalized hyperparameters can be seen above in the notebook cells. One interesting observation is that tuning additional hyperparameters only very slightly improved performance while increasing running time, which leads me to believe that a larger dataset is more important oftentimes (assuming the data is usable).

While it was nice to have this flexibility, it made optimization of the accuracy when detection part-of-speech tags much more difficult. In fact, the LSTM approach performed worse in terms of accuracy than the baseline and Viterbi methods. I hypothesize this since the previous two methods (baseline and Viterbi algorithm) rely on the entire corpus whereas to prevent data leakage, the deep learning model is split into train, validation, and test splits. This leads me to think that the deep learning model when given enough data will eventually surpass both methods. With less data, I can see the appeal of the Hidden Markov Models.

Optional:

You might want to try doing this problem with a transformer model such as BERT. There are plenty of blog posts out there describing the details, and, as usual, chatGPT would have plenty of things to say about the topic....