CS 4650 "Natural Language Processing" - Project 1

Georgia Tech, Spring 2023

(Instructor: Wei Xu; TAs: Mounica Maddela, Ben Podrazhansky, Marcus Ma, Rahul Katre)

Part 0. Google Colab Setup

Welcome to the first full programming project for CS 4650! If you're new to Google Colab we recommend looking at this intro notebook before getting started with this project. In short, Colab is a Jupyter notebook environment that runs in the cloud, it's recommended for all of the programming projects in this course due to its availability, ease of use, and hardware accessibility. Some features that you may find especially useful are on the left hand side, these being:

- Table of contents: displays the sections of the notebook made using text cells
- Variables: useful for debugging and see current values of variables
- Files: useful or uploading or downloading any files you upload to Colab or write while working on the projects

To begin this project, make a copy of this notebook and save it to your local drive so that you can edit it.

If you want GPU's (which will improve training times), you can always change your instance type to GPU by going to Runtime -> Change runtime type -> Hardware accelerator.

If you're new to PyTorch, or simply want a refresher, we recommend you start by looking through these Introduction to PyTorch slides and this interactive PyTorch Basics notebook. Additionally, this Text Sentiment notebook will provide some insight into working with PyTorch for NLP specific problems.

Part 1. Loading and Preprocessing Data [10 points]

The following cell loads the OnionOrNot dataset, and tokenizes each data item

```
!curl https://raw.githubusercontent.com/lukefeilberg/onion/master/OnionOrNot.csv > Oni
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1903k  100 1903k    0     0  8689k      0 --:--:-- --:--:-- --:--:-- 8689k
```

```
# DO NOT MODIFY #
import torch
import random
import numpy as np

RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

# this is how we select a GPU if it's avalible on your computer or in the Colab enviro
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Part 1.1 Preprocessing definitions:

The following cell define some methods to clean the dataset. Do not edit it, but feel free to take a look at some of the operations it's doing.

```
# DO NOT MODIFY THIS BLOCK
# example code taken from fast-bert

import re
import html

def spec_add_spaces(t: str) -> str:
    "Add spaces around / and # in `t`. \n"
    return re.sub(r"([/#\n])", r" \1 ", t)

def rm_useless_spaces(t: str) -> str:
    "Remove multiple spaces in `t`."
    return re.sub(" {2,}", " ", t)

def replace_multi_newline(t: str) -> str:
    return re.sub(r"(\n(\s)*){2,}", "\n", t)

def fix_html(x: str) -> str:
    "List of replacements from html strings in `x`."
    re1 = re.compile(r"  +")
    x = (
        x.replace("#39;", "'")
        .replace("amp;", "&")
        .replace("#146;", "'")
        .replace("nbsp;", " ")
        .replace("#36;", "$")
        .replace("\\n", "\n")
        .replace("quot;", "'")
        .replace("<br />", "\n")
        .replace('\\"', '"')
        .replace(" @.@ ", ".")
        .replace(" @-@ ", "-")
        .replace(" @,@ ", ",")
        .replace("\\", " \\ ")
    )
    return re1.sub(" ", html.unescape(x))

def clean_text(input_text):
    text = fix_html(input_text)
    text = replace_multi_newline(text)
    text = spec_add_spaces(text)
    text = rm_useless_spaces(text)
    text = text.strip()
    return text
```

Part 1.2 Clean the data using the methods above and tokenize it using NLTK

```
import pandas as pd
import nltk
from tqdm import tqdm

nltk.download('punkt')
df                = pd.read_csv("OnionOrNot.csv")
df["tokenized"] = df["text"].apply(lambda x: nltk.word_tokenize(clean_text(x.lower()))
```

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

Here's what the dataset looks like. You can index into specific rows with pandas, and try to guess some of these yourself :). If you're unfamiliar with pandas, it's a extremely useful and popular library for data analysis and manipulation. You can find their documentation here.

Pandas primary data structure is a DataFrame. The following cell will print out the basic information of this structure, including the labeled axes (both columns and rows) as well as show you what the first n (default=5) rows look like

```
df.head()
```

	text	label	tokenized
0	Entire Facebook Staff Laughs As Man Tightens P...	1	[entire, facebook, staff, laughs, as, man, tig...
1	Muslim Woman Denied Soda Can for Fear She Coul...	0	[muslim, woman, denied, soda, can, for, fear, ...
2	Bold Move: Hulu Has Announced That They’re Gon...	1	[bold, move, :, hulu, has, announced, that, th...
	Despondent Jeff Bezos Realizes He’ll Have To		[despondent, jeff, bezos, realizes, he, ’,

DataFrames can be indexed using .iloc[ ], this primarily uses interger based indexing and supports a single integer (i.e. 42), a list of integers (i.e. [1, 5, 42]), or even a slice (i.e. 7:42).

```
df.iloc[42]
```

```
    text         Customers continued to wait at drive-thru even...
    label                                                        0
    tokenized    [customers, continued, to, wait, at, drive-thr...
    Name: 42, dtype: object
```

Part 1.3 Split the dataset into training, validation, and testing

Now that we've loaded this dataset, we need to split the data into train, validation, and test sets. A good explanation of why we need these different sets can be found in subsection 2.2.5 of Eisenstein but at the end it comes down to having a trustworthy and generalized model. The validation (sometimes called a development or tuning) set is used to help choose hyperparameters for our model, whereas the training set is used to fit the learned parameters (weights and biases) to the task. The test set is used to provide a final unbiased evaluation of our trained model, hopefully providing some insight into how it would actually do in production. Each of these sets should be disjoint from the others, to prevent any "peeking" that could unfairly influence our understanding of the model's accuracy.

In addition to these different sets of data, we also need to create a vocab map for words in our Onion dataset, which will map tokens to numbers. This will be useful later, since torch PyTorch use tensors of sequences of numbers as inputs. **Go to the following cell, and fill out split_train_val_test and generate_vocab_map.**

```
from nltk.lm.preprocessing import padded_everygram_pipeline
# BEGIN - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.
from collections import Counter
PADDING_VALUE = 0
UNK_VALUE     = 1
# END - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.


# split_train_val_test
# This method takes a dataframe and splits it into train/val/test splits.
# It uses the props argument to split the dataset appropriately.
#
# args:
# df - the entire dataset DataFrame
# props - proportions for each split in the order of [train, validation, test].
#         the last value of the props array is repetitive, but we've kept it for clar
```

```
#
# returns:
# train DataFrame, val DataFrame, test DataFrame
#
def split_train_val_test(df, props=[.8, .1, .1]):
    assert round(sum(props), 2) == 1 and len(props) >= 2
    train_df, test_df, val_df = None, None, None

    ## YOUR CODE STARTS HERE (~3-5 lines of code) ##
    # hint: you can use df.iloc to slice into specific indexes or ranges.
    length = df.shape[0]
    train_df = df.iloc[:int(length * props[0])]
    test_df = df.iloc[int(length * props[0]): int(length * (props[0] + props[1]))]
    val_df = df.iloc[int(length * (props[0] + props[1])):]

    ## YOUR CODE ENDS HERE ##

    return train_df, val_df, test_df

# generate_vocab_map
# This method takes a dataframe and builds a vocabulary to unique number map.
# It uses the cutoff argument to remove rare words occuring <= cutoff times.
# *NOTE*: "" and "UNK" are reserved tokens in our vocab that will be useful
# later. You'll also find the Counter imported for you to be useful as well.
#
# args:
# df     - the entire dataset this mapping is built from
# cutoff - we exclude words from the vocab that appear less than or
#          eq to cutoff
#
# returns:
# vocab - dict[str] = int
#         In vocab, each str is a unique token, and each dict[str] is a
#         unique integer ID. Only elements that appear > cutoff times appear
#         in vocab.
#
# reversed_vocab - dict[int] = str
#                  A reversed version of vocab, which allows us to retrieve
#                  words given their unique integer ID. This map will
#                  allow us to "decode" integer sequences we'll encode using
#                  vocab!
#
def generate_vocab_map(df, cutoff=2):
    vocab          = {"": PADDING_VALUE, "UNK": UNK_VALUE}
    reversed_vocab = None

    ## YOUR CODE STARTS HERE (~5-15 lines of code) ##
    # hint: start by iterating over df["tokenized"]
    count = 0
    d = {}
    uid = 1
    for r in df["tokenized"]:
      for t in r:
        if str(t) in d.keys():
          d[str(t)] += 1
        else:
          d[str(t)] = 1
        if d.get(str(t)) > cutoff and str(t) not in vocab.keys():
          vocab[str(t)] = uid+1
          uid += 1
    count = 0
    reversed_vocab = {}
    for v in vocab.keys():
      reversed_vocab[str(count)] = v
      count += 1


    ## YOUR CODE ENDS HERE ##

    return vocab, reversed vocab
```

With the methods you have implemented above, we can now split the dataset into training, validation, and testing sets and generate our dictionaries mapping from word tokens to IDs (and vice versa).

Note: The props list currently being used splits the dataset so that 80% of samples are used to train, and the remaining 20% are evenly split between training and validation. How you split your dataset is itself a major choice and something you would need to consider in your own projects. Can you think of why?

```
df                          = df.sample(frac=1)
train_df, val_df, test_df  = split_train_val_test(df, props=[.8, .1, .1])
train_vocab, reverse_vocab = generate_vocab_map(train_df)


# This line of code will help test your implementation, the expected output is the sar
#   in the above cell. Try out some different values to ensure it works, but for submi
#   [.8, .1, .1]

(len(train_df) / len(df)), (len(val_df) / len(df)), (len(test_df) / len(df))
```

```
     (0.8, 0.1, 0.1)
```

Part 1.4 Building a Dataset Class

PyTorch has custom Dataset Classes that have very useful extentions, we want to turn our current pandas DataFrame into a subclass of Dataset so that we can iterate and sample through it for minibatch updates. **In the following cell, fill out the HeadlineDataset class.** Refer to PyTorch documentation on Dataset Classes for help.

```
# BEGIN - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.
from torch.utils.data import Dataset
# END - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.

# HeadlineDataset
# This class takes a Pandas DataFrame and wraps in a Torch Dataset.
# Read more about Torch Datasets here:
# https://pytorch.org/tutorials/beginner/basics/data_tutorial.html
#
class HeadlineDataset(Dataset):

    # initialize this class with appropriate instance variables
    def __init__(self, vocab, df, max_length=50):
        # For this method: We would *strongly* recommend storing the dataframe
        #                  itself as an instance variable, and keeping this method
        #                  very simple. Leave processing to __getitem__.
        #
        #                  Sometimes, however, it does make sense to preprocess in
        #                  __init__. If you are curious as to why, read the aside at t
        #                  bottom of this cell.
        #

        ## YOUR CODE STARTS HERE (~3 lines of code) ##
        self.df = df
        self.vocab = vocab
        self.max_length = max_length

        return
        ## YOUR CODE ENDS HERE ##

    # return the length of the dataframe instance variable
    def __len__(self):

        df_len = None
        ## YOUR CODE STARTS HERE (1 line of code) ##
        df_len = len(self.df)


        ## YOUR CODE ENDS HERE ##
        return df_len

    # __getitem__
    #
    # Converts a dataframe row (row["tokenized"]) to an encoded torch LongTensor,
    # using our vocab map created using generate_vocab_map. Restricts the encoded
    # headline length to max_length.
    #
    # The purpose of this method is to convert the row - a list of words - into
    # a corresponding list of numbers.
    #
    # i.e. using a map of {"hi": 2, "hello": 3, "UNK": 0}
    # this list ["hi", "hello", "NOT_IN_DICT"] will turn into [2, 3, 0]
    #
    # returns:
    # tokenized_word_tensor - torch.LongTensor
    #                         A 1D tensor of type Long, that has each
    #                         token in the dataframe mapped to a number.
    #                         These numbers are retrieved from the vocab_map
    #                         we created in generate_vocab_map.
    #
    #                         **IMPORTANT**: if we filtered out the word
    #                         because it's infrequent (and it doesn't exist
    #                         in the vocab) we need to replace it w/ the UNK
    #                         token
    #
    # curr_label            - int
    #                         Binary 0/1 label retrieved from the DataFrame.
    #
    def __getitem__(self, index: int):
        tokenized_word_tensor = None
        curr_label            = None
        ## YOUR CODE STARTS HERE (~3-7 lines of code) ##
        tokenized_word_tensor = []
        row = self.df.iloc[index]
        for t in row["tokenized"]:
          if str(t) in self.vocab:
            tokenized_word_tensor.append(self.vocab[str(t)])
          else:
            tokenized_word_tensor.append(self.vocab["UNK"])
        tokenized_word_tensor = torch.tensor(tokenized_word_tensor).long().to(device)
        curr_label = row["label"]

        ## YOUR CODE ENDS HERE ##
        return tokenized_word_tensor, curr_label



#
# Completely optional aside on preprocessing in __init__.
#
# Sometimes the compute bottleneck actually ends up being in __getitem__.
# In this case, you'd loop over your dataset in __init__, passing data
# to __getitem__ and storing it in another instance variable. Then,
# you can simply return the preprocessed data in __getitem__ instead of
# doing the preprocessing.
#
# There is a tradeoff though: can you think of one?
#
```

```
from torch.utils.data import RandomSampler

train_dataset = HeadlineDataset(train_vocab, train_df)
val_dataset   = HeadlineDataset(train_vocab, val_df)
test_dataset  = HeadlineDataset(train_vocab, test_df)

# Now that we're wrapping our dataframes in PyTorch datsets, we can make use of PyTorc
#   define how our DataLoaders sample elements from the HeadlineDatasets
train_sampler = RandomSampler(train_dataset)
val_sampler   = RandomSampler(val_dataset)
test_sampler  = RandomSampler(test_dataset)
```

Part 1.5 Finalizing our DataLoader

We can now use PyTorch DataLoaders to batch our data for us. **In the following cell fill out collate_fn.** Refer to PyTorch documentation on DataLoaders for help.

```
from tkinter.constants import Y
# BEGIN - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.
from torch.nn.utils.rnn import pad_sequence
# END - DO NOT CHANGE THESE IMPORTS/CONSTANTS OR IMPORT ADDITIONAL PACKAGES.

# collate_fn
# This function is passed as a parameter to Torch DataSampler. collate_fn collects
# batched rows, in the form of tuples, from a DataLoader and applies some final
# pre-processing.
#
# Objective:
# In our case, we need to take the batched input array of 1D tokenized_word_tensors,
# and create a 2D tensor that's padded to be the max length from all our tokenized_wor
# in a batch. We're moving from a Python array of tuples, to a padded 2D tensor.
#
# *HINT*: you're allowed to use torch.nn.utils.rnn.pad_sequence (ALREADY IMPORTED)
#
# Finally, you can read more about collate_fn here: https://pytorch.org/docs/stable/da
#
# args:
# batch - PythonArray[tuple(tokenized_word_tensor: 1D Torch.LongTensor, curr_label: ir
#         len(batch) == BATCH_SIZE
#
# returns:
# padded_tokens - 2D LongTensor of shape (BATCH_SIZE, max len of all tokenized_word_te
# y_labels      - 1D FloatTensor of shape (BATCH_SIZE)
#
def collate_fn(batch, padding_value=PADDING_VALUE):
    padded_tokens, y_labels = None, None
    ## YOUR CODE STARTS HERE (~4-8 lines of code) ##
```

```
        pt = []
        y = []
        for b in batch:
          pt.append(b[0])
          y.append(b[1])
        padded_tokens = torch.nn.utils.rnn.pad_sequence(pt, True, padding_value).long()
        y_labels = torch.tensor(y, dtype = torch.float).to(device)

        ## YOUR CODE ENDS HERE ##
        return padded_tokens, y_labels

    from torch.utils.data import DataLoader
    BATCH_SIZE = 16

    train_iterator = DataLoader(train_dataset, batch_size=BATCH_SIZE, sampler=train_sample
    val_iterator   = DataLoader(val_dataset, batch_size=BATCH_SIZE, sampler=val_sampler, c
    test_iterator  = DataLoader(test_dataset, batch_size=BATCH_SIZE, sampler=test_sampler,


    # Use this to test your collate_fn implementation.
    # You can look at the shapes of x and y or put print statements in collate_fn while ru

    for x, y in test_iterator:
        print(x, y)
        print(f'x: {x.shape}')
        print(f'y: {y.shape}')
        break
    test_iterator = DataLoader(test_dataset, batch_size=BATCH_SIZE, sampler=test_sampler,
```

```
    tensor([[  26,   24,  169,    1, 6120,   20,    1, 3102,    1,   11,    1,   43,
             147,   10, 5724,   11,  140,   11, 5945,    1,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 177,    6,  394, 8222, 1213,  125, 7709, 2927,  596,  174,    2,  379,
              22,    1,  269,  107,    7, 1168,   11,   73,    9,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [1053,    6,  284,   10,    1,    6,   36,   44, 1400,   54,   22, 3510,
             663,    9, 1053,   11,   60,  931, 1331,    3,  186,   10,  970,    7,
             443,    0,    0,    0,    0,    0,    0,    0],
            [ 450,  522,  120,   24, 7164,   44,  494,   89,  609,   10, 1447, 1188,
              13,   10, 4987,    1, 2693,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [  77,  285, 1939,   71,    1,   32,  685,   11,   60,    1,    3, 4487,
             179, 2741,   60,  572,  134,    1,    7, 2619,    9, 2620, 3830, 2135,
              60,  198,  224, 1868,  106,    0,    0,    0],
            [ 142,    6, 2185, 1422,    9,   23,   83, 1373,  235,  336,    1, 3951,
            3294,    7, 7754,    1,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 871,  925,  172,    2,    1,  570,   32,  871,  925,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [  83,    6, 2181,    6,   54, 4282, 1319,    1,  700, 6000,    9,    1,
             187,   49,    9,   23,  270,  819,   32,   38,  120,   38, 2148,  219,
             382,    3, 6756,    0,    0,    0,    0,    0],
            [1618, 1243,   90, 3482,    3, 3728, 7430,   39, 5431,  658,  350,   14,
              18, 2662,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 400,  786,  946, 1415,    2, 1836, 1177,    3,   42, 6016,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 912, 1152, 2017,  272,    3,  855, 1198,  500,    1, 1309,    6, 1762,
            3028,  780,    9,   97,  755,    4,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [  83,    6, 2024,   15, 3020,    6,   54,   99,  797,    1,  102,   27,
             116,   10,  101,    3,  945,  569,   12,   18,  664,   15,   33,  163,
             270,  819,   38,    9,   23,   59, 1575, 3809],
            [5313,  381,  477,  226,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [  47,    6, 3884,  391,   82, 1509,   43,  133,  224,    1, 5868,    1,
             494,   89,   10,    1,   16,  145,    1,   89,   24, 1974,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 177,    6,   38,   84,  649, 1073,    7, 1583, 3516,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0],
            [ 296,   37,  153,    3,  674, 2819, 8884,   32,  114,  655,  107,  127,
               2, 1603, 1712,    0,    0,    0,    0,    0,    0,    0,    0,    0,
               0,    0,    0,    0,    0,    0,    0,    0]], device='cuda:0') tens
          device='cuda:0')
    x: torch.Size([16, 32])
    y: torch.Size([16])
```

Part 2: Modeling [10 pts]

Let's move to modeling, now that we have dataset iterators that batch our data for us. In the following code block, you'll build a feed-forward neural network implementing a neural bag-of-words baseline, NBOW-RAND, described in section 2.1 of this paper. You'll find this page useful for understanding the different layers and this page useful for how to put them into action.

The core idea behind this baseline is that after we embed each word for a document, we average the embeddings to produce a single vector that hopefully captures some general information spread across the sequence of embeddings. This means we first turn each document of length *n* into a matrix of *nxd*, where *d* is the dimension of the embedding. Then we average this matrix to produce a vector of length *d*, summarizing the contents of the document and proceed with the rest of the network.

While you're working through this implementation, keep in mind how the dimensions change and what each axes represents, as documents will be passed in as minibatches requiring careful selection of which axes you apply certain operations too. You're more than welcome to experiment with the architecture of this network as well outside of the basic setup we describe below, such as adding in other layers, to see how this changes your results.

Part 2.1 Define the NBOW model class

```
# BEGIN - DO NOT CHANGE THESE IMPORTS OR IMPORT ADDITIONAL PACKAGES.
import torch.nn as nn
# END - DO NOT CHANGE THESE IMPORTS OR IMPORT ADDITIONAL PACKAGES.

class NBOW(nn.Module):
    # Instantiate layers for your model-
    #
    # Your model architecture will be a feed-forward neural network.
    #
    # You'll need 3 nn.Modules at minimum
    # 1. An embeddings layer (see nn.Embedding)
    # 2. A linear layer (see nn.Linear)
    # 3. A sigmoid output (see nn.Sigmoid)
    #
    # HINT: In the forward step, the BATCH_SIZE is the first dimension.
    #
    def __init__(self, vocab_size, embedding_dim):
        super().__init__()
        ## YOUR CODE STARTS HERE (~4 lines of code) ##

        self.embedding_layer = nn.Embedding(vocab_size, embedding_dim)
        self.linear_layer = nn.Linear(embedding_dim, 1)
        self.output = nn.Sigmoid()

        ## YOUR CODE ENDS HERE ##

    # Complete the forward pass of the model.
    #
    # Use the output of the embedding layer to create
    # the average vector, which will be input into the
    # linear layer.
    #
    # args:
    # x - 2D LongTensor of shape (BATCH_SIZE, max len of all tokenized_word_tensor))
    #     This is the same output that comes out of the collate_fn function you comple
    #
    # returns:
    # 1D FloatTensor of shape (BATCH_SIZE)
    def forward(self, x):
        ## YOUR CODE STARTS HERE (~4-5 lines of code) ##
        embd = self.embedding_layer(x)
        lin = self.linear_layer(torch.mean(embd, dim = 1))
        out = torch.flatten(self.output(lin))

        return out
        ## YOUR CODE ENDS HERE ##
```

Part 2.2 Initialize the NBOW classification model

Since the NBOW model is rather basic, assuming you haven't added any additional layers, there's really only one hyperparameter for the model architecture: the size of the embedding dimension.

The vocab_size parameter here is based on the number of unique words kept in the vocab after removing those occurring too infrequently, so this is determined by our dataset and is in turn not a true hyperparameter (though the cutoff we used previously might be). The embedding_dim parameter dictates what size vector each word can be embedded as.

If you added additional linear layers to the NBOW model then the input/output dimensions of each would be considered a hyperparameter you might want to experiment with. While the sizes are constrained based on previous & following layers (the number of dimensions need to match for the matrix multiplication), whatever sequence you used could still be tweaked in various ways.

A special note concerning the model initialization: We're specifically sending the model to the device set in Part 1, to speed up training if the GPU is available. **Be aware**, you'll have to ensure other tensors are on the same device inside your training and validation loops.

```
model = NBOW(vocab_size    = len(train_vocab.keys()),
             embedding_dim = 300).to(device)
```

Part 2.3 Instantiate the loss function and optimizer

In the following cell, **select and instantiate an appropriate loss function and optimizer.**

Hint: we already use sigmoid in our model. What loss functions are availible for binary classification? Feel free to look at PyTorch docs for help!

```
#while Adam is already imported, you can try other optimizers as well
from torch.optim import Adam

criterion, optimizer = None, None
### YOUR CODE GOES HERE ###

criterion = nn.BCELoss()
optimizer = Adam(model.parameters())

### YOUR CODE ENDS HERE ###
```

At this point, we have a NBOW model to classify headlines as being real or fake and a loss function/optimizer to train the model using the training dataset.

Part 3: Training and Evaluation [10 Points]

The final part of this HW involves training the model, and evaluating it at each epoch. **Fill out the train and test loops below. Treat real headlines as False, and Onion headlines as True.** Feel free to look at PyTorch docs for help!

```
# returns the total loss calculated from criterion
def train_loop(model, criterion, optim, iterator):
    model.train()
    total_loss = 0
    for x, y in tqdm(iterator):
        ### YOUR CODE STARTS HERE (~6 lines of code) ###
        optim.zero_grad()
        out = model(x)
        loss = criterion(out, y)
        loss.backward()
        optim.step()
        total_loss += loss.item()
        ### YOUR CODE ENDS HERE ###
    return total_loss

# returns:
# - true: a Python boolean array of all the ground truth values
#         taken from the dataset iterator
# - pred: a Python boolean array of all model predictions.
def val_loop(model, iterator):
    true, pred = [], []
    ### YOUR CODE STARTS HERE (~8 lines of code) ###
    model.eval()
    for x, y in tqdm(iterator):
      p = torch.round(model(x))
      for i in range(len(y)):
        true.append(bool(p[i]))
        pred.append(bool(y[i]))

    ### YOUR CODE ENDS HERE ###
    return true, pred
```

Part 3.1 Define the evaluation metrics

We also need evaluation metrics that tell us how well our model is doing on the validation set at each epoch and later how well the model does on the held-out test set. **Complete the functions in the following cell.** You'll find subsection 4.4.1 of Eisenstein useful for this task.

```
# DO NOT IMPORT ANYTHING IN THIS CELL. You shouldn't need any external libraries.

# accuracy
#
# What fraction of classifications are correct?
#
# true: ground truth, Python list of booleans.
# pred: model predictions, Python list of booleans.
# return: accuracy bounded between [0, 1]
#
def accuracy(true, pred):
    acc = None
    ## YOUR CODE STARTS HERE (~2-5 lines of code) ##

    acc_list = torch.tensor([1 if pred[i] == true[i] else 0 for i in range(len(true))
    acc = torch.sum(acc_list)/ acc_list.size(dim=0)

    ## YOUR CODE ENDS HERE ##
    return acc

# binary_f1
#
# A method to calculate F-1 scores for a binary classification task.
#
# args -
# true: ground truth, Python list of booleans.
# pred: model predictions, Python list of booleans.
# selected_class: Boolean - the selected class the F-1
#                 is being calculated for.
#
# return: F-1 score between [0, 1]
#
def binary_f1(true, pred, selected_class=True):
    f1 = None
    ## YOUR CODE STARTS HERE (~10-15 lines of code) ##
    #only true[i] == selected_class
    true_positive_list = [1 if true[i] == selected_class and pred[i]  == selected_clas
    false_negative_list = [1 if true[i] == selected_class and not(pred[i] == selected_
    true_negative_list = [1 if true[i] != selected_class and pred[i] != selected_class
    false_positive_list = [1 if true[i] != selected_class and pred[i] == selected_clas
    tp = sum(true_positive_list)
    fn = sum(false_negative_list)
    tn = sum(true_negative_list)
    fp = sum(false_positive_list)
    if tp+fp == 0:
      precision = 0
    else:
      precision = tp/(tp+fp)
    if tp+fn == 0:
      recall = 0
    else:
      recall = tp/(tp+fn)
    if (precision + recall) == 0:
      f1 = 0
    else:
      f1 = (2*precision*recall)/(precision + recall)


    ## YOUR CODE ENDS HERE ##
    return f1

# binary_macro_f1
#
# Averaged F-1 for all selected (true/false) classes.
#
# args -
# true: ground truth, Python list of booleans.
# pred: model predictions, Python list of booleans.
#
# return: F-1 score between [0, 1]
#
def binary_macro_f1(true, pred):
    averaged_macro_f1 = None
    ## YOUR CODE STARTS HERE (1 line of code) ##
    averaged_macro_f1 = (binary_f1(true, pred, True) + binary_f1(true, pred, False))/2

    ## YOUR CODE ENDS HERE ##
    return averaged_macro_f1

# To test your eval implementation, let's see how well the untrained model does on our
# It should do pretty poorly, but this can be random because of the initialization of
true, pred = val_loop(model, val_iterator)
print()
print(f'Binary Macro F1: {binary_macro_f1(true, pred)}')
print(f'Accuracy: {accuracy(true, pred)}')
```

```
100%|██████████████████| 150/150 [00:00<00:00, 253.70it/s]
Binary Macro F1: 0.46096857472018293
Accuracy: 0.4658333361148834
```

At this point, we have our datasets defined and split, our model and training tools/loops, and evaluation metrics so we can finally move on to train our model and see how it does!

Part 4: Actually training the model [1 point]

Watch your model train :D You should be able to achieve a validation F-1 score of at least .8 if everything went correctly. **Feel free to adjust the number of epochs to prevent overfitting or underfitting and to play with your model hyperparameters/optimizer & loss function.**

```
TOTAL_EPOCHS = 6
for epoch in range(TOTAL_EPOCHS):
    train_loss = train_loop(model, criterion, optimizer, train_iterator)
    true, pred = val_loop(model, val_iterator)
    print(f"EPOCH: {epoch}")
    print(f"TRAIN LOSS: {train_loss}")
    print(f"VAL F-1: {binary_macro_f1(true, pred)}")
    print(f"VAL ACC: {accuracy(true, pred)}")
```

```
100%|██████████| 1200/1200 [00:06<00:00, 187.86it/s]
100%|██████████| 150/150 [00:00<00:00, 250.99it/s]
EPOCH: 0
TRAIN LOSS: 620.3446144610643
VAL F-1: 0.7739599559468822
VAL ACC: 0.8029166460037231
100%|██████████| 1200/1200 [00:06<00:00, 190.66it/s]
100%|██████████| 150/150 [00:00<00:00, 244.10it/s]
EPOCH: 1
TRAIN LOSS: 402.4617502242327
VAL F-1: 0.8337455027605079
VAL ACC: 0.8487499952316284
100%|██████████| 1200/1200 [00:06<00:00, 187.83it/s]
100%|██████████| 150/150 [00:00<00:00, 254.63it/s]
EPOCH: 2
TRAIN LOSS: 315.853447265923
VAL F-1: 0.8488844230852726
VAL ACC: 0.8604166507720947
100%|██████████| 1200/1200 [00:06<00:00, 188.00it/s]
100%|██████████| 150/150 [00:00<00:00, 252.80it/s]
EPOCH: 3
TRAIN LOSS: 266.03455318696797
VAL F-1: 0.8530235317294305
VAL ACC: 0.8616666793823242
100%|██████████| 1200/1200 [00:06<00:00, 191.79it/s]
100%|██████████| 150/150 [00:00<00:00, 250.57it/s]
EPOCH: 4
TRAIN LOSS: 228.3734827330336
VAL F-1: 0.8601146730890372
VAL ACC: 0.8700000047683716
100%|██████████| 1200/1200 [00:06<00:00, 187.87it/s]
100%|██████████| 150/150 [00:00<00:00, 248.58it/s]EPOCH: 5
TRAIN LOSS: 201.04515806492418
VAL F-1: 0.859845725481799
VAL ACC: 0.8700000047683716
```

We can also look at the models performance on the held-out test set, using the same val_loop we wrote earlier.

```
true, pred = val_loop(model, test_iterator)
print()
print(f"TEST F-1: {binary_macro_f1(true, pred)}")
print(f"TEST ACC: {accuracy(true, pred)}")
```

```
    100%|██████████████| 150/150 [00:00<00:00, 255.62it/s]
    TEST F-1: 0.8663618260933696
    TEST ACC: 0.8741666674613953
```

Part 5: Analysis [5 points]

Answer the following questions:

1. What happens to the vocab size as you change the cutoff in the cell below? Can you explain this in the context of Zipf's Law?

Answer:

Zipf's Law states that the frequencies of an event are inversely proportional to rank r. In this case, we can consider the cutoff as the ranking, and the number of vocabulary that appears more than the "cutoff" value is related to the frequency. When cutoff = 0, the vocab_size of the whole vocabulary without cutoff is 25388. When cutoff = 1, vocab_size = 13298. When cutoff = 2, vocab_size = 9540. When cutoff = 3, vocab_size = 7612. We can see the cutoff is inversely proportional to the vocab_size. As the cutoff increase, the vocab_size is smaller.

```
tmp_vocab, _ = generate_vocab_map(train_df, cutoff = 3)
len(tmp_vocab)
# tmp_vocab
```

```
    7612
```

2. Can you describe what cases the model is getting wrong in the witheld test-set?

Answer:

When the predicted label differs from the true label, the sequences are incorrectly predicted cases in the test set. We can observe that nearly all incorrect sequences have padding. And, 'UNK' appears very often in incorrect sequences. Because padding is adding '' to make the list of the max_length, adding elements that are not in the original list may cause the label to be predicted incorrectly. The 'UNK' are words that are not in the vocabulary. So the original word is undetected in the vocabulary, possibly making the label incorrect. The cases that the model is getting wrong more often in the test set I observed are having padding and 'UNK' in the sequences.

To do this, you'll need to create a new val_train_loop (`val_train_loop_incorrect`) so it returns incorrect sequences **and** you'll need to decode these sequences back into words. Thankfully, you've already created a map that can convert encoded sequences back to regular English: you will find the `reverse_vocab` variable useful.

```
# i.e. using a reversed map of {"hi": 2, "hello": 3, "UNK": 0}
# we can turn [2, 3, 0] into this => ["hi", "hello", "UNK"]
```

```
# Implement this however you like! It should look very similar to val_loop.
# Pass the test_iterator through this function to look at errors in the test set.
def val_train_loop_incorrect(model, iterator):
  model.train()
  incorrect_seq = []
  for x, y in tqdm(iterator):
    p = torch.round(model(x))
    for i in range(len(y)):
      if p[i] != y[i]:
        seq = ""
        for j in x[i]:
          seq += " " + reverse_vocab[str(j.item())]
        incorrect_seq.append(seq)

  return incorrect_seq


val_train_loop_incorrect(model, test_iterator)
```

```
 ,
  ' reality check : trip to UNK cost less than UNK stadium          ',
  " autistic reporter , michael UNK , enchanted by prison 's UNK routine
",
  ' this 19-year-old is paying her way through college by naming over UNK chinese
babies      ',
  ' police use racist texts to mock the previous racist text scandal
',
  ' how police are UNK their tactics                     ',
  ' 8-year-old follows UNK lawmaker around capitol until he drops welfare bill
',
  ' man missing for 3 years found where last seen                 ',
  ' kim jong UNK announces plan to bring moon to north korea              ',
  ' war on string may be UNK , says cat general                 ',
  ' lawrence UNK interviews empty chair                  ',
  ' cleveland hero charles UNK rewarded with burgers for life               ',
  ' who drummer UNK moon asked to perform at 2012 olympics ceremony
',
  ' anti-vax mom changes her tune when all 7 of her children come down with UNK
cough        ',
  ' sleeping man UNK by laptop , phone , UNK like egyptian UNK buried with all
his UNK       ',
  " detectives UNK UNK anthony 's ' i killed my daughter ' UNK on reddit
",
  ' weather channel accused of UNK bias                    ',
  ' wtf ? UNK UNK body set for rebrand over use of UNK             ',
  ' scientists discover eating serves UNK other than UNK anxiety
',
  " man thanks god he 's not sexually attracted to children
",
  ' stormy daniels ‘ 60 minutes ’ interview leads to spike in pornhub searches
for anderson cooper                 ',
  " rob UNK ’ s wife gets out photo album to prove to him he 's met tom brady
",
  ' man named kim adds ‘ mr. ’ to resume , lands job       ',
  ' UNK meet with top pentagon leaders to talk about peace          ']
```

Part 6: LSTM Model [Extra-Credit, 4 points]

Part 6.1 Define the RecurrentModel class

Something that has been overlooked so far in this project is the sequential structure to language: a word typically only has a clear meaning because of its relationship to the words before and after it in the sequence, and the feed-forward network of Part 2 cannot model this type of data. A solution to this, is the use of recurrent neural networks. These types of networks not only produce some output given some step from a sequence, but also update their internal state, hopefully "remembering" some information about the previous steps in the input sequence. Of course, they do have their own faults, but we'll cover this more thoroughly later in the semester.

Your task for the extra credit portion of this assignment, is to implement such a model below using a LSTM. Instead of averaging the embeddings as with the FFN in Part 2, you'll instead feed all of these embeddings to a LSTM layer, get its final output, and use this to make your prediction for the class of the headline.

```
class RecurrentModel(nn.Module):
    # Instantiate layers for your model-
    #
    # Your model architecture will be an optionally bidirectional LSTM,
    # followed by a linear + sigmoid layer.
    #
    # You'll need 4 nn.Modules
    # 1. An embeddings layer (see nn.Embedding)
    # 2. A bidirectional LSTM (see nn.LSTM)
    # 3. A Linear layer (see nn.Linear)
    # 4. A sigmoid output (see nn.Sigmoid)
    #
    # HINT: In the forward step, the BATCH_SIZE is the first dimension.
    # HINT: Think about what happens to the linear layer's hidden_dim size
    #       if bidirectional is True or False.
    #
    def __init__(self, vocab_size, embedding_dim, hidden_dim, \
                 num_layers=1, bidirectional=True):
        super().__init__()
        ## YOUR CODE STARTS HERE (~4 lines of code) ##
        self.embd = nn. Embedding(vocab_size, embedding_dim)
        self.lstm = nn. LSTM(embedding_dim, hidden_dim, num_layers= num_layers, bidire
        if bidirectional:
          self.linear = nn. Linear(hidden_dim * 2, 1)
        else:
          self.linear = nn. Linear(hidden_dim, 1)
        self.sig = nn. Sigmoid()


        ## YOUR CODE ENDS HERE ##

    # Complete the forward pass of the model.
    #
    # Use the last timestep of the output of the LSTM as input
    # to the linear layer. This will only require some indexing
    # into the correct return from the LSTM layer.
    #
    # args:
    # x - 2D LongTensor of shape (BATCH_SIZE, max len of all tokenized_word_tensor))
    #     This is the same output that comes out of the collate_fn function you comple
    #
    # returns:
    # 1D FloatTensor of shape (BATCH_SIZE)
    def forward(self, x):
        ## YOUR CODE STARTS HERE (~4-5 lines of code) ##
        em = self.embd(x)
        # print(em.shape)
        ls, _ = self.lstm(em)
        # print(ls[0].shape)
        li = self.linear(ls[:,-1, :])
        # print(li.shape)
        out = torch.flatten(self.sig(li))
        # print(out.shape)

        return out
        ## YOUR CODE ENDS HERE ##
```

Now that the RecurrentModel is defined, we'll reinitialize our dataset iterators so they're back at the start.

```
train_iterator = DataLoader(train_dataset, batch_size=BATCH_SIZE, sampler=train_sample
val_iterator   = DataLoader(val_dataset, batch_size=BATCH_SIZE, sampler=val_sampler, c
test_iterator  = DataLoader(test_dataset, batch_size=BATCH_SIZE, sampler=test_sampler,
```

Part 6.2 Initialize the LSTM classification model

Next we need to initialize our new model, as well as define it's optimizer and loss function as we did for the FFN. Feel free to use the same optimizer you did above, or see how this model reacts to different optimizers/learning rates than the FFN.

```
lstm_model = RecurrentModel(vocab_size    = len(train_vocab.keys()),
                            embedding_dim = 300,
                            hidden_dim    = 300,
                            num_layers    = 1,
                            bidirectional = True).to(device)


lstm_criterion, lstm_optimizer = None, None
### YOUR CODE STARTS HERE ###

lstm_criterion = nn.MSELoss()
lstm_optimizer = Adam(lstm_model.parameters())

### YOUR CODE ENDS HERE ###
```

Part 6.3 Training and Evaluation

Because the only difference between this model and the FFN is the internal structure, we can use the same methods as above to evaluate and train it. You should be able to achieve a validation F-1 score of at least .8 if everything went correctly. **Feel free to adjust the number of epochs to prevent overfitting or underfitting and to play with your model hyperparameters/optimizer & loss function.**

```
#Pre-training to see what accuracy we can get with random parameters
true, pred = val_loop(lstm_model, val_iterator)
print()
print(f'Binary Macro F1: {binary_macro_f1(true, pred)}')
print(f'Accuracy: {accuracy(true, pred)}')
```

```
100%|██████████| 150/150 [00:00<00:00, 166.66it/s]
Binary Macro F1: 0.270344189670795
Accuracy: 0.3620833456516266
```

```
#Watch the model train!
TOTAL_EPOCHS = 6
for epoch in range(TOTAL_EPOCHS):
    train_loss = train_loop(lstm_model, lstm_criterion, lstm_optimizer, train_iterato
    true, pred = val_loop(lstm_model, val_iterator)
    print(f"EPOCH: {epoch}")
    print(f"TRAIN LOSS: {train_loss}")
    print(f"VAL F-1: {binary_macro_f1(true, pred)}")
    print(f"VAL ACC: {accuracy(true, pred)}")
```

```
100%|██████████| 1200/1200 [00:10<00:00, 116.17it/s]
100%|██████████| 150/150 [00:00<00:00, 202.93it/s]
EPOCH: 0
TRAIN LOSS: 189.75631734682247
VAL F-1: 0.8138683341643669
VAL ACC: 0.8262500166893005
100%|██████████| 1200/1200 [00:09<00:00, 120.71it/s]
100%|██████████| 150/150 [00:00<00:00, 202.73it/s]
EPOCH: 1
TRAIN LOSS: 95.66249386500567
VAL F-1: 0.8401942254798014
VAL ACC: 0.8537499904632568
100%|██████████| 1200/1200 [00:09<00:00, 120.60it/s]
100%|██████████| 150/150 [00:00<00:00, 205.02it/s]
EPOCH: 2
TRAIN LOSS: 53.88344313201378
VAL F-1: 0.8345075529706607
VAL ACC: 0.8429166674613953
100%|██████████| 1200/1200 [00:10<00:00, 119.55it/s]
100%|██████████| 150/150 [00:00<00:00, 203.72it/s]
EPOCH: 3
TRAIN LOSS: 30.48796252151078
VAL F-1: 0.8519711992690606
VAL ACC: 0.8637499809265137
100%|██████████| 1200/1200 [00:09<00:00, 120.34it/s]
100%|██████████| 150/150 [00:00<00:00, 200.98it/s]
EPOCH: 4
TRAIN LOSS: 21.09183402778831
```

```
VAL F-1: 0.8550092297365329
VAL ACC: 0.8650000095367432
100%|████████████████| 1200/1200 [00:09<00:00, 120.78it/s]
100%|████████████████| 150/150 [00:00<00:00, 193.32it/s]EPOCH: 5
TRAIN LOSS: 16.165403183288504
VAL F-1: 0.8490708139556777
VAL ACC: 0.8608333468437195
```

```
#See how your model does on the held out data
true, pred = val_loop(lstm_model, test_iterator)
print()
print(f"TEST F-1: {binary_macro_f1(true, pred)}")
print(f"TEST ACC: {accuracy(true, pred)}")
```

```
100%|████████████████| 150/150 [00:00<00:00, 201.61it/s]
TEST F-1: 0.850558856819468
TEST ACC: 0.85916668176651
```

Part 7: Submit Your Homework

This is the end. Congratulations!

Now, follow the steps below to submit your homework in Gradescope:

1. Rename this ipynb file to 'CS4650_p1_GTusername.ipynb'. We recommend ensuring you have removed any extraneous cells & print statements, clearing all outputs, and using the Runtime --> Run all tool to make sure all output is update to date. Additionally, leaving comments in your code to help us understand your operations will assist the teaching staff in grading. It is not a requirement, but is recommended.
2. Click on the menu 'File' --> 'Download' --> 'Download .py'.
3. Click on the menu 'File' --> 'Download' --> 'Download .ipynb'.
4. Download the notebook as a .pdf document. Make sure the output from Parts 4 & 6.3 are captured so we can see how the loss, F1, & accuracy changes while training.
5. Upload all 3 files to GradeScope.