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  2742k      0 --:--:-- --:--:-- --:--:-- 2738k
```

```
# 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')
# read data from the csv file
df              = pd.read_csv("OnionOrNot.csv")
# tokenizing text data -> nltk.word_tokenize() deals with tokening and clean_text pru
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 set (sometimes called a development or tuning) 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 models can only use tensors of sequences of numbers as inputs. **Go to the following cell, and fill out split_train_val_test and generate_vocab_map.**

```
# 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. the last value of the props array
#         is repetitive, but we've kept it for clarity.
#
# 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.
    df_size = len(df)
    train_bound, val_bound, test_bound = int(props[0]*df_size), int((props[0] + props

    train_df, val_df, test_df = df.iloc[: train_bound], df.iloc[train_bound : val_bou


    ## 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.
#
# args:
# df - the entire dataset DataFrame
# 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"]

    # compute how much each word appears in the entire dataset given
    df_counter = Counter()
    for sample in df["tokenized"]:
        df_counter.update(sample)
    # build 'vocab' and 'reversed_vocab'
    reversed_vocab = {UNK_VALUE: "UNK", PADDING_VALUE: ""} # 0, 1

    # word_id starts from 2 ("vocab" already has 2 elements in it (PADDING_VALUE & UNK
    word_id = 2
    for word in df_counter:
        if (df_counter[word] > cutoff) and (word not in vocab):
            vocab[word] = word_id
            reversed_vocab[word_id] = word
            word_id += 1

    ## YOUR CODE ENDS HERE ##

    return vocab, reversed_vocab
```

```
# With the methods you have implemented above, we can now split the dataset into trair
#   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
#   remaining 20% are evenly split between training and validation. How you split your
#   choice and something you would need to consider in your own projects. Can you thir

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.vocab = vocab
        self.df = df
        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 = self.df.shape[0]


        ## YOUR CODE ENDS HERE ##
        return df_len

    # __getitem__
    #
    # Converts a dataframe row (row["tokenized"]) to an encoded torch LongTensor,
    # using our vocab map we 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) ##
        vocab = self.vocab
        curr_data = self.df.iloc[index]["tokenized"]
        tokenized_word_tensor = torch.LongTensor([vocab[word] if word in vocab else vo

        curr_label = self.df.iloc[index]["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.

```
# 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_wo
# 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: i
#         len(batch) == BATCH_SIZE
#
# returns:
# padded_tokens - 2D LongTensor of shape (BATCH_SIZE, max len of all tokenized_word_t
# 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) ##
    y_labels = torch.FloatTensor([data[1] for data in batch])

    pre_tokens = [data[0] for data in batch]
    padded_tokens = pad_sequence(pre_tokens, True, padding_value)
    ## 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([[ 174, 2903, 9461,    1,  321,  563,   36, 2202,    1, 1845, 2854,   12,
          223,    1,   33, 7473,    0,    0,    0,    0,    0],
        [  11, 1086,  829, 6476,  411,    1,   36,   37,   51,  158, 3298,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [   1, 2698, 8440,   33,  321,   33,  274,   52,    1,   41,  617, 4959,
           12, 1017,    0,    0,    0,    0,    0,    0,    0],
        [  25,  883,  289,  552,  301,   61,  181,  593,  289, 2774, 2775,   36,
         4604, 5446,   36,  113,    1,    1, 1095,   52,    3],
        [3472, 4372, 1332,    1, 2430,   52,   57, 8872,    1,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [4223,  120,   15,  431,    1,  217, 5503,   24,    1,  608,   52,    1,
          926,    0,    0,    0,    0,    0,    0,    0,    0],
        [3019,   61, 3210, 2774,    1, 6017, 6636, 4447,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [3123, 4184, 3475,  217, 5633, 8439,   52, 8594, 5089,   21,  605,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [3283,  411,   61, 2461,  158,  476, 3157,  940, 3310, 2128, 6877,  411,
           33,  343,  327,   21, 7368,  301,    0,    0,    0],
        [  68,  147, 2389,   33, 7046,  952, 8886,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [1642, 5893,   61, 5898,  977, 8704,  133,  263,    1,    1,   56,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [   1, 2710,    1,  165,    1,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [  11,   21, 1022, 6182,  619,  148,  149, 3427,  128, 3880,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [5754, 4651,  321,    1,    1,   21, 1652,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0],
        [7450, 6727,  770, 5105, 3296,   36,   81,  784,    1,    1, 6871,  784,
          326, 4128,   30,    0,    0,    0,    0,    0,    0],
        [1281,   36, 6892, 3256,    1, 1753, 1011,   24, 2486, 4671,   30,   31,
         2122,  977,    1, 5918,    1,    0,    0,    0,    0]]) tensor([1., 0.,
x: torch.Size([16, 21])
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

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 = nn.EmbeddingBag(vocab_size, embedding_dim)
        self.W = nn.Linear(embedding_dim, 1)
        self.activation = 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
    def forward(self, x):
        ## YOUR CODE STARTS HERE (~4-5 lines of code) ##
        embedded = self.embedding(x)
        output = self.W(embedded)
        output = self.activation(output)



        return output
        ## 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(), lr=0.0002)


### 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 True, and Onion headlines as False.** 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) ###

        # move data from cpu to gpu
        gold_labels = y.unsqueeze(1).to(device)
        headlines = x.to(device)
        # clear gradients
        optim.zero_grad()
        # predict headlines
        predicted_outputs = model(headlines)
        # compute loss
        loss = criterion(predicted_outputs, gold_labels)
        total_loss += loss
        # backpropagation
        loss.backward()
        # weight updates
        optim.step()

        ### 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) ###
    for sentences, gold_labels in iterator:
        # model prediction: this converting to True False values are actually a redunc
        predicted_outputs = [True if output >= 0.5 else False for output in model(sent

        # append records: predicted_output (True, False), gold_label (1, 0)
        for predicted_output, gold_label in zip(predicted_outputs, gold_labels):
            if predicted_output:
                pred.append(True)
            else:
                pred.append(False)

            if int(gold_label):
                true.append(True)
            else:
                true.append(False)
    ### 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 percent of classifications are correct?
#
# true: ground truth, Python list of booleans.
# pred: model predictions, Python list of booleans.
# return: percent accuracy bounded between [0, 1]
#
def accuracy(true, pred):
    acc = None
    ## YOUR CODE STARTS HERE (~2-5 lines of code) ##
    number_of_correct_predictions = sum(1 for prediction, label in zip(pred, true)
                                    if prediction == label)
    number_of_total_predictions = len(pred)

    acc = (number_of_correct_predictions)/(number_of_total_predictions)
    ## 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) ##
    # recall = (# True Positive)/(# True Positive + # of False Negative)
    # precision = (# True Positive)/(# True Positive + # of False Positive)
    # f1_measure = (2*recall*precision)/(recall + precision)
    number_of_true_pos = 0
    number_of_false_neg = 0
    number_of_false_pos = 0

    for predicted, label in zip(pred, true):
        if selected_class == label == predicted:
            # true positive case
            number_of_true_pos += 1

        if (selected_class == label) and (label != predicted):
            # false negative case
            number_of_false_neg += 1

        if (selected_class == predicted) and (label != predicted):
            # false positive case
            number_of_false_pos += 1
    #print(f'##########{number_of_true_pos}###########')

    # recall
    recall = number_of_true_pos/(number_of_true_pos + number_of_false_neg)
    # precision
    precision = number_of_true_pos/(number_of_true_pos + number_of_false_pos)
    # f1 measure
    f1 = (2*recall*precision)/(recall + precision)
    ## 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.
#
#
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(f'Binary Macro F1: {binary_macro_f1(true, pred)}')
print(f'Accuracy: {accuracy(true, pred)}')
```

```
    Binary Macro F1: 0.34252355464685647
    Accuracy: 0.3975
```

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 = 20
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)}")
```

```
    TRAIN LOSS: 353.0101318359375
    VAL F-1: 0.8497835624065907
    VAL ACC: 0.8595833333333334
    100%|██████████| 1200/1200 [00:09<00:00, 129.08it/s]
```

```
EPOCH: 9
TRAIN LOSS: 333.6537780761719
VAL F-1: 0.8572814885143081
VAL ACC: 0.8654166666666666
100%|██████████| 1200/1200 [00:09<00:00, 129.04it/s]
EPOCH: 10
TRAIN LOSS: 316.3703308105469
VAL F-1: 0.8566641110076736
VAL ACC: 0.8666666666666667
100%|██████████| 1200/1200 [00:09<00:00, 130.23it/s]
EPOCH: 11
TRAIN LOSS: 300.5408630371094
VAL F-1: 0.8600503031599604

VAL ACC: 0.8679166666666667
100%|██████████| 1200/1200 [00:09<00:00, 129.84it/s]
EPOCH: 12
TRAIN LOSS: 286.3997497558594
VAL F-1: 0.8547991938004469
VAL ACC: 0.8645833333333334
100%|██████████| 1200/1200 [00:09<00:00, 129.34it/s]
EPOCH: 13
TRAIN LOSS: 277.2650146484375
VAL F-1: 0.8574960398396717
VAL ACC: 0.8679166666666667
100%|██████████| 1200/1200 [00:09<00:00, 129.97it/s]
EPOCH: 14
TRAIN LOSS: 264.3644714355469
VAL F-1: 0.8668327938623632
VAL ACC: 0.8745833333333334
100%|██████████| 1200/1200 [00:09<00:00, 129.65it/s]
EPOCH: 15
TRAIN LOSS: 252.5731658935547
VAL F-1: 0.8632800648863015
VAL ACC: 0.8720833333333333
100%|██████████| 1200/1200 [00:09<00:00, 130.40it/s]
EPOCH: 16
TRAIN LOSS: 243.3887176513672
VAL F-1: 0.8686584347247346
VAL ACC: 0.87625
100%|██████████| 1200/1200 [00:09<00:00, 129.87it/s]
EPOCH: 17
TRAIN LOSS: 235.56724548339844
VAL F-1: 0.8654043358558988
VAL ACC: 0.8729166666666667
100%|██████████| 1200/1200 [00:09<00:00, 129.39it/s]
EPOCH: 18
TRAIN LOSS: 226.7405548095703
VAL F-1: 0.8612852790479649
VAL ACC: 0.8695833333333334
100%|██████████| 1200/1200 [00:09<00:00, 129.40it/s]
EPOCH: 19
TRAIN LOSS: 220.14549255371094
VAL F-1: 0.8633524206142633
VAL ACC: 0.8716666666666667
```

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)}")
```

```
    TEST F-1: 0.8537071149257807
    TEST ACC: 0.8654166666666666
```

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: As can be seen in the bottom graph plotted, as we increase the cutoff value, the size of the vocab decreases drastically. From the graph we can gain an intuition that the cutoff value and the vocab size are in inverse relation. Meaning that there are way more number of words having low frequencies than that of having high frequencies.

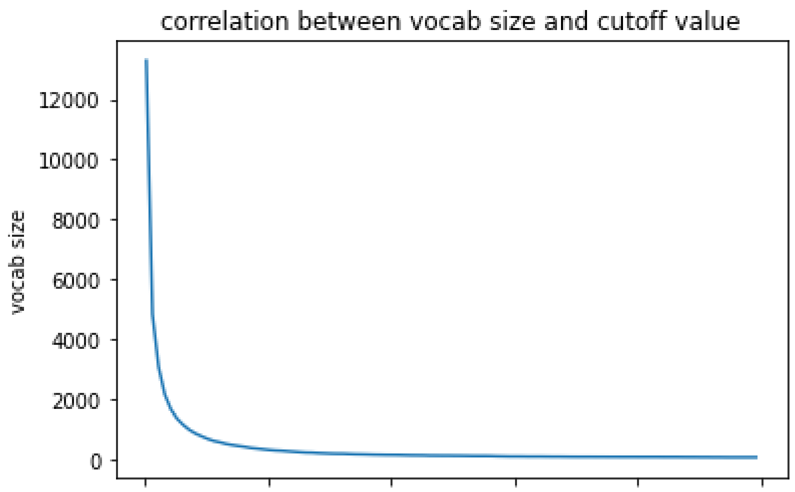
```
tmp_vocab, _ = generate_vocab_map(train_df, cutoff = 1)
len(tmp_vocab)
```

```
    13296
```

```
from matplotlib import pyplot as plt

# compute (cutoff value, vocab size) pairs
cutoff_vals = [i for i in range(1, 500, 5)]
vocab_sizes = [len(generate_vocab_map(train_df, cutoff = cutoff_val)[0]) for cutoff_va

# plot data
plt.title("correlation between vocab size and cutoff value")
plt.xlabel("cutoff value")
plt.ylabel("vocab size")
plt.plot(cutoff_vals, vocab_sizes)
plt.show()
```



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

Answer: In the case where the words in the given sentence is not in the vocab dictionary, meaning that for cases where the model cannot identify the words ("UNK"), it seems like the model has less countable predictions over the given data. Also, one problem with the feedforward nn is that it does not deal with the given sentence as sequeuntial data. Meaning that the sequence (ordering) of the words in a given sentence do influence the contextual meaning of the sentence and thus the model could show weakness upon understandig the correlation between the previous words to posterior words in the sentences.

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.

# using reverse_vocab built earlier in pre-processing step
def val_train_loop_incorrect(model, iterator):
    decoded_incorrect_predictions = []
    for sentences, gold_labels in iterator:
        # model predictions: this converting to True False values are actually a redur
        predicted_outputs = [True if output >= 0.5 else False for output in model(sent

        for sentence, predicted_output, gold_label in zip(sentences, predicted_outputs
            if predicted_output is (not int(gold_label)): # parentheses were necessary
```

```
                # append mis-predicted inputs
                decoded_incorrect_predictions.append([reverse_vocab[int(word_id)] for

        return decoded_incorrect_predictions



    incorrect_predictions = val_train_loop_incorrect(model, test_iterator)

    for incorrect_prediction in incorrect_predictions:
        print(incorrect_prediction)
```

```
['house', 'republicans', 'would', 'let', 'employers', 'demand', 'workers', '',
['smithsonian', 'interested', 'in', 'UNK', 'trayvon', '’', 's', 'UNK', 'for', 'n
['kim', 'kardashian', 'tries', 'to', 'escape', 'l.a.', 'in', 'UNK', 'after', 're
['the', 'onion', 'said', 'bill', 'UNK', 'wanted', 'to', 'join', 'squad', 'of', '
['the', 'UNK', 'tale', 'ends', ':', 'cap', '’', 'n', 'crunch', 'and', 'UNK', 'wa
['climate', 'change', 'researcher', 'describes', 'challenge', 'of', 'pulling', '
['american', 'flags', 'on', 'jeffrey', 'epstein', "'s", 'private', 'UNK', 'lower
['ncaa', 'UNK', 'kentucky', 'soccer', 'players', 'who', 'played', 'a', 'pickup',
['UNK', 'informs', 'george', 'h.w', '.', 'bush', 'that', 'dying', 'so', 'soon',
['lawmaker', "'s", 'war', 'hero', 'son', 'would', 'have', 'wanted', 'road', 'bil
['‘', 'UNK', 'patrol', '’', 'writers', 'defend', 'episode', 'where', 'german', '

['artifacts', 'discovered', 'buried', 'in', 'washington', 'd.c.', 'suggest', 'hu
['kim', 'jong-un', 'thrown', 'into', 'labor', 'camp', 'for', 'attempting', 'to',
['good', 'smell', 'UNK', 'new', 'yorkers', '', '', '', '', '', '', '', '', '', '
['UNK', 'hero', ':', 'man', 'stranded', 'in', 'desert', 'uses', 'last', 'of', 'h
['trump', 'UNK', 'out', 'at', 'ap', 'photographer', 'who', 'UNK', 'empty', 'chai
['infant', 'injuries', 'on', 'the', 'rise', '', '', '', '', '', '', '', '', '',
['naacp', 'issues', 'travel', 'warning', 'for', 'black', 'americans', 'visiting'
['clinton', 'aide', 'told', 'to', 'leave', 'behind', 'weak', 'volunteer', 'who',
['paul', 'ryan', 'worried', 'history', 'may', 'judge', 'him', 'UNK', 'for', 'fai
['new', 'ketchup', 'gets', 'horrifying', 'look', 'at', 'UNK', ',', 'almost', 'em
['2016', 'in', 'entertainment', '', '', '', '', '', '', '', '', '', '', '', '',
['``', 'it', '’', 's', 'an', 'honor', 'to', 'continue', 'being', 'UNK', 'over',
['if', 'you', 'come', 'into', 'my', 'pawn', 'shop', 'with', 'a', 'UNK', 'machine
['UNK', '14-year-old', 'has', 'UNK', 'crisis', '', '', '', '', '', '', '', '', '
['nsa', 'chief', ':', 'i', 'didn', '’', 't', 'lie', 'to', 'congress', 'about', '
['donald', 'trump', 'jr.', 'takes', 'son', 'on', 'hunting', 'trip', 'in', 'natio
['man', 'pours', 'all', 'his', 'UNK', 'UNK', 'into', 'UNK', ',', 'removing', 'pi
['saudi', 'arabia', 'announces', 'UNK', 'of', 'human', 'rights', 'abuses', 'to',
['bernie', 'sanders', 'refuses', 'UNK', 'abc', 'podium', 'in', 'favor', 'of', 'o
['bro', ',', 'you', "'re", 'a', 'god', 'among', 'bros', '', '', '', '', '', '',
['tired', 'but', 'UNK', 'friends', 'meet', 'at', 'bar', 'to', 'discuss', 'their'
['UNK', 'foster', 'inspires', 'teens', 'to', 'come', 'out', 'using', 'UNK', ',',
['new', 'york', 'monument', 'honors', 'victims', 'of', 'giant', 'octopus', 'atta
['spotify', 'and', 'UNK', 'team', 'up', 'to', 'create', 'UNK', 'UNK', '', '', ''
['UNK', 'fraternity', 'at', 'yale', 'university', 'accused', 'of', 'hosting', "'
['shark', 'falls', 'from', 'sky', 'onto', 'golf', 'course', '?', '-', 'yahoo', '
['restaurant', 'UNK', 'loses', 'job', 'to', '‘', 'please', 'seat', 'yourself', '
['arizona', 'unveils', 'new', 'death', 'penalty', 'plan', ':', 'bring', 'your',
['UNK', 'UNK', 'UNK', 'UNK', 'past', 'earth', 'this', 'week', 'causing', 'no', '
['shitty', 'region', 'of', 'country', 'figures', 'it', 'might', 'as', 'well', 'g
['man', 'escapes', 'death', 'after', 'crawling', 'from', 'car', 'dangling', 'on'
['in', 'response', 'to', 'the', 'onion', '’', 's', 'apology', 'to', 'UNK', 'UNK'
['epic', 'UNK', ':', 'wendy', '’', 's', 'posted', 'a', 'UNK', 'tweet', 'about',
['UNK', 'UNK', '!', 'doctors', 'around', 'the', 'world', 'share', 'images', 'of'
```

```
['UNK', 'UNK', ':', 'doctors', 'around', 'the', 'world', 'share', 'images', 'of
['it', "'s", 'an', 'UNK', 'horror', '.', 'a', '14-year-old', 'girl', 'with', 'sp
['giuliani', 'insists', 'breaking', 'the', 'law', 'not', 'a', 'crime', '', '', '
['researchers', 'find', 'yet', 'another', 'reason', 'why', 'naked', 'UNK', 'are'
['pizza', 'crust', 'saved', 'to', 'make', 'pizza', 'stock', '', '', '', '', '',
['black', 'man', 'in', 'support', 'of', 'confederate', 'flag', 'UNK', 'his', 'me
['most', 'people', 'don', ''', 't', 'want', 'to', 'see', 'friends', ''', 'vacati
['scientists', 'confess', 'to', 'sneaking', 'bob', 'dylan', 'lyrics', 'into', 't
['the', 'death', 'UNK', 'in', 'yemen', 'is', 'so', 'high', 'the', 'red', 'cross'
['UNK', 'farts', 'makes', 'you', 'live', 'longer', '-', 'and', 'doing', 'them',
['UNK', 'and', 'UNK', '', '', '', '', '', '', '', '', '', '', '', '', '', '', ''
['veteran', 'told', 'what', 'offends', 'him', '', '', '', '', '', '', '', '', ''
['republicans', 'start', 'donating', 'to', 'UNK', 'williamson', 'to', 'keep', 'h
['syria', 'conflict', 'UNK', 'as', 'bears', 'enter', 'war', '', '', '', '', '',
```

Part 6: LSTM Model [Extra-Credit, 5 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) ##



        ## 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
    def forward(self, x):
        ## YOUR CODE STARTS HERE (~4-5 lines of code) ##



        #return x
        ## YOUR CODE ENDS HERE ##
```

```
  File "<ipython-input-201-7df457497b8a>", line 41

    ^
SyntaxError: unexpected EOF while parsing
```

SEARCH STACK OVERFLOW

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 ###



### 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)}')


#Watch the model train!
TOTAL_EPOCHS = 10
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)}")


#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)}")
```

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.

0초 오후 11:56에 완료됨