```
# Licensing Information:  You are free to use or extend this project for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to The Georgia Institute of Technology, including a link to https://aritter.github.io/CS-7650/

# Attribution Information:
# This Project was developed at the Georgia Institute of Technology by Ashutosh Baheti (ashutosh.baheti@cc.gatech.edu),
# adapted from the Neural Machine Translation Project (Project 2)
# of the UC Berkeley NLP course https://cal-cs288.github.io/sp20/
```

# Project #3: Neural Chatbot

Welcome to the third and final programming assignment for CS 4650!

Neural Dialog Model are Sequence-to-Sequence (Seq2Seq) models that produce conversational response given the dialog history. State-of-the-art dialog models are trained on millions of multi-turn conversations. However, in this assignment we will narrow our scope to single turn conversations to make the problem easier.

In this assignment you will implement,

1. Seq2Seq encoder-decoder model
2. Seq2Seq model with attention mechanism
3. Greedy and Beam search decoding algorithms
4. Fine-tune and Evaluate BERT on disaster tweets

## Part 0: Setup

First, we'll import the various libraries needed for this project and define some of the utility functions to help with loading and manipulating the dataset. Since you've had experience in the previous project with splitting and tokenizing the dataset this is done for you in this project.

First import libraries required for the implementation

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import torch
from torch.jit import script, trace
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import numpy as np
import csv
import random
import re
import os
import unicodedata
import codecs
from io import open
import itertools
import math
import pickle
import statistics

from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
import tqdm
import nltk
from google.colab import files
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Then we implement some standard util functions that will be useful in the rest of the code.

```
# General util functions
def make_dir_if_not_exists(directory):
    if not os.path.exists(directory):
        logging.info("Creating new directory: {}".format(directory))
        os.makedirs(directory)

def print_list(l, K=None):
    # If K is given then only print first K
    for i, e in enumerate(l):
        if i == K:
            break
```

```
        print(e)
    print()

def remove_multiple_spaces(string):
    return re.sub(r'\s+', ' ', string).strip()

def save_in_pickle(save_object, save_file):
    with open(save_file, "wb") as pickle_out:
        pickle.dump(save_object, pickle_out)

def load_from_pickle(pickle_file):
    with open(pickle_file, "rb") as pickle_in:
        return pickle.load(pickle_in)

def save_in_txt(list_of_strings, save_file):
    with open(save_file, "w") as writer:
        for line in list_of_strings:
            line = line.strip()
            writer.write(f"{line}\n")

def load_from_txt(txt_file):
    with open(txt_file, "r") as reader:
        all_lines = list()
        for line in reader:
            line = line.strip()
            all_lines.append(line)
        return all_lines
```

Finally we will check if GPU is available and set the device accordingly.

Tip: While debugging use `CPU` to get clearer stack traces and change the runtime type to `GPU` when you are ready to train your models efficiently

```
print(torch.cuda.is_available())
if torch.cuda.is_available():
  device = torch.device("cuda")
else:
  device = torch.device("cpu")
print("Using device:", device)
```

```
True
Using device: cuda
```

## Dataset

For the dataset we will be using a small sample of single turn input and response pairs from Cornell Movie Dialog Corpus. We filter conversational pairs with sentences > 10 tokens. We have already created a sample of tokenized, lowercased single turn conversations from Cornell Movie Dialog Corpus. The preprocessed dataset sample is stored in pickle format and can be downloaded from this link. Please download the `processed_CMDC.pkl` file from the link and upload it in colab.

```
# Loading the pre-processed conversational exchanges (source-target pairs) from pickle data files
all_conversations = load_from_pickle('/content/drive/MyDrive/processed_CMDC.pkl')
# Extract 100 conversations from the end for evaluation and keep the rest for training
eval_conversations = all_conversations[-100:]
all_conversations = all_conversations[:-100]

# Logging data stats
print(f"Number of Training Conversation Pairs = {len(all_conversations)}")
print(f"Number of Evaluation Conversation Pairs = {len(eval_conversations)}")
```

```
Number of Training Conversation Pairs = 53065
Number of Evaluation Conversation Pairs = 100
```

Let's print a couple of conversations to check if they are loaded properly.

```
print_list(all_conversations, 5)
```

```
('there .', 'where ?')
('you have my word . as a gentleman', 'you re sweet .')
('hi .', 'looks like things worked out tonight huh ?')
('have fun tonight ?', 'tons')
('well no . . .', 'then that s all you had to say .')
```

## Vocabulary

The words in the sentences need to be converted into integer tokens so that the neural model can operate on them. For this purpose, we will create a vocabulary which will convert the input strings into model recognizable integer tokens.

```
pad_word = "<pad>"
bos_word = "<s>"
```

```
eos_word = "</s>"
unk_word = "<unk>"
pad_id = 0
bos_id = 1
eos_id = 2
unk_id = 3

def normalize_sentence(s):
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    s = re.sub(r"\s+", r" ", s).strip()
    return s

class Vocabulary:
    def __init__(self):
        self.word_to_id = {pad_word: pad_id, bos_word: bos_id, eos_word:eos_id, unk_word: unk_id}
        self.word_count = {}
        self.id_to_word = {pad_id: pad_word, bos_id: bos_word, eos_id: eos_word, unk_id: unk_word}
        self.num_words = 4

    def get_ids_from_sentence(self, sentence):
        sentence = normalize_sentence(sentence)
        sent_ids = [bos_id] + [self.word_to_id[word] if word in self.word_to_id \
                               else unk_id for word in sentence.split()] + \
                               [eos_id]
        return sent_ids

    def tokenized_sentence(self, sentence):
        sent_ids = self.get_ids_from_sentence(sentence)
        return [self.id_to_word[word_id] for word_id in sent_ids]

    def decode_sentence_from_ids(self, sent_ids):
        words = list()
        for i, word_id in enumerate(sent_ids):
            if word_id in [bos_id, eos_id, pad_id]:
                # Skip these words
                continue
            else:
                words.append(self.id_to_word[word_id])
        return ' '.join(words)

    def add_words_from_sentence(self, sentence):
        sentence = normalize_sentence(sentence)
        for word in sentence.split():
            if word not in self.word_to_id:
                # add this word to the vocabulary
                self.word_to_id[word] = self.num_words
                self.id_to_word[self.num_words] = word
                self.word_count[word] = 1
                self.num_words += 1
            else:
                # update the word count
                self.word_count[word] += 1

vocab = Vocabulary()
for src, tgt in all_conversations:
    vocab.add_words_from_sentence(src)
    vocab.add_words_from_sentence(tgt)
print(f"Total words in the vocabulary = {vocab.num_words}")
```

```
Total words in the vocabulary = 7727
```

Let's print the top 30 vocab words:

```
print_list(sorted(vocab.word_count.items(), key=lambda item: item[1], reverse=True), 30)
```

```
('.', 84255)
('?', 36822)
('you', 25093)
('i', 18946)
('what', 10765)
('s', 10089)
('it', 9668)
('!', 8872)
('the', 8011)
('t', 7411)
('to', 6929)
('a', 6582)
('that', 5992)
('no', 4931)
('me', 4839)
('do', 4745)
('is', 4434)
('don', 3577)
('are', 3503)
('he', 3413)
('yes', 3384)
('m', 3382)
('not', 3252)
('we', 3252)
('know', 3171)
('re', 2965)
```

```
('your', 2809)
('this', 2726)
('yeah', 2708)
('in', 2678)
```

We can also print a couple of sentences to verify that the vocabulary is working as intended, as well as ensure our encoding/decoding process works as expected.

```
for src, tgt in all_conversations[:3]:
    sentence = tgt
    word_tokens = vocab.tokenized_sentence(sentence)
    # Automatically adds bos_id and eos_id before and after sentence ids respectively
    word_ids = vocab.get_ids_from_sentence(sentence)
    print(sentence)
    print(word_tokens)
    print(word_ids)
    print(vocab.decode_sentence_from_ids(word_ids))
    print()

word = "the"
word_id = vocab.word_to_id[word]
print(f"Word = {word}")
print(f"Word ID = {word_id}")
print(f"Word decoded from ID = {vocab.decode_sentence_from_ids([word_id])}")
```

```
where ?
['<s>', 'where', '?', '</s>']
[1, 6, 7, 2]
where ?

you re sweet .
['<s>', 'you', 're', 'sweet', '.', '</s>']
[1, 8, 15, 16, 5, 2]
you re sweet .

looks like things worked out tonight huh ?
['<s>', 'looks', 'like', 'things', 'worked', 'out', 'tonight', 'huh', '?', '</s>']
[1, 18, 19, 20, 21, 22, 23, 24, 7, 2]
looks like things worked out tonight huh ?

Word = the
Word ID = 47
Word decoded from ID = the
```

## Part 1: Dataset Preparation (5 points)

We will use built-in dataset utilities, `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`, to get batched data readily useful for training like what you saw in Project 1.

Most of the dataset has been filled out for you, however the `collate_fn` needs to be finished.

```
class SingleTurnMovieDialog_dataset(Dataset):
    """Single-Turn version of Cornell Movie Dialog Cropus dataset."""

    def __init__(self, conversations, vocab, device):
        """
        Args:
            conversations: list of tuple (src_string, tgt_string)
                          - src_string: String of the source sentence
                          - tgt_string: String of the target sentence
            vocab: Vocabulary object that contains the mapping of
                    words to indices
            device: cpu or cuda
        """
        self.conversations = conversations
        self.vocab = vocab
        self.device = device

        def encode(src, tgt):
            src_ids = self.vocab.get_ids_from_sentence(src)
            tgt_ids = self.vocab.get_ids_from_sentence(tgt)
            return (src_ids, tgt_ids)

        # We will pre-tokenize the conversations and save in id lists for later use
        self.tokenized_conversations = [encode(src, tgt) for src, tgt in self.conversations]

    def __len__(self):
        return len(self.conversations)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        return {"conv_ids":self.tokenized_conversations[idx], "conv":self.conversations[idx]}

def collate_fn(data):
    """Creates mini-batch tensors from the list of tuples (src_seq, trg_seq).
```

```
    We should build a custom collate_fn rather than using default collate_fn,
    because merging sequences (including padding) is not supported in default.
    Seqeuences are padded to the maximum length of mini-batch sequences (dynamic padding).
    Args:
        data: list of dicts {"conv_ids":(src_ids, tgt_ids), "conv":(src_str, trg_str)}.
            - src_ids: list of src piece ids; variable length.
            - tgt_ids: list of tgt piece ids; variable length.
            - src_str: String of src
            - tgt_str: String of tgt
    Returns: dict { "conv_ids":     (src_ids, tgt_ids),
                    "conv":         (src_str, tgt_str),
                    "conv_tensors": (src_seqs, tgt_seqs)}
            src_seqs: torch tensor of shape (src_padded_length, batch_size).
            tgt_seqs: torch tensor of shape (tgt_padded_length, batch_size).
            src_padded_length = length of the longest src sequence from src_ids
            tgt_padded_length = length of the longest tgt sequence from tgt_ids
    """
    # Sort conv_ids based on decreasing order of the src_lengths.
    # This is required for efficient GPU computations.
    src_ids = [torch.LongTensor(e["conv_ids"][0]) for e in data]
    tgt_ids = [torch.LongTensor(e["conv_ids"][1]) for e in data]
    src_str = [e["conv"][0] for e in data]
    tgt_str = [e["conv"][1] for e in data]
    data = list(zip(src_ids, tgt_ids, src_str, tgt_str))
    data.sort(key=lambda x: len(x[0]), reverse=True)
    src_ids, tgt_ids, src_str, tgt_str = zip(*data)

    # Pad the src_ids and tgt_ids using token pad_id to create src_seqs and tgt_seqs

    # HINT: You can use the nn.utils.rnn.pad_sequence utility
    # function to combine a list of variable-length sequences with padding.

    # YOUR CODE HERE

    src_seqs = nn.utils.rnn.pad_sequence(src_ids, padding_value = pad_id)
    tgt_seqs = nn.utils.rnn.pad_sequence(tgt_ids, padding_value = pad_id)

    return {"conv_ids":(src_ids, tgt_ids), "conv":(src_str, tgt_str), "conv_tensors":(src_seqs.to(device), tgt_seqs.to(device))}
```

```
# Create the DataLoader for all_conversations
dataset = SingleTurnMovieDialog_dataset(all_conversations, vocab, device)

batch_size = 5

data_loader = DataLoader(dataset=dataset, batch_size=batch_size,
                               shuffle=True, collate_fn=collate_fn)
```

Let's test a batch of data to make sure everything is working as intended

*HINT*: If you've padded the targets correctly, each column should start with the beginning of sequence ID (i.e. 1) and should follow the end of sequence ID with some number of the pad ID (i.e. 0) if the sequence in that column is shorter than the max in the minibatch.

```
# Test one batch of training data
first_batch = next(iter(data_loader))
print(f"Testing first training batch of size {len(first_batch['conv'][0])}")
print(f"List of source strings:")
print_list(first_batch["conv"][0])
print(f"Tokenized source ids:")
print_list(first_batch["conv_ids"][0])
print(f"Padded source ids as tensor (shape {first_batch['conv_tensors'][0].size()}):")
print(first_batch["conv_tensors"][0])
```

```
Testing first training batch of size 5
List of source strings:
i don t know i think . . .
but i get a red light .
i will do no such thing .
pardon ?
okay

Tokenized source ids:
tensor([  1,  54, 198, 103,  97,  54,  66,   5,   5,   5,   2])
tensor([   1,   36,   54,  142,   13,  667, 1492,    5,    2])
tensor([  1,  54, 728,  41,  28, 590, 169,   5,   2])
tensor([   1, 1274,    7,    2])
tensor([ 1, 53,  2])

Padded source ids as tensor (shape torch.Size([11, 5])):
tensor([[   1,    1,    1,    1,    1],
        [  54,   36,   54, 1274,   53],
        [ 198,   54,  728,    7,    2],
        [ 103,  142,   41,    2,    0],
        [  97,   13,   28,    0,    0],
        [  54,  667,  590,    0,    0],
        [  66, 1492,  169,    0,    0],
        [   5,    5,    5,    0,    0],
        [   5,    2,    2,    0,    0],
        [   5,    0,    0,    0,    0],
        [   2,    0,    0,    0,    0]], device='cuda:0')
```

# Part 2: Baseline Seq2Seq model (25 points)

In this section you will initialize the layers needed for your Seq2Seq model, define the encode and decode functions of your model, and define a loss function to handle the padded tokens when training your model.

With the training `Dataset` and `DataLoader` ready, we can implement our Seq2Seq baseline model.

The model will consist of

1. Shared embedding layer between encoder and decoder that converts the input sequence of word ids to dense embedding representations
2. Bidirectional GRU encoder that encodes the embedded source sequence into hidden representation
3. Unidirectional GRU decoder that predicts target sequence using final encoder hidden representation

```
class Seq2seqBaseline(nn.Module):
    def __init__(self, vocab, emb_dim = 300, hidden_dim = 300, num_layers = 2, dropout=0.1):
        super().__init__()

        # Initialize your model's parameters here. To get started, we suggest
        # setting all embedding and hidden dimensions to 300, using encoder and
        # decoder GRUs with 2 layers each, and using a dropout rate of 0.1.

        # HINT: To create a bidirectional GRU, you don't need to create two GRU
        # networks, instead use the bidirectional flag when initializing the layer.

        self.num_words = num_words = vocab.num_words
        self.emb_dim = emb_dim
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        # YOUR CODE HERE
        self.embedding = nn.Embedding(num_words, emb_dim)
        self.encoder = nn.GRU(emb_dim, hidden_dim, num_layers, dropout = dropout, bidirectional = True)
        self.decoder = nn.GRU(emb_dim, hidden_dim, num_layers, dropout = dropout, bidirectional = False)
        self.fc = nn.Linear(hidden_dim, num_words)

    def encode(self, source):
        """Encode the source batch using a bidirectional GRU encoder.

        Args:
            source: An integer tensor with shape (max_src_sequence_length,
                batch_size) containing subword indices for the source sentences.

        Returns:
            A tuple with three elements:
                encoder_output: The output hidden representation of the encoder
                    with shape (max_src_sequence_length, batch_size, hidden_size).
                    Can be obtained by adding the hidden representations of both
                    directions of the encoder bidirectional GRU.
                encoder_mask: A boolean tensor with shape (max_src_sequence_length,
                    batch_size) indicating which encoder outputs correspond to padding
                    tokens. Its elements should be True at positions corresponding to
                    padding tokens and False elsewhere.
                encoder_hidden: The final hidden states of the bidirectional GRU
                    (after a suitable projection) that will be used to initialize
                    the decoder. This should be a tensor h_n with shape
                    (num_layers, batch_size, hidden_size). Note that the hidden
                    state returned by the bi-GRU cannot be used directly. Its
                    initial dimension is twice the required size because it
                    contains state from two directions.

        The first two return values are not required for the baseline model and will
        only be used later in the attention model. If desired, they can be replaced
        with None for the initial implementation.
        """

        # Implementation tip: consider using packed sequences to more easily work
        # with the variable-length sequences represented by the source tensor.
        # See https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.PackedSequence.

        # https://stackoverflow.com/questions/51030782/why-do-we-pack-the-sequences-in-pytorch

        # HINT: there are many simple ways to combine the forward
        # and backward portions of the final hidden state, e.g. addition, averaging,
        # or a linear transformation of the appropriate size. Any of these
        # should let you reach the required performance.

        # Compute a tensor containing the length of each source sequence.
        source_lengths = torch.sum(source != pad_id, axis=0).cpu()

        # YOUR CODE HERE
        embd = self.embedding(source).to(device)
        packed_source = torch.nn.utils.rnn.pack_padded_sequence(embd, source_lengths)
        packed_encoder_output, hn = self.encoder(packed_source)
        encoder_output, _ = nn.utils.rnn.pad_packed_sequence(packed_encoder_output)
        new_encoder_output = torch.zeros(encoder_output.size()[0], encoder_output.size()[1], self.hidden_dim).to(device)
        for i in range(self.hidden_dim):
          new_encoder_output[:, :, i] = encoder_output[:,:, 2*i] + encoder_output[:, :, 2*i+1]
        encoder_output = new_encoder_output
```

```
        # encoder_hidden = encoder_hidden.view(self.num_layers, 2, -1, self.hidden_dim)
        # encoder_1 = (encoder_hidden[0, :, :] + encoder_hidden[1, :, :]).unsqueeze(0)
        # encoder_2 = (encoder_hidden[2, :, :] + encoder_hidden[3, :, :]).unsqueeze(0)
        # encoder_hidden = torch.cat((encoder_1, encoder_2), 0)
        encoder_hidden = torch.zeros(self.num_layers, hn.size()[1], self.hidden_dim).to(device)
        for i in range(self.num_layers):
          encoder_hidden[i,:,:] = hn[2*i, :, :] + hn[2*i+1, :, :]
        # print(encoder_hidden.shape)
        # encoder_output = encoder_output[:, :, :self.hidden_dim] + encoder_output[:, :, self.hidden_dim:]
        encoder_mask = (source == pad_id)
        return encoder_output, encoder_mask, encoder_hidden

    def decode(self, decoder_input, last_hidden, encoder_output, encoder_mask):
        """Run the decoder GRU for one decoding step from the last hidden state.

        The third and fourth arguments are not used in the baseline model, but are
        included for compatibility with the attention model in the next section.

        Args:
            decoder_input: An integer tensor with shape (1, batch_size) containing
                the subword indices for the current decoder input.
            last_hidden: A tensor h_{t-1} representing the last hidden
                state of the decoder, has the shape (num_layers, batch_size,
                hidden_size). For the first decoding step the last_hidden will be
                encoder's final hidden representation.
            encoder_output: The output of the encoder with shape
                (max_src_sequence_length, batch_size, hidden_size).
            encoder_mask: The output mask from the encoder with shape
                (max_src_sequence_length, batch_size). Encoder outputs at positions
                with a True value correspond to padding tokens and should be ignored.

        Returns:
            A tuple with three elements:
                logits: A tensor with shape (batch_size,
                    vocab_size) containing unnormalized scores for the next-word
                    predictions at each position.
                decoder_hidden: tensor h_n with the same shape as last_hidden
                    representing the updated decoder state after processing the
                    decoder input.
                attention_weights: This will be implemented later in the attention
                    model, but in order to maintain compatible type signatures, we also
                    include it here. This can be None or any other placeholder value.
        """

        # These arguments are not used in the baseline model.
        del encoder_output
        del encoder_mask
        # YOUR CODE HERE
        embd = self.embedding(decoder_input).to(device)
        output, decoder_hidden = self.decoder(embd, last_hidden)
        logits = self.fc(output.squeeze())
        return logits, decoder_hidden, None


    def compute_loss(self, source, target):
        """Run the model on the source and compute the loss on the target.
           The loss for this project should use teacher forcing, where the
           output of the model is used only to compute loss and not passed
           back in to get the next predicted token.

        Args:
            source: An integer tensor with shape (max_source_sequence_length,
                batch_size) containing subword indices for the source sentences.
            target: An integer tensor with shape (max_target_sequence_length,
                batch_size) containing subword indices for the target sentences.

        Returns:
            A scalar float tensor representing cross-entropy loss on the current batch
            divided by the number of target tokens in the batch.
            Many of the target tokens will be pad tokens. You should mask the loss
            from these tokens using appropriate mask on the target tokens loss.
        """

        # Hint: don't feed the target tensor directly to the decoder.
        # To see why, note that for a target sequence like <s> A B C </s>, you would
        # want to run the decoder on the prefix <s> A B C and have it predict the
        # suffix A B C </s>.

        # You may run self.encode() on the source only once and decode the target
        # one step at a time.

        total_loss = 0
        # YOUR CODE HERE
        batch_size = source.shape[1]
        encoder_output, encoder_mask, encoder_hidden = self.encode(source)
        decoder_hidden = encoder_hidden
        decoder_input = target[0, :].unsqueeze(0)

        # decoder_input = target[0].unsqueeze(0)
        for i in range(target.shape[0]-1):
```

```
            # print(target[i].shape)
            logits, decoder_hidden, _ = self.decode(decoder_input, decoder_hidden, encoder_output, encoder_mask)
            # print(decoder_hidden.shape)
            # print(decoder_input.shape)
            loss = nn.functional.cross_entropy(logits, target[i+1])
            mask = (target[i+1] != pad_id)
            # print(mask)
            total_loss += (loss*mask).sum()
            decoder_input = target[i+1, :].unsqueeze(0)

          return total_loss / torch.sum(target != pad_id).float()

```

## Training

We provide a training loop for training the model. You are welcome to modify the training loop by adjusting the learning rate or changing optmization settings.

**Important:** During our testing we found that training the encoder and decoder with different learning rates is crucial for getting good performance over the small dialog corpus. Specifically, the decoder parameter learning rate should be 5 times the encoder parameter learning rate. Hence, add the encoder parameter variable names in the `encoder_parameter_names` as a list. For example, if encoder is using `self.embedding_layer` and `self.encoder_gru` layer then the `encoder_parameter_names` should be `['embedding_layer', 'encoder_gru']`

```
from tqdm.notebook import trange, tqdm
def train(model, data_loader, num_epochs, model_file):
    """Train the model for given number of epochs and save the trained model in
    the final model_file.
    """

    # feel free to edit these values!
    decoder_learning_ratio = 5.0
    learning_rate = 0.0001

    encoder_parameter_names = ['embedding', 'encoder']

    encoder_named_params = list(filter(lambda kv: any(key in kv[0] for key in encoder_parameter_names), model.named_parameters()))
    decoder_named_params = list(filter(lambda kv: not any(key in kv[0] for key in encoder_parameter_names), model.named_parameters()))
    encoder_params = [e[1] for e in encoder_named_params]
    decoder_params = [e[1] for e in decoder_named_params]
    optimizer = torch.optim.AdamW([{'params': encoder_params},
                {'params': decoder_params, 'lr': learning_rate * decoder_learning_ratio}], lr=learning_rate)

    clip = 50.0
    for epoch in trange(num_epochs, desc="training", unit="epoch"):
        # print(f"Total training instances = {len(train_dataset)}")
        # print(f"train_data_loader = {len(train_data_loader)} {1180 > len(train_data_loader)/20}")
        with tqdm(
                data_loader,
                desc="epoch {}".format(epoch + 1),
                unit="batch",
                total=len(data_loader)) as batch_iterator:
            model.train()
            total_loss = 0.0
            for i, batch_data in enumerate(batch_iterator, start=1):
                source, target = batch_data["conv_tensors"]
                optimizer.zero_grad()
                loss = model.compute_loss(source, target)
                total_loss += loss.item()
                loss.backward()
                # Gradient clipping before taking the step
                _ = nn.utils.clip_grad_norm_(model.parameters(), clip)
                optimizer.step()

                batch_iterator.set_postfix(mean_loss=total_loss / i, current_loss=loss.item())
    # Save the model after training
    torch.save(model.state_dict(), model_file)
```

We can now train the baseline model. This should take about 5 minutes with a GPU and will take >40 minutes on just the CPU, so we highly recommend using a Colab Pro account.

A correct implementation should get a average train loss of < 3.00, however be aware, as this may not be the best sign your model will behave as desired. While the loss will give you some idea concerning the correctness of your implementation, you should also "talk" with it to confirm. Please check Piazza (specifically, the pinned post on Part 2) to see an example of a correct implementation.

The code will automatically save and download the model at the end of training, that way you won't have to retrain if you come back to the notebook later.

```
# You are welcome to adjust these parameters based on your model implementation.
num_epochs = 6
batch_size = 64
# Reloading the data_loader to increase batch_size
data_loader = DataLoader(dataset=dataset, batch_size=batch_size,
                               shuffle=True, collate_fn=collate_fn)

```

```
baseline_model = Seq2seqBaseline(vocab).to(device)
train(baseline_model, data_loader, num_epochs, "baseline_model.pt")
# Download the trained model to local for future use
files.download('baseline_model.pt')
```

```
training: 100%    6/6 [04:17<00:00, 43.08s/epoch]
epoch 1: 100%    830/830 [00:42<00:00, 22.03batch/s, current_loss=1.79, mean_loss=2.99]
epoch 2: 100%    830/830 [00:42<00:00, 21.52batch/s, current_loss=2.43, mean_loss=2.58]
epoch 3: 100%    830/830 [00:43<00:00, 21.76batch/s, current_loss=2.79, mean_loss=2.44]
epoch 4: 100%    830/830 [00:42<00:00, 21.20batch/s, current_loss=2.79, mean_loss=2.33]
epoch 5: 100%    830/830 [00:43<00:00, 19.78batch/s, current_loss=2.54, mean_loss=2.23]
epoch 6: 100%    830/830 [00:43<00:00, 15.58batch/s, current_loss=2.04, mean_loss=2.14]
```

```
# Reload the model from the model file.
# Useful when you have already trained and saved the model
baseline_model = Seq2seqBaseline(vocab).to(device)
baseline_model.load_state_dict(torch.load("baseline_model.pt", map_location=device))
```

```
<All keys matched successfully>
```

## Part 3: Greedy Search (10 points)

For evaluation, we also need to be able to generate entire strings from the model. We'll first define a greedy inference procedure here. Later on, we'll implement beam search. *Hint:* Use the **normalize_sentence** and **vocab.get_ids_from_sentence** functions to prepare your input.

```
def predict_greedy(model, sentence, max_length=100):
    """Make predictions for the given input using greedy inference.

    Args:
        model: A sequence-to-sequence model.
        sentence: A input string.
        max_length: The maximum length at which to truncate outputs in order to
            avoid non-terminating inference.

    Returns:
        Model's predicted greedy response for the input, represented as string.

    HINT: Make sure to terminate your models prediction when it outputs the end of
    sequence ID, even if the models reponse hasn't reached the max length.
    """

    # You should make only one call to model.encode() at the start of the function,
    # and make only one call to model.decode() per inference step.
    model.eval()

    # YOUR CODE HERE
    model = model.to(device)
    input = torch.tensor(vocab.get_ids_from_sentence(normalize_sentence(sentence))).unsqueeze(1).to(device)
    encode_output, encode_mask, encode_hidden = model.encode(input)
    decode_hidden = encode_hidden
    output = []
    decoder_input = torch.tensor([bos_id]).unsqueeze(0).to(device)
    for i in range(max_length):
      logits, decode_hidden, _ = model.decode(decoder_input, decode_hidden, encode_output, encode_mask)
      predicted_id = int(logits.argmax())
      output.append(predicted_id)
      decoder_input = torch.tensor([predicted_id]).unsqueeze(0).to(device)
    return vocab.decode_sentence_from_ids(output)

```

Let's chat interactively with our trained baseline Seq2Seq dialog model and save the generated conversations for submission (please make sure to keep the conversations in your submission "PG-13"). We will reuse the conversational inputs while testing Seq2Seq + Attention model.

The output of your model isn't likely to be very colorful given the simplicity of the dataset we're working on. Instead, you should expect responses that are generally grammatically correct and do not degrade (i.e. your model keeps repeating the same word(s) over and over).

**IMPORTANT: FOR YOUR FINAL SUBMISSION TO GRADESCOPE, PLEASE "TALK" WITH YOUR CHATBOT IN THE CELLS BELOW FOR ABOUT FIVE TURNS AND MAKE SURE THE RESPONSES ARE VISIBLE IN YOUR UPLOADED NOTEBOOK.**

Note: enter "q" or "quit" to end the interactive chat.

```
def chat_with_model(model, mode="greedy"):
    if mode == "beam":
        predict_f = predict_beam
    else:
        predict_f = predict_greedy
    chat_log = list()
    input_sentence = ''
    while(1):
        # Get input sentence
```

```
        input_sentence = input('Input > ')
        # Check if it is quit case
        if input_sentence == 'q' or input_sentence == 'quit': break

        generation = predict_f(model, input_sentence)
        if mode == "beam":
            generation = generation[0]
        print('Greedy Response:', generation)
        print()
        chat_log.append((input_sentence, generation))
    return chat_log
```

```
baseline_chat = chat_with_model(baseline_model)
```

```
Input > how are you?
Greedy Response: fine .

Input > what do you want for lunch?
Greedy Response: i m going to go .

Input > did you like our time together?
Greedy Response: i don t know .

Input > who is your favorite actor?
Greedy Response: i don t know .

Input > what do you want to eat for dinner?
Greedy Response: i m going to go .

Input > What's your favorite movie?
Greedy Response: yes .

Input > thanks.
Greedy Response: you re a good man .

Input > did you like our time together?
Greedy Response: i don t know .

Input > why not?
Greedy Response: i don t know .

Input > q
```

## Part 4: Seq2Seq + Attention Model (15 points)

Next, we extend the baseline model to include an attention mechanism in the decoder. With attention mechanism, the model doesn't need to encode the input into a fixed dimensional hidden representation. Rather, it creates a new context vector for each turn that is a weighted sum of encoder hidden representation.

Your implementation can use any attention mechanism to get weight distribution over the source words. One simple way to include attention in decoder goes as follows (reminder: the decoder processed one token at a time),

1. Process the current decoder_input through embedding layer and decoder GRU layer.
2. Use the current decoder token representation, $d$ of shape $(1 * b * h)$ and encoder representation, $e_1, \ldots, e_n$ of shape $(n * b * h)$, where $n$ is max_src_length after padding) to compute attention score matrix of shape $(b * n)$. There are multiple options to compute this score matrix. A few of such options are available in [the table provided in this blog](). Please leave a comment in your code with the name of the method you choose to implement
3. Normalize the attention scores $(b * n)$ so that they sum up to $1.0$ by taking a `softmax` over the second dimention.

After computing the normalized attention distribution, take a weighted sum of the encoder outputs to obtain the attention context $c = \sum_i w_i e_i$, and add this to the decoder output $d$ to obtain the final representation to be passed to the vocabulary projection layer (you may need another linear layer to make the sizes match before adding $c$ and $d$).

```
class Seq2seqAttention(Seq2seqBaseline):
    def __init__(self, vocab):
        super().__init__(vocab)

        # Initialize any additional parameters needed for this model that are not
        # already included in the baseline model.

        # YOUR CODE HERE
        self.attention = nn.Linear(self.hidden_dim*2, self.hidden_dim)


    def decode(self, decoder_input, last_hidden, encoder_output, encoder_mask):
        """Run the decoder GRU for one decoding step from the last hidden state.

        The third and fourth arguments are not used in the baseline model, but are
        included for compatibility with the attention model in the next section.

        Args:
            decoder_input: An integer tensor with shape (1, batch_size) containing
                the subword indices for the current decoder input.
            last_hidden: A pair of tensors h_{t-1} representing the last hidden
                state of the decoder, each with shape (num_layers, batch_size,
                hidden_size). For the first decoding step the last_hidden will be
```

```
            encoder's final hidden representation.
        encoder_output: The output of the encoder with shape
            (max_src_sequence_length, batch_size, hidden_size).
        encoder_mask: The output mask from the encoder with shape
            (max_src_sequence_length, batch_size). Encoder outputs at positions
            with a True value correspond to padding tokens and should be ignored.

    Returns:
        A tuple with three elements:
            logits: A tensor with shape (batch_size, vocab_size)
                containing unnormalized scores for the next-word
                predictions at each position.
            decoder_hidden: tensor h_n with the same shape as last_hidden
                representing the updated decoder state after processing the
                decoder input.
            attention_weights: A tensor with shape (batch_size, max_src_sequence_length)
                representing the normalized attention weights. This should sum to 1
                along the last dimension.
    """

    # YOUR CODE HERE
    batch_size = decoder_input.size(1)
    embd = self.embedding(decoder_input)
    decoder_out, decoder_hidden = self.decoder(embd, last_hidden)
    top_hidden = decoder_hidden[-1].unsqueeze(1)
    encoder_output = encoder_output.permute(1,0,2)

    # use dot product scores
    scores = torch.bmm(top_hidden, encoder_output.transpose(1,2))
    attn_weights = nn.functional.softmax(scores, dim=1)
    context = torch.bmm(attn_weights, encoder_output).squeeze(1)
    decoder_output = decoder_out.squeeze(0)
    joined = torch.cat((decoder_output, context), dim=1)
    combined = self.attention(joined)
    logits = self.fc(combined)
    return logits, decoder_hidden, attn_weights
```

## Training

We can now train the attention model.

A correct implementation should also get an average train loss of < 3.00, however you should still check your models output to confirm you've implemented the attention mechanism correctly.

The code will automatically save and download the model at the end of training.

It may happen that the baseline model achieves a worse loss than attention model. This is because our dataset is very small and the attention model may be over parameterized for our toy dataset. Regardless, we would consider this as acceptable submission if the attention model generated responses look comparable to the baseline model.

```
# You are welcome to adjust these parameters based on your model implementation.
num_epochs = 8
batch_size = 64
data_loader = DataLoader(dataset=dataset, batch_size=batch_size,
                               shuffle=True, collate_fn=collate_fn)

attention_model = Seq2seqAttention(vocab).to(device)
train(attention_model, data_loader, num_epochs, "attention_model.pt")
# Download the trained model to local for future use
files.download('attention_model.pt')
```

training: 100% 8/8 [13:04<00:00, 97.24s/epoch]

epoch 1: 100% 830/830 [01:40<00:00, 7.10batch/s, current_loss=2.39, mean_loss=2.92]

epoch 2: 100% 830/830 [01:39<00:00, 8.67batch/s, current_loss=2.76, mean_loss=2.53]

epoch 3: 100% 830/830 [01:38<00:00, 8.99batch/s, current_loss=2.37, mean_loss=2.36]

epoch 4: 100% 830/830 [01:37<00:00, 9.60batch/s, current_loss=2.72, mean_loss=2.2]

epoch 5: 100% 830/830 [01:37<00:00, 6.99batch/s, current_loss=1.77, mean_loss=2.05]

epoch 6: 100% 830/830 [01:38<00:00, 8.86batch/s, current_loss=1.81, mean_loss=1.93]

epoch 7: 100% 830/830 [01:36<00:00, 9.31batch/s, current_loss=2.38, mean_loss=1.82]

epoch 8: 100% 830/830 [01:36<00:00, 9.62batch/s, current_loss=1.58, mean_loss=1.72]

```
# Reload the model from the model file.
# Useful when you have already trained and saved the model
attention_model = Seq2seqAttention(vocab).to(device)
attention_model.load_state_dict(torch.load("attention_model.pt", map_location=device))
```

```
<All keys matched successfully>
```

Let's test the attention model on the some sample inputs.

```
def test_conversations_with_model(model, conversational_inputs = None, include_beam = False):
    # Some predefined conversational inputs.
    # You may append more inputs at the end of the list, if you want to.
    basic_conversational_inputs = [
                                "hello.",
                                "please share you bank account number with me",
                                "i have never met someone more annoying that you",
                                "i like pizza. what do you like?",
                                "give me coffee, or i'll hate you",
                                "i'm so bored. give some suggestions",
                                "stop running or you'll fall hard",
                                "what is your favorite sport?",
                                "do you believe in a miracle?",
                                "which sport team do you like?"
    ]
    if not conversational_inputs:
        conversational_inputs = basic_conversational_inputs
    for input in conversational_inputs:
        print(f"Input > {input}")
        generation = predict_greedy(model, input)
        print('Greedy Response:', generation)
        if include_beam:
            # Also print the beam search responses from models
            generations = predict_beam(model, input)
            print('Beam Responses:')
            print_list(generations)
        print()
```

```
baseline_chat_inputs = [inp for inp, gen in baseline_chat]
attention_chat = test_conversations_with_model(attention_model, baseline_chat_inputs)
```

```
Input > how are you?
Greedy Response: i m fine .

Input > what do you want for lunch?
Greedy Response: i want to go to cash .

Input > did you like our time together?
Greedy Response: no .

Input > who is your favorite actor?
Greedy Response: my employer .

Input > what do you want to eat for dinner?
Greedy Response: i can t take a walk .

Input > What's your favorite movie?
Greedy Response: yeah .

Input > thanks.
Greedy Response: i m glad you d like me .

Input > did you like our time together?
Greedy Response: no .

Input > why not?
Greedy Response: i don t know .
```

## Part 5: Automatic Evaluation (5 points)

Automatic evaluation of chatbots is an active research area. For this assignment we are going to use 3 very simple evaluation metrics.

1. Average Length of the Responses
2. Distinct1 = proportion of unique unigrams / total unigrams
3. Distinct2 = proportion of unique bigrams / total bigrams

Length in this case refers to the number of tokens in the models response. You will evaluate your baseline and attention models by running the cells below.

```
# Evaluate diversity of the models
def evaluate_diversity(model, mode="greedy"):
    """Evaluates the model's greedy or beam responses on eval_conversations

    Args:
        model: A sequence-to-sequence model.
        mode: "greedy" or "beam"

    Returns: avg_length, distinct1, distinct2
        avg_length: average length of the model responses
        distinct1: proportion of unique unigrams / total unigrams
        distinct2: proportion of unique bigrams / total bigrams
    """
    if mode == "beam":
        predict_f = predict_beam
```

```
    else:
        predict_f = predict_greedy
    generations = list()
    for src, tgt in eval_conversations:
        generation = predict_f(model, src)
        if mode == "beam":
            generation = generation[0]
        generations.append(generation)
    # Calculate average length, distinct unigrams and bigrams from generations
    avg_length, distinct1, distinct2 = 0, 0, 0

    # YOUR CODE HERE
    total_length = sum(len(g) for g in generations)
    num_g = len(generations)
    avg_length = total_length/num_g

    all_tokens = [token for generation in generations for token in vocab.tokenized_sentence(generation)]
    num_tokens = len(all_tokens)
    distinct_tokens = set(all_tokens)
    num_distinct_tokens = len(distinct_tokens)

    all_bigrams = list(nltk.bigrams(all_tokens))
    num_bigrams = len(all_bigrams)
    distinct_bigrams = set(all_bigrams)
    num_distinct_bigrams = len(distinct_bigrams)

    distinct1 = num_distinct_tokens / num_tokens
    distinct2 = num_distinct_bigrams / num_bigrams


    return avg_length, distinct1, distinct2
```

```
print(f"Baseline Model evaluation:")
avg_length, distinct1, distinct2 = evaluate_diversity(baseline_model)
print(f"Greedy decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
print(f"Attention Model evaluation:")
avg_length, distinct1, distinct2 = evaluate_diversity(attention_model)
print(f"Greedy decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
```

```
Baseline Model evaluation:
Greedy decoding:
Avg Response Length = 11.56
Distinct1 = 0.0777027027027027
Distinct2 = 0.1404399323181049
Attention Model evaluation:
Greedy decoding:
Avg Response Length = 14.89
Distinct1 = 0.13574660633484162
Distinct2 = 0.26132930513595165
```

## Part 6: BERT Finetuning (5 points)

Introduced in the paper BERT" Pre-training of Deep Bidirectional Transformers for Language Understanding" (https://arxiv.org/pdf/1810.04805.pdf), the pretrained transformer model BERT is heavily used within NLP research and engineering. This section will walk you through the use of the popular Huggingface Transformers library so that you can utilize it for your final projects and any research you may pursue.

The HuggingFace documentation can be found here: https://huggingface.co/transformers/. You will need to refer to the documentation frequently through this section.

The Dataset preparation and Model Helpers subsections contain utility code to setup this portion of the project. **Your first task begins in the second cell in the Model Setup subsection** where you will download the pretrained model. After this, you will add a classification head to the model so that we can classify disaster tweets.

### Dataset Preparation

Kaggle is a popular machine learning website that runs competitions for machine learning datasets. We will be using the Kaggle dataset "Natural Language Processing with Disaster Tweets" for this assignment. This dataset contains tweets that were sent in response to an actual disaster or that merely contain language similar to that used to describe a disaster. The goal of this challenge, and of this section, is to train a model that can classify tweets as either disaster related or non disaster related. For the following section, we are using the data from https://www.kaggle.com/c/nlp-getting-started/overview. Feel free to create a Kaggle account and look at the competition in more depth; for this project, however, we will download the training data directly from the class repository.

```
import pandas as pd
import numpy as np
import sys
```

```
from functools import partial
import time
```

```
#load the data into a pandas dataframe
!wget https://raw.githubusercontent.com/cocoxu/CS4650_projects_spring2023/master/p3_bert_train.csv
full_df = pd.read_csv('p3_bert_train.csv', header=0)
```

```
--2023-04-06 16:22:59--  https://raw.githubusercontent.com/cocoxu/CS4650_projects_spring2023/master/p3_bert_train.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 987712 (965K) [text/plain]
Saving to: 'p3_bert_train.csv.1'

p3_bert_train.csv.1 100%[===================>] 964.56K  --.-KB/s    in 0.004s

2023-04-06 16:23:00 (221 MB/s) - 'p3_bert_train.csv.1' saved [987712/987712]
```

```
#divide data into train, validation, and test datasets
num_tweets = len(full_df)
idxs = list(range(num_tweets))
print('Total tweets in dataset: ', num_tweets)
test_idx = idxs[:int(0.1*num_tweets)]
val_idx = idxs[int(0.1*num_tweets):int(0.2*num_tweets)]
train_idx = idxs[int(0.2*num_tweets):]

train_df = full_df.iloc[train_idx].reset_index(drop=True)
val_df = full_df.iloc[val_idx].reset_index(drop=True)
test_df = full_df.iloc[test_idx].reset_index(drop=True)

train_data = train_df[['id', 'text', 'target']]
val_data   = val_df[['id', 'text', 'target']]
test_data  = test_df[['id', 'text', 'target']]
```

```
Total tweets in dataset:  7613
```

```
#Defining torch dataset class for disaster tweet dataset
class TweetDataset(Dataset):
    def __init__(self, df):
        self.df = df

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        return self.df.iloc[idx]
```

```
#set up train, validation, and testing datasets
train_dataset = TweetDataset(train_data)
val_dataset   = TweetDataset(val_data)
test_dataset  = TweetDataset(test_data)
```

The following code creates a collate function for our tweet dataset that will tokenize the input tweets for use with our BERT models.

```
def transformer_collate_fn(batch, tokenizer):
  bert_vocab = tokenizer.get_vocab()
  bert_pad_token = bert_vocab['[PAD]']
  bert_unk_token = bert_vocab['[UNK]']
  bert_cls_token = bert_vocab['[CLS]']

  sentences, labels, masks = [], [], []
  for data in batch:
    tokenizer_output = tokenizer([data['text']])
    tokenized_sent = tokenizer_output['input_ids'][0]
    mask = tokenizer_output['attention_mask'][0]
    sentences.append(torch.tensor(tokenized_sent))
    labels.append(torch.tensor(data['target']))
    masks.append(torch.tensor(mask))
  sentences = pad_sequence(sentences, batch_first=True, padding_value=bert_pad_token)
  labels = torch.stack(labels, dim=0)
  masks = pad_sequence(masks, batch_first=True, padding_value=0.0)
  return sentences, labels, masks
```

## Model Helpers

This section defines helper functions for model training, evaluation, and inspection. You do not need to modify any code in the Model Helpers section.

```
#computes the amount of time that a training epoch took and displays it in human readable form
def epoch_time(start_time: int,
               end_time: int):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

```
#count the number of trainable parameters in the model
def count_parameters(model: nn.Module):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```
#train a given model, using a pytorch dataloader, optimizer, and scheduler (if provided)
def train(model,
          dataloader,
          optimizer,
          device,
          clip: float,
          scheduler = None):

    model.train()

    epoch_loss = 0

    for batch in dataloader:
        sentences, labels, masks = batch[0], batch[1], batch[2]

        optimizer.zero_grad()

        output = model(sentences.to(device), masks.to(device))
        loss = F.cross_entropy(output, labels.to(device))
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()
        if scheduler is not None:
          scheduler.step()

        epoch_loss += loss.item()
    return epoch_loss / len(dataloader)
```

```
#calculate the loss from the model on the provided dataloader
def evaluate(model,
             dataloader,
             device):

    model.eval()

    epoch_loss = 0
    with torch.no_grad():
      for batch in dataloader:
          sentences, labels, masks = batch[0], batch[1], batch[2]
          output = model(sentences.to(device), masks.to(device))
          loss = F.cross_entropy(output, labels.to(device))

          epoch_loss += loss.item()
    return epoch_loss / len(dataloader)
```

```
#calculate the prediction accuracy on the provided dataloader
def evaluate_acc(model,
                 dataloader,
                 device):

    model.eval()

    epoch_loss = 0
    with torch.no_grad():
      total_correct = 0
      total = 0
      for i, batch in enumerate(dataloader):

          sentences, labels, masks = batch[0], batch[1], batch[2]
          output = model(sentences.to(device), masks.to(device))
          output = F.softmax(output, dim=1)
          output_class = torch.argmax(output, dim=1)
          total_correct += torch.sum(torch.where(output_class == labels.to(device), 1, 0))
          total += sentences.size()[0]

    return total_correct / total
```

## Model Setup

```
#first, install the hugging face transformer package in your colab
!pip install -q transformers
from transformers import get_linear_schedule_with_warmup
from tokenizers.processors import BertProcessing
```

Having prepared our datasets, we now need to load in a BERT model for use as an encoder. Fortunately, the Hugging Face Library makes this easy for us. Use the hugging face AutoClass functionality to set up a pretrained Distill BERT Model and its corresponding tokenizer (1 Point). You will need to import functionality from the Hugging Face library for this question. If you are curious about the differences between BERT and Distil Bert, please see this page within the Huggingface Documentation: [https://huggingface.co/transformers/model_summary.html](https://huggingface.co/transformers/model_summary.html)

```
# Do not change this line, as it sets the model the model that Hugging Face will load
# If you are interested in what other models are available, you can find the list of model names here:
# https://huggingface.co/transformers/pretrained_models.html
bert_model_name = 'distilbert-base-uncased'

##YOUR CODE HERE##

from transformers import DistilBertTokenizer, DistilBertModel
bert_model = DistilBertModel.from_pretrained(bert_model_name)
tokenizer = DistilBertTokenizer.from_pretrained(bert_model_name)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing DistilBertModel: ['vocab_transform.bias
- This IS expected if you are initializing DistilBertModel from the checkpoint of a model trained on another task or with another archi
- This IS NOT expected if you are initializing DistilBertModel from the checkpoint of a model that you expect to be exactly identical (
```

If you've loaded the archtiecture correctly, the displayed name of the model below should be "DistilBertModel"

```
#print the loaded model architecture
bert_model
```

```
DistilBertModel(
  (embeddings): Embeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (transformer): Transformer(
    (layer): ModuleList(
      (0-5): 6 x TransformerBlock(
        (attention): MultiHeadSelfAttention(
          (dropout): Dropout(p=0.1, inplace=False)
          (q_lin): Linear(in_features=768, out_features=768, bias=True)
          (k_lin): Linear(in_features=768, out_features=768, bias=True)
          (v_lin): Linear(in_features=768, out_features=768, bias=True)
          (out_lin): Linear(in_features=768, out_features=768, bias=True)
        )
        (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (ffn): FFN(
          (dropout): Dropout(p=0.1, inplace=False)
          (lin1): Linear(in_features=768, out_features=3072, bias=True)
          (lin2): Linear(in_features=3072, out_features=768, bias=True)
          (activation): GELUActivation()
        )
        (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      )
    )
  )
)
```

After loading the pretrained Distil BERT Model, we need to add our own classification head that we can train for our task. Assuming that the BERT encoder is a pretrained DistilBert model, add a BERT sequence classification head to architecture below. The classification head should take the encoded classification token as an input and output raw, unnormalized classification scores for each input sentence in the batch. You will need to look at the Huggingface documentation for DistilBert to complete this question, and you may want to look at the DistilBertForSequenceClassification architecture for guidance on creating a bert sequence classification head. Both can be found here: [https://huggingface.co/transformers/model_doc/distilbert.html](https://huggingface.co/transformers/model_doc/distilbert.html) . (2 Points)

Please note that we are not allowing you to directly use the DistilBertForSequenceClassification architecture, as we want you to implement the BERT sequence classification head yourself.

```
class TweetClassifier(nn.Module):
    def __init__(self,
                 bert_encoder: nn.Module,
                 enc_hid_dim=768, #default embedding size
                 outputs=2,
                 dropout=0.1):
        super().__init__()

        self.bert_encoder = bert_encoder

        self.enc_hid_dim = enc_hid_dim


        ### YOUR CODE HERE ###
        # Define a linear layer to map the output from BERT to the classification layer
        self.fc = nn.Linear(enc_hid_dim, outputs)

        # Define a dropout layer to prevent overfitting
        self.dropout = nn.Dropout(dropout)


```

```
    def forward(self,
                src,
                mask):
        bert_output = self.bert_encoder(src, mask)

        ### YOUR CODE HERE ###
        last_hidden_state = bert_output.last_hidden_state

        # Average pool across tokens to get a single vector representation
        avg_pool = torch.mean(last_hidden_state, 1)

        # Apply dropout to avoid overfitting
        x = self.dropout(avg_pool)

        # Pass through fully connected layer to get logits
        x = self.fc(x)
        return x


```

Finally, we want to intialize the weights of our classification head without overwriting the weights within the DistilBert encoder. The init_weights function below will overwrite all weights within the model. Fill in the init_classification_head_weights function so that it will only overwrite weights in the classification head (using the same initialization scheme as the init_weights function). It may be helpful to refer to the PyTorch documentation on nn.module.named_parameters() while working on this question (1 point)

It should be noted that the weight initialization scheme utilized here is automatically implemented by PyTorch Linear layers. The goal of this question is to show how to change aspects of your model's set up at the parameter level basis, not just to initialize the correct weights for this architecture. As such, stating that the PyTorch Linear layer already implements this initialiazation scheme is not sufficient to earn points for this question.

```
def init_weights(m: nn.Module, hidden_size=768):
    k = 1/hidden_size
    for name, param in m.named_parameters():
        if 'weight' in name:
            print(name)
            nn.init.uniform_(param.data, a=-1*k**0.5, b=k**0.5)
        else:
            print(name)
            nn.init.uniform_(param.data, 0)
```

```
def init_classification_head_weights(m: nn.Module, hidden_size=768):
    ### YOUR CODE STARTS HERE ###
    k = 1/hidden_size
    for name, param in m.named_parameters():
        if 'classifier.weight' in name:
            nn.init.uniform_(param.data, a=-1*k**0.5, b=k**0.5)
        elif 'classifier.bias' in name:
            nn.init.zeros_(param.data)
```

## Model Training

Once you have written the init_classification_head_weights function, you are done coding for this question. Run the following cells to initialize your model, to set up training, validation, and test dataloaders, and to train/evaluate the model. If you have completed the previous steps correctly, your model should achieve a test accuracy of 80% or greater without any hyperparameter tuning. Please note that if you need to train your model more than once, you will need to reload the BERT model to ensure that you are starting with fresh weights. Make sure that your submitted colab notebook file for includes the printed test accuracy to receive full credit for this question. (1 Point)

```
#define hyperparameters
BATCH_SIZE = 10
LR = 1e-5
WEIGHT_DECAY = 0
N_EPOCHS = 3
CLIP = 1.0

#define models, move to device, and initialize weights
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = TweetClassifier(bert_model).to(device)
model.apply(init_classification_head_weights)
model.to(device)
print('Model Initialized')
```

```
Model Initialized
```

```
#create pytorch dataloaders from train_dataset, val_dataset, and test_datset
train_dataloader = DataLoader(train_dataset,batch_size=BATCH_SIZE,collate_fn=partial(transformer_collate_fn, tokenizer=tokenizer), shuffl
val_dataloader = DataLoader(val_dataset,batch_size=BATCH_SIZE,collate_fn=partial(transformer_collate_fn, tokenizer=tokenizer))
test_dataloader = DataLoader(test_dataset,batch_size=BATCH_SIZE,collate_fn=partial(transformer_collate_fn, tokenizer=tokenizer))
```

```
optimizer = optim.Adam(model.parameters(), lr=LR)

scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=10, num_training_steps=N_EPOCHS*len(train_dataloader))

print(f'The model has {count_parameters(model):,} trainable parameters')

train_loss = evaluate(model, train_dataloader, device)
train_acc = evaluate_acc(model, train_dataloader, device)

valid_loss = evaluate(model, val_dataloader, device)
valid_acc = evaluate_acc(model, val_dataloader, device)

print(f'Initial Train Loss: {train_loss:.3f}')
print(f'Initial Train Acc: {train_acc:.3f}')
print(f'Initial Valid Loss: {valid_loss:.3f}')
print(f'Initial Valid Acc: {valid_acc:.3f}')

for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss = train(model, train_dataloader, optimizer, device, CLIP, scheduler)
    end_time = time.time()
    train_acc = evaluate_acc(model, train_dataloader, device)
    valid_loss = evaluate(model, val_dataloader, device)
    valid_acc = evaluate_acc(model, val_dataloader, device)
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f}')
    print(f'\tTrain Acc: {train_acc:.3f}')
    print(f'\tValid Loss: {valid_loss:.3f}')
    print(f'\tValid Acc: {valid_acc:.3f}')
```

```
The model has 66,364,418 trainable parameters
Initial Train Loss: 0.691
Initial Train Acc: 0.497
Initial Valid Loss: 0.696
Initial Valid Acc: 0.470
Epoch: 01 | Time: 0m 50s
        Train Loss: 0.446
        Train Acc: 0.866
        Valid Loss: 0.359
        Valid Acc: 0.854
Epoch: 02 | Time: 0m 47s
        Train Loss: 0.344
        Train Acc: 0.895
        Valid Loss: 0.364
        Valid Acc: 0.846
Epoch: 03 | Time: 0m 49s
        Train Loss: 0.295
        Train Acc: 0.906
        Valid Loss: 0.382
        Valid Acc: 0.852
```

```
#run this cell and save its outputs. A 75% test accuracy is needed for full credit.
test_loss = evaluate(model, test_dataloader, device)
test_acc = evaluate_acc(model, test_dataloader, device)
print(f'Test Loss: {test_loss:.3f}')
print(f'Test Acc: {test_acc:.3f}')
```

```
Test Loss: 0.511
Test Acc: 0.808
```

## Part 7: Beam Search (Extra Credit, 10 points)

Similar to greedy search, beam search generates one token at a time. However, rather than keeping only the single best hypothesis, we instead keep the top $k$ candidates at each time step. This is accomplished by computing the set of next-token extensions for each item on the beam and finding the top $k$ across all candidates according to total log-probability.

Candidates that are finished should be extracted in a final list of `generations` and removed from the beam. This strategy is useful for doing re-ranking the beam candidates using alternate scorers (example, Maximum Mutual Information Objective from Li et. al. 2015). For this assignment, you will re-rank the beam generations as follows,
$final\_score_i = \frac{score_i}{|generation_i|^{\alpha}}$, where $\alpha \in [0.5, 2]$.
Terminate the search process once you have $k$ items in the `generations` list.

*HINT*: Given the simplicity of the dataset we're working with, it's likely that the resonses from your model will be similar to each other but they should not be the exact same.

```
def predict_beam(model, sentence, k=5, max_length=100):
    """Make predictions for the given inputs using beam search.

    Args:
        model: A sequence-to-sequence model.
        sentence: An input sentence, represented as string.
        k: The size of the beam.
        max_length: The maximum length at which to truncate outputs in order to
            avoid non-terminating inference.

```

```
    Returns:
        A list of k beam predictions. Each element in the list should be a string
        corresponding to one of the top k predictions for the corresponding input,
        sorted in descending order by its final score.
    """

    # Implementation tip: once an eos_token has been generated for any beam,
    # remove its subsequent predictions from that beam by adding a small negative
    # number like -1e9 to the appropriate logits. This will ensure that the
    # candidates are removed from the beam, as its probability will be very close
    # to 0. Using this method, uou will be able to reuse the beam of an already
    # finished candidate

    # Implementation tip: while you are encouraged to keep your tensor dimensions
    # constant for simplicity (aside from the sequence length), some special care
    # will need to be taken on the first iteration to ensure that your beam
    # doesn't fill up with k identical copies of the same candidate.

    # You are welcome to tweak alpha
    alpha = 0.7
    model.eval()

    # YOUR CODE HERE
    # Tokenize input sentence
    input = torch.tensor(vocab.get_ids_from_sentence(normalize_sentence(sentence))).unsqueeze(1).to(device)
    inputs, encode_mask, encode_hidden = model.encode(input)
    hidden = encode_hidden

    # Number of input tokens
    num_tokens = inputs.size(1)

    # Expand inputs to size k
    inputs = inputs.expand(k, num_tokens)

    # Initialize scores and output sequences
    seq_scores = torch.zeros(k, 1)
    seq_outputs = inputs.clone()

    # Initialize the hidden state and cell state of the decoder with zeros
    # hidden = torch.zeros(model.decoder.num_layers, k, model.decoder.hidden_size)
    cell = torch.zeros(model.decoder.num_layers, k, model.decoder.hidden_size)

    # The first input to the decoder is the <sos> token
    decoder_input = torch.tensor([[bos_id]])

    # List to store completed sequences and their scores
    completed_seqs = []
    completed_seq_scores = []

    for i in range(max_length):

        # Pass the inputs and the decoder state through the decoder to get
        # the logits and the new decoder state
        logits, hidden, cell = model.decode(decoder_input, hidden)

        # Apply softmax to the logits to get the probabilities over the vocabulary
        probs = F.softmax(logits, dim=-1)

        # Multiply the probabilities by the scores of the corresponding sequence
        # and take the sum over the sequence dimension to get the new scores
        scores = seq_scores.expand_as(probs) * probs
        scores = scores.reshape(-1, k)

        # Keep the top k scores and their corresponding indices
        top_scores, top_indices = torch.topk(scores, k, dim=1)

        # Convert the flattened indices to the indices within the sequence and
        # the beam
        beam_indices = top_indices // len(tokenizer)
        token_indices = top_indices % len(tokenizer)

        # Append the new tokens and scores to the output sequences and scores
        new_seq_outputs = torch.cat([seq_outputs[beam_indices, :], token_indices.unsqueeze(-1)], dim=-1)
        new_seq_scores = top_scores.view(-1, 1)

        # Check if any of the sequences have reached the end token
        eos_mask = token_indices == eos_id
        if eos_mask.any():

            # Remove the completed sequences from the current sequences
            new_seq_outputs = new_seq_outputs[~eos_mask, :]
            new_seq_scores = new_seq_scores[~eos_mask, :]

            # Get the completed sequences and their scores
            completed_seqs.extend(new_seq_outputs[eos_mask, :])
            completed_seq_scores.extend(new_seq_scores[eos_mask, :])

            # If all sequences have been completed, break out of the loop
            if len(completed_seqs) == k:
                break

```

```
        # Otherwise, reduce k to the number of incomplete sequences
        k -= eos_mask.sum().item()
```

Now let's test both baseline and attention models on some predefined inputs and compare their greedy and beam responses side by side.

```
test_conversations_with_model(baseline_model, include_beam=False)
```

```
Input > hello.
Greedy Response: hello .

Input > please share you bank account number with me
Greedy Response: i m not sure .

Input > i have never met someone more annoying that you
Greedy Response: i know .

Input > i like pizza. what do you like?
Greedy Response: i don t know .

Input > give me coffee, or i'll hate you
Greedy Response: you re not going to get my car ?

Input > i'm so bored. give some suggestions
Greedy Response: i m sorry .

Input > stop running or you'll fall hard
Greedy Response: i don t know .

Input > what is your favorite sport?
Greedy Response: i don t know .

Input > do you believe in a miracle?
Greedy Response: no .

Input > which sport team do you like?
Greedy Response: i don t know .
```

```
test_conversations_with_model(baseline_model, include_beam=True)
```

```
Input > hello.
Greedy Response: hello .
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-206-4c9801536792> in <cell line: 1>()
----> 1 test_conversations_with_model(baseline_model, include_beam=True)

1 frames
<ipython-input-205-609fb518387a> in predict_beam(model, sentence, k, max_length)
     41 
     42     # Expand inputs to size k
---> 43     inputs = inputs.expand(k, num_tokens)
     44 
     45     # Initialize scores and output sequences

RuntimeError: expand(torch.cuda.FloatTensor{[4, 1, 300]}, size=[5, 1]): the number of sizes provided (2) must be greater or equal to
the number of dimensions in the tensor (3)
```

SEARCH STACK OVERFLOW

+ 代码 + 文本

```
test_conversations_with_model(attention_model, include_beam=False)
```

```
test_conversations_with_model(attention_model, include_beam=True)
```

Let's also check how our models do using our automatic evaluation metrics.

```
print(f"Baseline Model evaluation:")
avg_length, distinct1, distinct2 = evaluate_diversity(baseline_model)
print(f"Greedy decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
avg_length, distinct1, distinct2 = evaluate_diversity(baseline_model, mode='beam')
print(f"Beam search decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
print(f"Attention Model evaluation:")
avg_length, distinct1, distinct2 = evaluate_diversity(attention_model,)
print(f"Greedy decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
avg_length, distinct1, distinct2 = evaluate_diversity(attention_model, mode='beam')
print(f"Beam decoding:")
print(f"Avg Response Length = {avg_length}")
print(f"Distinct1 = {distinct1}")
print(f"Distinct2 = {distinct2}")
```

## What to turn in?

This is the end. Congratulations!

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

1. Rename this ipynb file to 'CS4650_p2_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 your training loops are captured so we can see how the loss and accuracy changes while training.
5. Upload all 3 files to GradeScope.