Recurrent Neural Network and Multi-Head Attention (MHA)

In this task, you will implement a conventional RNN cell, a GRU, and an MHA to understand these models. Then you will configurate GRU in special ways such that it either recovers a conventional RNN or keeps its memory in long term. NOTE: you should not change the provided function interfaces and test cases.

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/cs137assignments/assignment4')
```

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

```
# As usual, a bit of setup
import time
import numpy as np
import torch
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%autosave 180

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
Autosaving every 180 seconds
```

Recurrent Neural Networks

In this task, you will need to implement forward calculation of recurrent neural networks. Let's first initialize a problem for RNNs.

```
import torch.nn as nn
## Setup an example. Provide sizes and the input data.

# set sizes
time_steps = 12
batch_size = 4
input_size = 3
hidden_size = 2

# create input data with shape [batch_size, time_steps, num_features]
np.random.seed(137)
input_data = torch.randn(batch_size, time_steps, input_size, dtype = torch.float32)
## Create RNN layers

# initialize a state of zero for both RNN and GRU
# 'state' is a tensor of shape [batch_size, hidden_size]
initial_state = torch.randn(batch_size, hidden_size, dtype = torch.float32).unsqueeze(
```

Implement an RNN and a GRU with PyTorch

```
# create an RNN with only one layer from torch
t_rnn = nn.RNN(input_size, hidden_size, num_layers = 1, batch_first = True)

# 'outputs' is a tensor of shape [batch_size, time_steps, hidden_size]
# RNN cell outputs the hidden state directly, so the output at each step is the hidden
# final_state is the last state of the sequence. final_state == outputs[:, -1, :]

# create a GRU RNN
t_gru = nn.GRU(input_size, hidden_size, num_layers = 1, batch_first = True)

with torch.no_grad():
    t_rnn_outputs, t_rnn_final_state = t_rnn(input_data, initial_state)
    # 'outputs' and `final_state` are the same for a GRU.
    t_gru_outputs, t_gru_final_state = t_gru(input_data, initial_state)
```

Read out parameters from RNN and GRU cells

Q1 (0 points) Understanding `RNN` and `GRU` parameters

Please read the code and documentation of `get_rnn_params` and `get_gru_params` to see how to read out parameters from these to models. You will need to use these parameters in your own implementations. NO implementation is needed here.

```
from rnn_param_helper import get_rnn_params, get_gru_params
```

```
wt_h, wt_x, bias = get_rnn_params(t_rnn)

# NOTE: please check the documentation of `torch.nn.GRU` and the implementation of `ge
# understand the three returning arguments.

linear_trans_r, linear_trans_z, linear_trans_n = get_gru_params(t_gru)
```

Numpy Implementation

Q2 (3 points) Please implement your own simple RNN.

Your implementation needs to match the tensorflow calculation.

Q3 (5 points) Please implement your own GRU.

Your implementation needs to match the tensorflow calculation.

```
from implementation import rnn,gru

# calculation from your own implemenation of a basic RNN
nprnn_outputs, nprnn_final_state = rnn(wt_h, wt_x, bias, initial_state.numpy(), input_

print("Difference between your RNN implementation and tf RNN",
                     rel_error(t_rnn_outputs.numpy(), nprnn_outputs) + rel_error(t_rnn

# calculation from your own implemenation of a GRU RNN
npgru_outputs, npgru_final_state = gru(linear_trans_r, linear_trans_z, linear_trans_n,

print("Difference between your GRU implementation and tf GRU",
      rel_error(t_gru_outputs.numpy(), npgru_outputs) + rel_error(t_gru_final_state.nu
```

```
Difference between your RNN implementation and tf RNN 4.474195e-07
Difference between your GRU implementation and tf GRU 4.826031e-07
```

GRU includes RNN as a special case

Q4 (2 points) Can you assign a special set of parameters to GRU such that its outputs is almost the same as RNN?

```
# Assign some value to a parameter of GRU

from implementation import init_gru_with_rnn

linear_trans_r, linear_trans_z, linear_trans_n = init_gru_with_rnn(wt_h, wt_x, bias)

# concatenate these parameters to initialize GRU kernels
```

```
kernel_init = np.concatenate([linear_trans_r[0], linear_trans_z[0], linear_trans_n[0]]
rec_kernel_init = np.concatenate([linear_trans_r[2], linear_trans_z[2], linear_trans_r
bias_init0 = np.concatenate([linear_trans_r[1], linear_trans_z[1], linear_trans_n[1]],
bias_init1 = np.concatenate([linear_trans_r[3], linear_trans_z[3], linear_trans_n[3]])

grurnn = nn.GRU(input_size, hidden_size, num_layers = 1, batch_first = True)
wt_x1, wt_h1, bias_ih1, bias_hh1 = grurnn._flat_weights

wt_x1.data = torch.tensor(kernel_init, dtype =torch.float32)
wt_h1.data = torch.tensor(rec_kernel_init, dtype = torch.float32)
bias_ih1.data = torch.tensor(bias_init0, dtype = torch.float32)
bias_hh1.data = torch.tensor(bias_init1, dtype = torch.float32)


# 'outputs' is a tensor of shape [batch_size, time_steps, hidden_size]
# Same as the basic RNN cell, final_state == outputs[:, -1, :]
with torch.no_grad():
    t_rnn_outputs, t_rnn_final_state = t_rnn(input_data, initial_state)
    grurnn_outputs, grurnn_final_state = grurnn(input_data, initial_state)

# they are the same as the calculation from the basic RNN
print("Difference between RNN and a special GRU", rel_error(t_rnn_outputs.numpy(), gru
```

```
Difference between RNN and a special GRU 4.316596e-07
```

Long-term dependency in GRUs

Q5 (2 points) Can you set GRU parameters such that it maintains the initial state in the memory for a long term?

```
from implementation import init_gru_with_long_term_memory

linear_trans_r, linear_trans_z, linear_trans_n = init_gru_with_long_term_memory(input_

# concatenate these parameters to initialize GRU kernels
kernel_init = np.concatenate([linear_trans_r[0], linear_trans_z[0], linear_trans_n[0]]
rec_kernel_init = np.concatenate([linear_trans_r[2], linear_trans_z[2], linear_trans_r
bias_init0 = np.concatenate([linear_trans_r[1], linear_trans_z[1], linear_trans_n[1]],
bias_init1 = np.concatenate([linear_trans_r[3], linear_trans_z[3], linear_trans_n[3]])

gru2 = nn.GRU(input_size, hidden_size, num_layers=1, batch_first=True)
wt_xg, wt_hg, bias_ihg, bias_hhg = gru2._flat_weights

wt_xg.data = torch.tensor(kernel_init, dtype = torch.float32)
wt_hg.data = torch.tensor(rec_kernel_init, dtype = torch.float32)
bias_ihg.data = torch.tensor(bias_init0, dtype = torch.float32)
bias_hhg.data = torch.tensor(bias_init1, dtype = torch.float32)
```

```
with torch.no_grad():
    outputs, _ = gru2(input_data, initial_state)
    outputs = outputs.numpy()
    print('Difference between a later hidden state and the initial state is', np.mean(
```

```
Difference between a later hidden state and the initial state is 0.0
```

Double-click (or enter) to edit

Implement a multi-head attention layer

Q6 (5 points) In the task, you need to implement the forward calculation of a multi-head attention layer. Your calculation needs to match the calculation of the torch MHA layer in the following test case.

```
from rnn_param_helper import get_mha_params
from implementation import mha


batch_size = 4
time_steps = 8
input_size = 10
num_heads = 5

input_data = torch.randn(batch_size, time_steps, input_size, dtype = torch.float32)


# run torch implementation of MHA
with torch.no_grad():

    t_mha = nn.MultiheadAttention(embed_dim=input_size, num_heads=num_heads, dropout=(

    t_output, _ = t_mha(input_data, input_data, input_data, need_weights=False)


# extract model parameters from the torch MHA layer
Wq, Wk, Wv, Wo = get_mha_params(t_mha)

# run the same calculation with your implementation
output = mha(Wq, Wk, Wv, Wo, input_data )

print('Difference between my output and torch output is ', np.mean(np.abs(output - t_o
```

```
Difference between my output and torch output is  1.7877756e-08
```

0s completed at 11:10 PM

RNN/Transformer for Modeling Sentences

In this task, we will use an RNN or a transformer model to model sentences. The task is to predict the next character in a sentence.

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/cs137assignments/assignment4')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

```
# As usual, a bit of setup
import time
import numpy as np
import torch
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%autosave 180
```

```
Autosaving every 180 seconds
```

```
# If you have cuda, do the following
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda:0
```

Load the data

```
import csv
import string
import numpy as np
```

```
def load_data(data_file):
    """Load the data into a list of strings"""

    with open('/content/drive/MyDrive/cs137assignments/assignment4/'+data_file) as csv
        reader = csv.reader(csv_file, delimiter=',')
        rows = list(reader)

    if data_file == 'train.csv':
        sentences, labels = zip(*rows[1:])
        sentences = list(sentences)
    elif data_file == 'test.csv':
        sentences = [row[0] for row in rows[1:]]
    else:
        print("Can only load 'train.csv' or 'test.csv'")

    # replace non ascii chars to spaces
    count = 0
    for i, sen in enumerate(sentences):
        count = count + sum([0 if ord(i) < 128 else 1 for i in sen])

        # '\n' indicates the end of the sentence
        sentences[i] = ''.join([i if ord(i) < 128 else ' ' for i in sen]) + '\n'

    print('The total of ', count, 'non-ascii chars are removed \n')

    return sentences

def char_to_index(sentence, str_voc):
    """Convert a string to an array by using the index in the vocabulary"""

    sen_int = np.array([str_voc.index(c) for c in sentence])
    return sen_int

def convert_sen_to_data(sentences, str_voc):
    """ Convert a list of strings to a list of numpy arrays"""
    data = [None] * len(sentences)
    for i, sen in enumerate(sentences):
        data[i] = char_to_index(sen, str_voc)

        # sanity check
        #if i < 5:
        #    recover = "".join([str_voc[k] for k in data[i]])
        #    print(recover)
    return data

train_sentences = load_data('train.csv')

# NOTE: you need to use the same vocabulary to handle your test sentences
vocabulary = list(set("".join(train_sentences)))
vocabulary.sort()
```

```
str_voc = "".join(vocabulary)

train_data = convert_sen_to_data(train_sentences, str_voc)


num_sen = len(train_data)
sen_lengths = [sen.shape[0] for sen in train_data]
max_len = max(sen_lengths)
min_len = min(sen_lengths)
num_chars = sum(sen_lengths)

print('Data statistics:')
print('Number of sentences: ', num_sen)
print('Maximum and minimum sentence lengths:', max_len, min_len)
print('Total number of characters:', num_chars)
print('Vocabulary size: ', len(vocabulary))

uniq, uniq_counts = np.unique(np.concatenate(train_data), return_counts=True)
freq = np.zeros_like(uniq_counts)
freq[uniq] = uniq_counts

print('Chars in vocabulary and their frequencies:')
print(list(zip(vocabulary, freq.tolist())))


# a sample sentence
print("Data exploration -- showing an example sentence:")

sample = ""
for i in train_data[5]:
    sample +=str_voc[i]
print(sample)
```

```
The total of  4328 non-ascii chars are removed

Data statistics:
Number of sentences:  160000
Maximum and minimum sentence lengths: 100 32
Total number of characters: 10954565
Vocabulary size:  95
Chars in vocabulary and their frequencies:
[('\n', 160000), (' ', 1762678), ('!', 12100), ('#', 496), ('$', 1212), ('%', 45
Data exploration -- showing an example sentence:
Martha stewart tweets hideous food photo, twitter responds accordingly
```

Implement an RNN or a Transformer with torch

Q7 (10 points) In this problem, you are supposed to train an RNN or a transformer to model sentences. Particuarly, your model will receive 10 starting characters and should predict the rest of sentence. The model will be evaluated by per-character cross-entropy loss. You will get

- 5 points if your per-character cross-entropy loss is less than 2.5 (the loss by predicting with character frequencies is 3.13. Your model needs to be better than that).
- 8 points if your per-character cross-entropy loss is less than 2
- 10 points if your per-character cross-entropy loss is less than 1.5

*The performance from a paper indicates that an LSTM can achieve performance of 1.43 * ln(2) = 0.991. *The `zip` program for compressing files roughly can achieve a performances of 3.522 bits

```
# Set up dataloader

# TODO: please read through the code in this cell so you know the data your model will

from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence


class StrData(Dataset):
    def __init__(self, data):
        self.sentence = data
    def __len__(self):
        return len(self.sentence)
    def __getitem__(self, idx):
        return self.sentence[idx]

BEGIN_ID = freq.shape[0]
END_ID = BEGIN_ID + 1
PAD_ID = BEGIN_ID + 2

def add_begin_and_end(tokens):
    return torch.cat([torch.tensor([BEGIN_ID], dtype = torch.long),
                      torch.tensor(tokens, dtype = torch.long),
                      torch.tensor([END_ID], dtype = torch.long)])

def collate_fn(batch):
    batch_ret = []
    for sentence in batch:
        batch_ret.append(add_begin_and_end(sentence))
    batch_ret = pad_sequence(batch_ret, padding_value = PAD_ID).T # pad_sequence is nc
    return batch_ret
```

Set up a model

Suggestion: you may want to put your model in a `.py` file. Your code might look cleaner if you do so.

```
from rnn_lm import SentenceModel

model = SentenceModel(freq)
model.to(device)
```

```
SentenceModel(
  (emb): Embedding(98, 256)
  (rnn): RNN(256, 1000, batch_first=True)
  (linear): Linear(in_features=1000, out_features=98, bias=True)
)
```

Train the model

NOTE: this example only uses 20 sentences for fast showcase the code, but you should use the entire training set. You can also split out a subset as the validation set. You can make any changes as long as you don't touch the test set.

```
epochs = 5

train_loader = DataLoader(StrData(train_data), shuffle=True, batch_size=64, collate_fr

opt = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = torch.nn.CrossEntropyLoss(ignore_index = PAD_ID)
for ep in range(epochs):
    running_loss = 0
    for i, batch in enumerate(train_loader):

        m_input = batch[:, :-1]
        m_output = batch[:, 1:]

        if device.type == "cuda":
            m_input = m_input.to(device)
            m_output = m_output.to(device)

         # zero the parameter gradients
        opt.zero_grad()

        logits = model(m_input) # batch x no_sequences x logits
        # Question: is this teacher forcing?
        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), m_output.reshape(-1)) #
        loss.backward()

        opt.step()
```

```
        # TODO: Record loss values to some variable
        if device.type == "cuda":
            loss = loss.cpu()

        running_loss += loss.item()
    print(f"Epoch {ep+1}/{epochs}: Training Loss {running_loss / (i+1)}")
```

```
Epoch 1/5: Training Loss 1.598863556098938
Epoch 2/5: Training Loss 1.410946901845932
Epoch 3/5: Training Loss 1.372951064491272
Epoch 4/5: Training Loss 1.3548198021888733
Epoch 5/5: Training Loss 1.3454062554836272
```

Save the model

```
torch.save(model, "rnn_lm.sav")
```

Test the trained model

```
# load the test data. NOTE: need to use the same vocabulary as the training data
test_sentences = load_data('test.csv')
test_data = convert_sen_to_data(test_sentences, str_voc)


print('Number of test instances:', len(test_data))

# TODO: replace this stub model with your powerful model
model = torch.load("rnn_lm.sav")

test_loader = DataLoader(StrData(test_data), shuffle=True, batch_size=50, collate_fn=c
loss_fn = torch.nn.CrossEntropyLoss(ignore_index = PAD_ID)
print('Evaluating the model ...')
loss_sum = 0
char_count = 0
with torch.no_grad():
    for i, batch in enumerate(test_loader):
        m_input = batch[:, :-1]
        m_output = batch[:, 1:]

        if device.type == "cuda":
            m_input = m_input.to(device)
            m_output = m_output.to(device)

        logits = model(m_input)
        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), m_output.reshape(-1))
```

```
        if device.type == "cuda":
            loss = loss.to(device)
            batch = batch.to(device)

        loss_sum += loss.item()
        char_count += torch.sum((batch != PAD_ID) & (batch != BEGIN_ID) & (batch != EN

per_char_loss = loss_sum / (i+1)

print('The total number of chars in the test set is ', char_count)

print('The per-char-loss is %.3f' % per_char_loss)
```

```
The total of  1131 non-ascii chars are removed

Number of test instances: 40000
Evaluating the model ...
The total number of chars in the test set is  tensor(2739550, device='cuda:0')
The per-char-loss is 1.363
```

Use the model to generate sentences

Now we can use the trained model to generate text with a starting string. The naive model just predict frequent characters in the text, so there is no meaningful generation yet. See what you get from your models.

```
import torch.distributions as distributions

def generate_text(model, start_string, str_voc):
    """ Generate random text from a starting string. """

    # Number of characters to generate
    num_generate = 100 - len(start_string)

    # Converting our start string to numbers (vectorizing)
    input_int = [BEGIN_ID] + [str_voc.index(s) for s in start_string]
    input_tensor = torch.tensor(input_int, dtype = torch.long).view([1, -1])

    # Empty string to store our results
    text_generated = []

    # Low temperature results in more predictable text.
    # Higher temperature results in more surprising text.
    # Experiment to find the best setting.
    temperature = 0.5

    # Here batch size == 1
    other_voc = {BEGIN_ID: "<BEG>", END_ID: "<END>", PAD_ID: "<PAD>"}
```

```
    for i in range(num_generate):

        if device.type == "cuda":
            input_tensor = input_tensor.to(device)

        outputs = model(input_tensor)

        # remove the batch dimension
        prediction = torch.softmax(outputs[0, -1, :], dim=0)

        # using a categorical distribution to predict the character returned by the mc
        prediction = prediction / temperature
        predicted_id = int(distributions.Categorical(probs = prediction).sample())


        # The calculation has a lot of repeatition because computation for the first p
        # of the sequence is the same at every iteration. But it's fine for our exampl
        input_int.append(predicted_id)
        input_tensor = torch.tensor(input_int, dtype = torch.long).view([1, -1])


        text_generated.append(str_voc[predicted_id] if (predicted_id < len(str_voc)) e

    return (start_string + ''.join(text_generated))


start_string = 'I hav'
gen_sen = generate_text(model, start_string, str_voc)
gen_sen = gen_sen.split('\n')[0]

print('Starting from "' + start_string + '", the generated sentence is:')
print('"' + gen_sen + '"')
```

```
Starting from "I hav", the generated sentence is:
"I haven test (photos)"
```

✓ 6s completed at 3:27 PM