503 Final Pedagogy Example Assignment

This is sample End-to-End ASR code. This model is meant to be broken and played with to help you better understand the standard CTC E2E architecture. As such, it's not a particularly powerful model, but shares the basic structure of other models like Deep Speech.

For the assignment, there are a number of items to change in the code, to help you understand it better.

Q1 Commenting

- Show understanding of the model by commenting in the indicated sections. Some of the comments have questions filled in, be sure to think about these questions and answer appropriately.

Q2 Minor Changes

- The core model performs extremely poorly. Try adding a CNN layer as well as increasing the power of the GRU to enhance performance. Think carefully about dimensions between layers!
- To help debug the code, edit the dataset creation code to only partially load the dataset.
- Experiment with using MFCC instead of Mel Spectrogram. (Think about how this might impact features!)

Q3 Implement Beam Search for CTC

- A major way to improve E2E ASR is through better decoding as well as incorporating a language model. For this question, you should first implement a bigram language model (based simply off of transitions between characters in the dataset). Then use this language model and implement the Beam Search as outlined in Graves and Jaitly's paper Towards End to End Speech Recognition with Recurrent Neural Networks. (http://proceedings.mlr.press/v32/graves14.pdf). This will require you to carefully think through the outlined algorithm.

In [ ]:
```
import pandas as pd
import numpy as np
import os
import librosa
from torch import nn, optim
import torch.nn.init as init
import torch
import soundfile as sf
from pathlib import Path
from torch.utils.data import DataLoader, Dataset
import sys
from datetime import datetime
```

In [ ]:
```
manual_seed = 503
torch.manual_seed(manual_seed)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
n_gpu = torch.cuda.device_count()
if n_gpu > 0:
    torch.cuda.manual_seed(manual_seed)
```

In [ ]:
```
config = {"n_mels": 40,
          "n_fft": 512,
          "hop_length": 32,
          "fmax" : 8000,
         }
```

In [ ]:
```
import re

# Q1- Comment:
# Why does this start from 1?
char2index = {
          "A": 1,
          "B": 2,
          "C": 3,
          "D": 4,
          "E": 5,
          "F": 6,
          "G": 7,
          "H": 8,
          "I": 9,
          "J": 10,
          "K": 11,
          "L": 12,
          "M": 13,
          "N": 14,
          "O": 15,
          "P": 16,
          "Q": 17,
          "R": 18,
          "S": 19,
          "T": 20,
          "U": 21,
          "V": 22,
          "W": 23,
          "X": 24,
          "Y": 25,
          "Z": 26,
          " ": 27,
        }

index2char = {value: key for key, value in char2index.items()}
index2char[0] = ""

# Q1- Comment -
def preprocess(txt):
    return [char2index[a] for a in list(re.sub('[^a-zA-Z ]+', '', txt))[1:]]


class LibrisDataset(Dataset):
    def __init__(self, path, config):
        self.directory = Path(path)
        starttime =  datetime.now()
        texts = []
        files = []
        X = []
        y = []

        # Q2 Change this somehow to only partially load the the dataset (usefu
l for testing)
        for book_dir in os.listdir(self.directory):
            for rec_dir in os.listdir(self.directory / book_dir):
                tempfiles = []
```

```
                for filename in os.listdir(self.directory / book_dir / rec_dir
):
                    if filename.endswith(".flac"):
                        tempfiles.append(self.directory / book_dir / rec_dir /
filename)
                    if filename.endswith(".txt"):
                        with open(self.directory / book_dir / rec_dir / filena
me,"r") as fp:
                            for line in fp:
                                texts.append(preprocess(line))
                tempfiles.sort()
                files = files + tempfiles

        d = {"f": files,"trans": texts}
        files = []
        texts = []
        df = pd.DataFrame(data=d)
        _min, _max = float('inf'), -float('inf')
        for f in df.index:
            # Q1- Comment

            signal, rate = sf.read(df["f"][f])
            df.at[f, 'length'] = signal.shape[0] / rate

            # - Q2 Switch this to an MFCC
            S = librosa.feature.melspectrogram(signal, sr=rate, n_mels=config[
"n_mels"], n_fft=config["n_fft"],
                                               hop_length=config["hop_length"
], fmax=config["fmax"])
            X_sample = librosa.power_to_db(S, ref=np.max)

            # Q1- Comment
            _min = min(np.amin(X_sample), _min)
            _max = max(np.amax(X_sample), _max)
            X.append(X_sample.swapaxes(1,0))
            y.append(torch.tensor(df["trans"][f]))

        # Q1- Comment

        for j, sample in enumerate(X):
            X[j] = (sample - _min) / (_max - _min)
        print('Elapsed Time: {}'.format(datetime.now()- starttime))
        self.X, self.y = X , y


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

    def __getitem__(self, idx):

        # Q1- Comment
        # Why is the tuple important?
        return self.X[idx], self.y[idx], (len(self.X[idx]), len(self.y[idx]))
```

In [ ]:
```
# Q0 Download Libris 100 clean training to load here.
trainDataset = LibrisDataset("", config)
```

In [ ]:
```
def pad_batch(DataLoaderBatch):
    # Q1- Comment
    batch_size = len(DataLoaderBatch)
    batch_split = list(zip(*DataLoaderBatch))

    seqs, targs, lengths = batch_split[0], batch_split[1], batch_split[2]
    input_lengths, targ_lengths = list(zip(*lengths))
    max_input_length = max(input_lengths)
    max_targ_length = max(targ_lengths)
    dims = len(seqs[0][0])

    padded_input_seqs = np.zeros((batch_size, max_input_length, dims))
    for i, l in enumerate(input_lengths):
        padded_input_seqs[i, 0:l] = seqs[i][0:l]

    padded_targ_seqs = np.zeros((batch_size, max_targ_length))
    for i, l in enumerate(targ_lengths):
        padded_targ_seqs[i, 0:l] = targs[i][0:l]

    return torch.tensor(padded_input_seqs), torch.tensor(padded_targ_seqs), le
ngths
```

In [ ]:
```
BATCH=16
# Q2 - Change this to make use of our pad_batch function
trainLoader = DataLoader(trainDataset, batch_size=BATCH, shuffle=True)
```

In [ ]:
```
class ASR(nn.Module):
    def __init__(self, input_dim, vocab):
        super(ASR, self).__init__()

        #Q2 TO DO: Expand this model to include a 1D CNN as first layer
        # Change the GRU to be bidirectional, increase size to 3 layers and 20
0 hidden dimensions
        # There is one more layer needed to make the dimensions work, add it i
n as well.

        self.rnn = nn.GRU(input_dim, vocab, batch_first=True, bidirectional=Fa
lse, num_layers=1)
        self.sm = nn.LogSoftmax(dim=2)
    def forward(self, x):

        # Q1 Comment - Why might flatten params be helpful?
        self.rnn.flatten_parameters()
        output, hn = self.rnn(x)
        return self.sm(output)
```

In [ ]:
```
# Q3 - Build a Bigram Character model to work with decoding. Should take in th
e list of labels from the training set and
# produce a log-probability chance of character-to-character transitions occur
ing.

def buildLM(labels, vocab):
    #TO DO

def language_model(seq, lm):
    #TO DO
```

In [ ]:
```
LM = buildLM(trainDataset.y, 28)
```

In [ ]:
```
def process_output(indexes, lengths, dictionary):
    sequences = ["" for x in range(len(indexes))]
    last_labels = [0 for x in range(len(indexes))]
    # Q1 Comment
    for t in range(max(lengths)):
        for i, index in enumerate(indexes[:,t]):
            if lengths[i] > t and last_labels[i] != index:
                last_labels[i] = index
                sequences[i] +=dictionary[int(index)]
    return sequences

def best_path_decoding(model_output, lengths, dictionary):
    # Comment -
    model_output = model_output.permute(1,0,2)
    indexes = torch.argmax(model_output, dim=2)
    return process_output(indexes, lengths, dictionary)
```

In [ ]:
```
# Q3 To Do:
# Implement Beam search for CTC as described in http://proceedings.mlr.press/v
32/graves14.pdf
```

In [ ]:
```
model = ASR(40,28)
model =model.double()
model= model.to(device)
criterion = nn.CTCLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=.9)
```

In [ ]:
```
def init_weights(m):
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.normal_(param.data, mean=0, std=0.1)
        else:
            nn.init.constant_(param.data, 0)
model.apply(init_weights)
```

In [ ]:
```
def train(loader, model, device):
    model.train()
    total_loss = 0.0
    num_batches = 0
    for i, (batch_input, batch_targs, batch_lengths) in enumerate(loader):
        print("processing batch {}".format(i))
        input_lengths, targ_lengths = list(zip(*batch_lengths))
        input_lengths = torch.tensor(input_lengths).to(device)
        targ_lengths = torch.tensor(targ_lengths).to(device)
        batch_input=batch_input.to(device)
        batch_targs= batch_targs.to(device)
        output = model(batch_input).to(device)
        cur_loss = criterion(output, batch_targs, input_lengths, targ_lengths
)
        total_loss += cur_loss.item()

        optimizer.zero_grad()

        cur_loss.backward()
        optimizer.step()

        num_batches += 1

    return total_loss / num_batches
```

In [ ]:
```
import Levenshtein as Lev

#-Q1 Comment
# Does the order of s1 s2 matter?
# What is levenshtein distance?

def wer (s1 , s2) :
    b = set(s1.split() + s2.split())
    word2char = dict(zip(b,range(len(b))))

    w1 = [chr(word2char[w] ) for w in s1.split()]
    w2 = [chr(word2char[w] ) for w in s2.split()]
    werlev = Lev.distance(" ".join(w1)," ".join(w2))
    werinst = float(werlev) / len(s1.split()) * 100
    return werinst


def cer(s1,s2) :
    s1,s2 = s1.replace(" " , ""),s2.replace(" ","")
    cerinst = float(Lev.distance(s1,s2))/len(s1) * 100
    return cerinst
```

In [ ]:
```
def evaluate(loader, model, device):
    model.eval()
    char_error, num_examples = 0.0, 0
    with torch.no_grad():
        for i, data in enumerate(loader):
            inputs, transcript, lengths = data
            inputs = inputs.to(device)
            transcript = transcript.to(device)
            model_outputs = model(inputs).to(device)
            input_lengths, targ_lengths = list(zip(*lengths))

            predicted_seq = best_path_decoding(model_outputs, input_lengths, i
ndex2char)


            for seq, gold_seq in zip(predicted_seq,transcript):
                gs = ""
                for l in gold_seq:
                    gs +=index2char[int(l)]
                char_error += cer(gs, seq)
            print(char_error)
            print(inputs.size(0))
            num_examples += inputs.size(0)

    return char_error / num_examples
```

In [ ]:
```
MAX_EPOCHS =20
for epoch in range(MAX_EPOCHS):
    train_loss = train(trainLoader, model, device)
    train_cer = evaluate(trainLoader,model, device)

    print('Epoch [{}/{}], Loss: {:.4f}, Training CER: {:.4f}'.format(epoch+1,
MAX_EPOCHS, train_loss, train_cer))
```