Homework 3 Part 2 (ver. 1.0)
Utterance to Phoneme Mapping

11-785: Introduction to Deep Learning (Fall 2022)

Out: **October 27, 2022, 11:59PM**
Early Deadline/MCQ Deadline: **November 3, 2022, 11:59PM**
Due: **November 17, 2022, 11:59PM**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with and work with other students on homework assignments.
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
  - You are allowed to help your friends debug
  - You are allowed to look at your friends code
  - You are allowed to copy math equations from any source that are not in code form
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - Meeting regularly with your study group to work together is highly encouraged. You may discuss ideas and help debug each other's code. You can even see from each other's solution what is effective, and what is ineffective. You can even "divide and conquer" to explore different strategies together before piecing together the most effective strategies. However, the actual code used to obtain the final submission must be entirely your own.

- **Overview:**
  - **Part 2**: This section of the homework is an open ended competition hosted on Kaggle.com, a popular service for hosting predictive modeling and data analytics competitions. **Automatic Speech Recognition (ASR):** Kaggle
  - **Part 2 Multiple Choice Questions**: You need to take a quiz before you start with HW3-Part 2. This quiz can be found on Canvas under **HW3P2-MCQ (Early deadline)**. It is **mandatory** to complete this quiz before the early deadline for HW3-Part 2.

Homework objective

After this homework, you would ideally have learned:

- To solve a sequence-to-sequence problem using Sequence models.
  - How to set up GRU/LSTM based models on pytorch
  - How to utilize CNNs as feature extractors
  - How to handle sequential data
  - How to pad / pack baches of variable length data
  - How to train the model using CTC Loss
  - How to optimize the model
  - How to implement and utilize decoders such as greedy and beam decoders
- To explore architectures and hyperparameters for the optimal solution
  - To identify and tabulate all the various design/architecture choices, parameters and hyperparameters that affect your solution
  - To devise strategies to search through this space of options to find the best solution
- The process of staging the exploration
  - To initially set up a simple solution that is easily implemented and optimized
  - To stage your data to efficiently search through the space of solutions
  - To subset promising configurations/settings and tune them to obtain higher performance
- To engineer the solution using your tools
  - To use objects from the PyTorch framework to build a GRU/LSTM based model
  - To deal with issues of data loading, memory usage, arithmetic precision etc. to maximize the time efficiency of your training and inference

1 Checklist

This homework is fairly straightforward, as such, below is a short checklist of things you can use to take inventory of your progress.

1. Write train and test (possibly val) dataset classes.
2. Build the basic LSTM based network with packing and padding
3. Write the Levenshtein distance function
4. Write training and evaluation loops
5. Write code for final inference and kaggle submission
6. Build a feature extractor into your model
7. Use more complex MLPs for classification
8. Win at life.

2 Introduction

Key new concepts:
CTC (Connectionist Temporal Classification)
Decoding Strategies (Beam Search)

Restrictions: You may not use any data besides that provided as part of this homework. Usage of any attention based techniques is not allowed.

2.1 Overview

This homework, much like HW1 works with speech data but in several ways, as we will discuss soon, is very different from it. This time we work with Automatic Speech Recognition (ASR) and we try to transcribe speech vectors into phonemes. What are phonemes you might ask, well, it's any **distinct unit of sound in speech** - or for the linguistics among you, a letter of the phonetic alphabet of a language.

If you recall, HW1 was what we referred to as frame-level speech recognition. What this really meant, to refresh, was that there was an output corresponding to every frame of the recorded speech, and once we had that mapping, our job was done. We didn't particularly care about whether the sequence of outputs we produced made sense or not. In HW3P2, however, **we do care**. Given some speech data, we now want to transcribe it into a sequence of phonemes and hopefully in such a manner that it makes sense. And this is where a fundamental problem that we deal with in this homework comes in - **lack of time synchrony**.

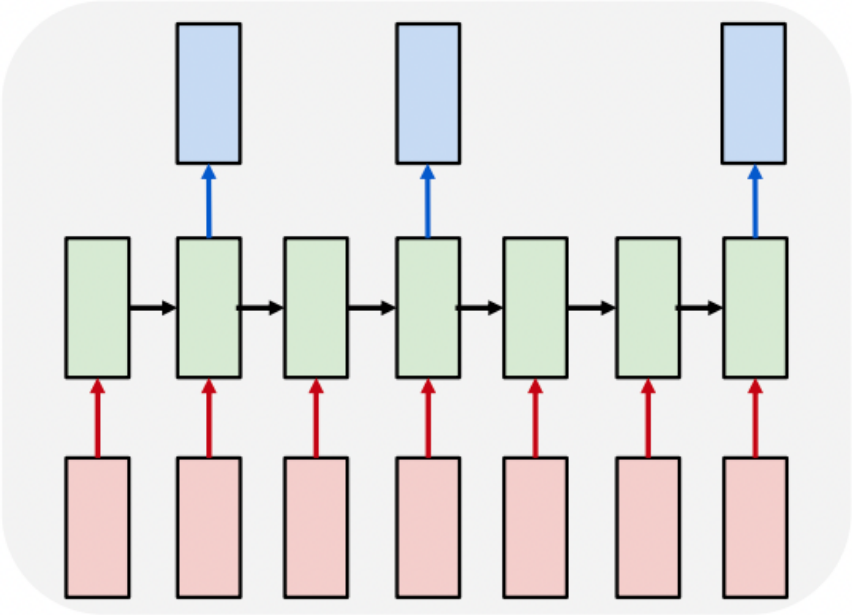

Figure 1: The Alignment Problem

2.2 Synchrony

Recall from lectures that we talked about two kinds of synchronizations in sequential data - order and time synchrony. We see **order synchrony** when a sequence of outputs is in the same order as the corresponding inputs, and we see time synchrony when the outputs occur after regular intervals of time. Let's give examples for a few scenarios:

- **Both Order and Time Synchrony**

  Think of a simple cipher which mapped a to e, b to f and so on till z to d. When I pass "Deep Learning" through this cipher, I get back "Ijju Qjewsnsl". This sort of a process has both order and time synchrony. The outputs are aligned with corresponding inputs in the same order and also occur at regular intervals (in this case, every timestep). In fact, your task in HW1P2 was also both order and time aligned.

- **Order Synchronous but no Time Synchrony**

  Think about playing a game in which you can use cheatcodes. Say you're playing on a console, and the remote only has so many buttons, and you use those buttons for controling the gameplay as well as entering cheatcodes. Let's say there's a code for jumping very high that goes UP UP LEFT RIGHT R1 R2, a code for summoning a car goes UP UP LEFT RIGHT R1 R2 L1 L2. If you, as the player, enter the code for summoning a car, the system needs to decide at every keypress whether to execute the gameplay action, execute the cheat corresponding to the past keypress sequence, or to wait for further keypresses. Though not exactly, this scenario captures some of the characteristics of a system of inputs and outputs having order but not time synchrony.

Along similar lines, now let's consider a scenario where you're given a recording of someone saying "..you've really got to *feel* it.." with a strong emphasis on 'feel' like one of those self-help ted talks. More over, you're given the recording in 100ms frames, and you're asked to write a phonetic transcription of the recording. All's fine till you get to the word *feel*. you hear the first frame, and think /f/, the next frame, /f/ again, next frame, /i:/, next, /i:/, next, /i:/, next, /i:/ - and so on for another 12 frames till you reach the /l/.

This is weird. /f//i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://i://l/ just seems silly. Clearly, only one of those /i:/'s are needed. So what do we do about the rest of them? This problem points us to the task we are trying to solve in HW3P2, though perhaps not quite as bizarre as this. We will further discuss the different aspects in the coming sections.

- **Neither Time nor Order Synchrony**

  Think of a machine translation task from English to Japanese. "The big red apple fell from the tree" gets (roughly) translated to "Ōkina akai ringo ga ki kara ochimashita" - which if you spend some time analyzing, you'll realize that the words are all jumbled up and there are fewer words in the translation than the source. It loses both time and order synchrony. This is a situation we will deal with in the next homework, so we can table this discussion for then.

2.3 Data Description

In this homework we aim to train a sequence to sequence model (seq2seq) that takes MFCCs (Recall Recitation 0J) as input and outputs the respective phonemes in it. Though our input is similar to HW1P2 and is time-stepped by frames, our output sequence will be a sequence of phonemes that **may or may not correspond to every frame**. This naturally keeps order synchrony but breaks time synchrony. The input data is 15 dimensional in terms of features and is variable in sequence length from one sequence to another.

2.4 The Alignment Problem

As we have alluded to before, the fact that there is a lack of time synchrony in our problem setup is a central issue that we need to deal with. More specifically, this situation leads to an alignment problem between the outputs and the inputs. As in, (Refer to lectures and Figure 1) there is no real relation between the timings of the input and the outputs. If there was any pattern whatsoever, we could have aligned the outputs with the inputs using that pattern, but unfortunately that isn't the case here. So how *do* we align our outputs with the input?

Now, it's important to remember that in a recurrent network (the kind of deep learning architecture we are using in this homework) each time-step is actually the same set of layers repeated over time. As such, the problem becomes, how do we make a model with this characteristic such that some times it gives an output, and sometimes it doesn't?

At this point we have thrown a bunch of questions your way without giving many answers. Let's start solving this problem now. Though there is no direct way to bake this irregularity into our recurrent net, we can take a short detour. We can covert this problem into one that we *can* and know how to solve. And we will do this using a magical BLANK symbol which we will represent with a hyphen '-' (The starter notebook refers to this symbol with the mapping `["" : " "]`). All you need to remember is that all symbols between two blanks will collapse to one, for example /F/ - /IY/ /IY/ /IY/ - /T/ gets collapsed to /F//IY//T/ for 'feet' (If this notation seems new to you, recall the /B//IY//F//IY/ example from lecture. From here on, we will follow this notation for phonetics). Essentially, the blank symbol marks the boundary between one phoneme and another. There is a nuance here that shouldn't be missed: There can be situations in which repeating phonemes need to be maintained and not collapsed.

We start by simplifying the problem to one that has time synchrony, we want our network to produce outputs at every time-step and now our possible set of output symbols also includes the /BLANK/ symbol. Now, all we want from the model is a set of probabilities corresponding to every symbol (including BLANK) at every time-step and we will leave the problem of *making sense out of this* to another component, a decoder. We will talk about the decoder shortly, for now all you need to understand is that we have converted an unknown problem of no time synchrony into a known problem and added a new symbol into the mix - BLANK. We have essentially circumvented the alignment problem, but we aren't done yet. We still don't know how to train this thing or how to arrive at an actual output sequence from this probability table (table, because it is 2D in time and symbols).

2.5 Decoding - From probabilities to phoneme sequences

Let's say we have a model that produces a vector of probabilities for every time-step of the input (we will get into the model specifics later), now we have two questions to answer; How do we convert this output table of probabilities into a sequence? and How do we train it to produce the (mostly) correct phoneme sequence? We will deal with these problems one at a time.

To answer the first question we can forget that there is a difference between training and inference, and simply devise a strategy to come up with a sequence of symbols from just probabilities - or - find an appropriate decoding strategy. For the purposes of this assignment, we will stick to two such methods: Greedy Decoding and Beam Search Decoding.

2.5.1 Greedy Decoding

As the name suggests, this strategy follows a simple process - choose the most likely phoneme at each time. As such we will go through our generated probability table and essentially $argmax$ along the phoneme probabilities to find the most likely phoneme at each time-step.

2.5.2 Beam Search Decoding

Although Greedy Search is easy to implement, it misses out on alignments that can lead to outputs with higher probability because it simply selects the most probable output at each time-step.

One way to deal with this issue would be to check all possible paths, and find the best one. But can you think of a problem with this? Yes, it becomes an exponential search. Can we strike some balance between these two extremes?

The answer is beam search. The main idea behind beam search is limiting the scope of your exhaustive exponential search by an upper bound of k "beams" or sequences. Though this does mean that there is a very real chance of missing the absolute best path, the sub-optimality is largely okay in this case when you consider just how much better it is compared to greedy decoding.

In HW3P1 you (hopefully) have implemented beam search from scratch (if not, you will when you attempt it), but we won't ask you to do that in P2. In fact as you'll find in the starter notebook and in the `Problem Specifics` section, you can use the decoder you get from the `CTC` library in python. Please do go through the docs for the module linked here: `https://pytorch.org/docs/stable/generated/torch.nn.CTCLoss.html`.

2.6 Connectionist Temporal Classification - How do we train this?

Had we had an output label corresponding to each frame of input, we could have easily used Cross Entropy loss over the model's output and the target. The loss computed could be sent back to update the model's parameters and train it. But we don't have such a mapping here. This is where Connectionist Temporal Classification (CTC) comes into play.

Before, diving straight into the specifics and properties of CTC, let's look at it what it means. Temporal classification is a task of classification of sequences (time series data) into given categories - in our case: phonemes.

The model that we intend to build outputs a vector of probabilities corresponding to each phoneme present in our dictionary of phonemes (including BLANK). Picking out the characters based on their probabilities at each time step, we create a condensed graph of all possible alignments. We then calculate the expected divergence by weighting the divergence between each possible alignment with the target sequence. With the probability of any alignment being the product of the probabilities of all the symbol sequences in its path, it becomes a horrendous task as there can be many possible alignments.

CTC hence uses the forward backward algorithm, a dynamic programming algorithm which computes the posterior marginals at each timestep for all possible alignments. The CTC loss function being differentiable with respect to the per time-step output probabilities, we can compute the gradient and update the model parameters through backpropagation.

2.7 Problem Specifics

2.7.1 Recurrent Neural Networks

In this homework you will work with PyTorch's variants of Recurrent Neural Networks (RNNs), such as Long-Short Term Memory (LSTM) Networks and Gated Recurrent Units (GRU) Networks. Please get familiar (preferably thorough) with the docs for each of these. It is particularly importance to understand what they take as input and outputs.

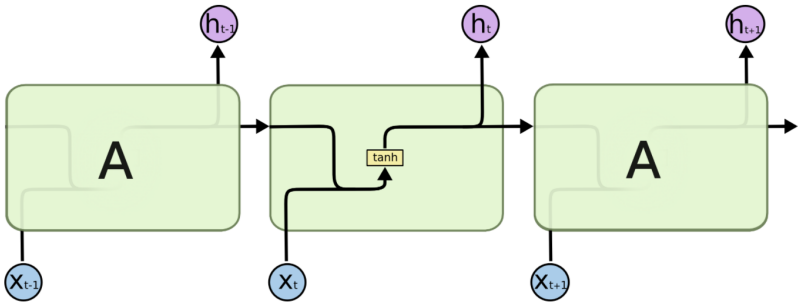

Figure 2: RNN Cells

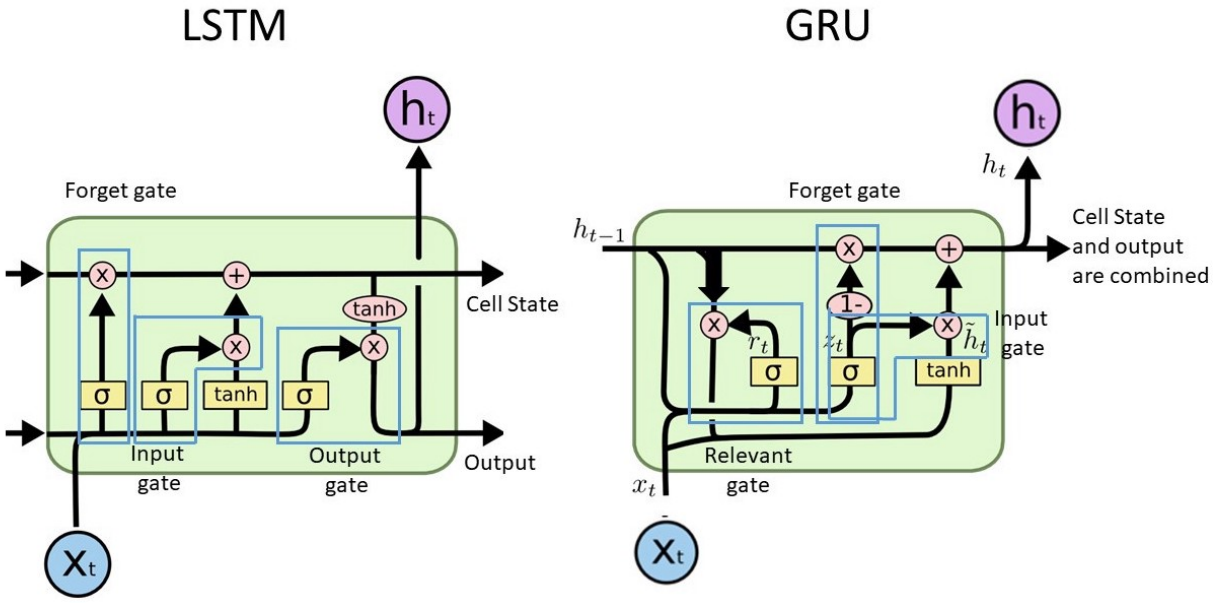

Figure 3: LSTM and GRU Cells

2.7.2 Feature Engineering

During this homework you will greatly benefit from using feature engineering, more specifically extraction, techniques to enrich the data going into the Recurrent network. What kind of networks can help you extract features from time-series data?

2.7.3 Padding and Packing

PyTorch gives us two functionalities that come in very handy when dealing with recurrent models:

1. pad_packed_sequence()
2. pack_padded_sequence()

Of the two, padding is rather simple to understand. Say in a batch, you have six sequences of variable lengths in the range 2 to 9. For batch processing to be possible, all six need to be of the same length, and as such, we (usually) pad all sequences to be the same length as the longest sequence (9 in our case). This can be visualized in figure (4).

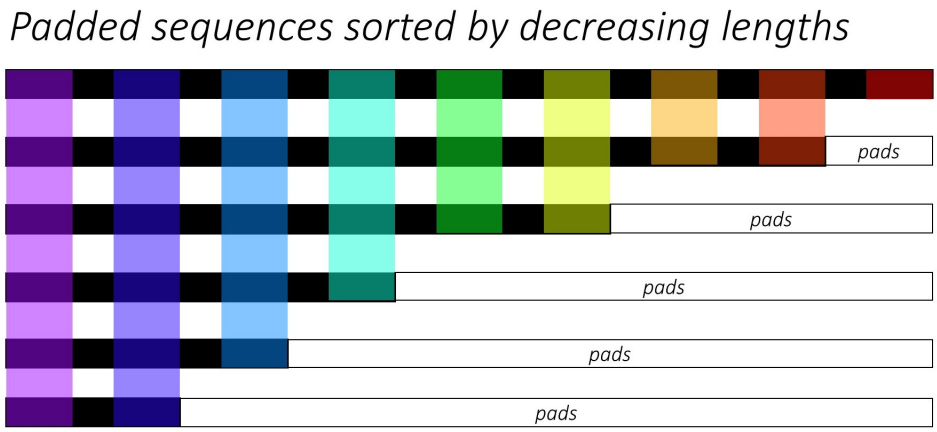

Figure 4: Batch with variable length sequences

Packing is the one that is not immediately intuitive. To understand why we do this, lets take the same example of a batch with variable length sequences as shown in figure (4). After padding, we perform 54 sets of computations, but only need 31. This is where packing comes in. After passing your padded sequence through the pack_padded_sequence() function, we get back a tuple of tensors with one containing containing the padded sequences and one containing the actual (ignoring the padding) batch size at each time step. This essentially allows PyTorch to optimize computation by flattening the sequences (sorted in order of sequence lengths) by timestep while keeping track of the *effective* batch-size at each time-step. This is shown in figure (5).

2.8 Why CTC? Does there not exist an alternative?

As explained above, CTC helps label unsegmented data sequences. There do exist alternative techniques where we pose the problem at a frame level. This can be achieved through sequence labelling using Hidden Markov Models.

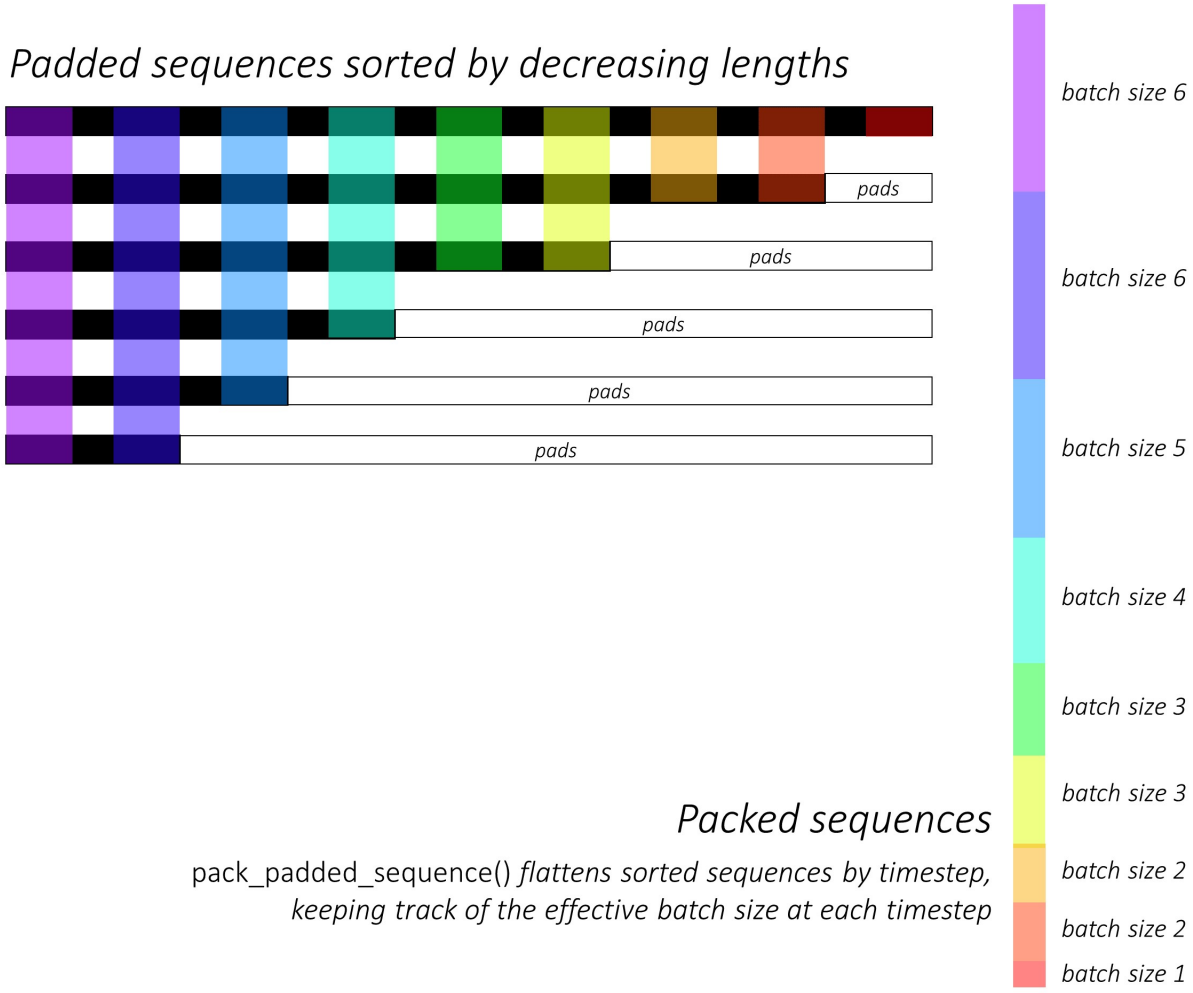

Figure 5: Packed padded sequence

The reason why posing the problem statement at a frame level feels like a mammoth is because we need a good understanding of the task, in order to design state models.
Note: Are there any other issues ?

2.9 Getting Started Early

As always, you will benefit greatly from starting early in this homework.

3 Dataset

Similar to HW1P2, you will be provided with mel-spectrograms that have 15 band frequencies for each time step of the speech data. However in this assignment, the labels will not have a direct mapping to each time step of your feature, instead they are simply the list of phonemes in the utterance [0-43]. There are 44 phoneme labels (including BLANK). The phoneme array will be as long as however many phonemes are in the utterance. We provide a look-up, mapping each phoneme to a single character for the purposes of this competition. (Refer to `phonetics.py` which is downloaded from Kaggle. **Note:** This mapping is slightly modified and overloaded using a cell in the starter notebook, please use the mapping in that cell.)

The feature data is an array of utterances, whose dimensions are `(frames,time step,15)`, and the labels will be of the dimension (frames, frequencies). The second dimension, viz., frequencies will have variable length which has no correlation to the time step dimension in feature data.

3.1 File Structure

- train-clean-360 : Full training set
- train-clean-100 : Small training set
- dev-clean : Dev (val) set
- test-clean : Testing set
- */mfcc/* : Same as HW1P2 for train, dev and test datasets
- */transcripts/raw/* : Contains sequence of labels which is not frame specific as in HW1P2.
- random_submission.csv: This is a random submission file that contains a randomized submission in the required format for kaggle.
- **ORDER**: This is important. In your test dataset, please load the files in sorted order of their names in the directory.

4 Implementation

Let's walk through in detail about all the functionalities to be implemented and how to tackle the problem.

4.1 Data Loaders

The data loader implemented in this homework would be similar to homework except for 2 small changes:

- We no longer require context.
- Padding sequences for processing inputs as batches would be achieved by functionalities provided by PyTorch: padding and packing.

Note: The data provided to us being of varying lengths, it is incumbent upon us to pad all sequences in a batch for use to the model. Padding all inputs would consolidate all input MFCC's to be of equal lengths. Hence, we must calculate and store the actual (unpadded) length of each recording before forwarding it to the model for use.

To achieve the above, we must accumulate all the sequences (features and target) in a batch and pad them first. This can be achieved using PyTorch's implementation of pad_sequence. Documentation: Pad Sequence

4.2 Model

4.2.1 Feature Extraction

We need to use convolutional neural networks to increase the number of features otherwise 15 in the MFCC's. The primary objective behind introducing these layers is to create rich feature embeddings of the data, from the otherwise 15 features.

A few things to keep in mind:
1. Make sure to use residual connections, if implementing a deep CNN architecture.
2. Down-sampling by increasing the stride can help reduce the number of time steps and fasten training.

4.2.2 Forward Pass - RNNs

In order to perform operations in batch, we must and pack and unpack the data every time when passing data through an RNN.

- pack_padded_sequence:
  - Input Parameters:
    1. Data Sequence
    2. Lengths of each data sequence (as a list)

    Example:
    packed = pack_padded_sequence(recordings, length_of_each_recording, batch_first=True, enforce_sorted=False)

- Output: PackedSequence(data=tensor([]), batch_sizes=tensor([]), sorted_indices=tensor([]), unsorted_indices=tensor([]))

- pad_packed_sequence:
  - Input Parameters:
    1. Takes in a Packed data sequence as input

    Example: seq_unpacked, lens_unpacked = pad_packed_sequence(packed, batch_first=True)
  - Output Parameters:
    1. Original data sequence
    2. Actual (unpadded) length of each sequence

The primary purpose behind the above 2 functions is that RNNs use a Packed Sequence object as input. It performs all computations required for each input at a given time-step before moving onto the next, enabling faster processing.

It is important to pack a paded sequence before passing it through an RNN as well as pad a packed sequence after computation from the RNN.

4.3 Training

In order to train any model, we must calculate the loss based on a criterion. To calculate the loss, we need the predictions from the model. We will use beam search to decode the phonemes from the model, and CTC Loss as our criterion to calculate loss.

4.3.1 CTC Loss

As described above, there is no alignment between utterances and their corresponding phonemes. Thus, train your network using CTC loss. Decode your predictions, preferably using beam search. Use the list of phonemes provided on the data page to make each prediction into a text string.

In Pytorch, you can use nn.CTCLoss.
Note: Take a close look at the documentation to determine the input shape and format required to compute CTC Loss

4.3.2 CTC Decoding

You can manually install the library, ctcdecode while working with PyTorch. They have an implementation of beam search, use it in your code to decode the output of your model.

You have already implemented Beam Search in BeamSearch.py in your part-1, you can use that implementation here. The Beam Search implementation of part-1 outputs 2 arguments one of which is the best sequence path which can be used to predict your sequence of phoneme.

Note: Increasing the beam width can exponentially increase the training time. Please stick to a smaller beam width (3) while training. You may increase the beam width while making predictions.

4.3.3 Model Evaluation

To check the model performance, we will be resorting to Levenshtein Distance. In order to calculate the Levenshtein distance, we will make use of this library.

Levenshtein Distance takes as input two strings. The distance computed is the number of character differences between the strings. The number of character differences is calculated in an order synchronous fashion.

Levenshtein Distance is calculated as such:
Levenshtein.distance(target, prediction)

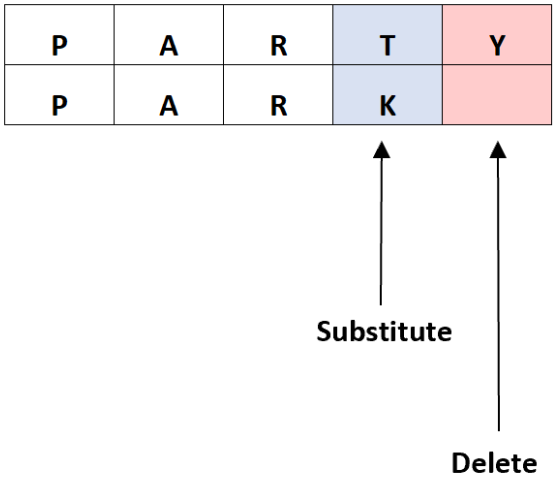

Figure 6: Calculation of Levenshtein Distance

The Levenshtein distance in this case = 2

5 Submission

5.1 Preliminary Submission

There is a mandatory preliminary submission and an associated MCQ that, together, are worth 10% of the points for the homework. The deadline for this preliminary submission is **November 3, 2022, 11:59PM.** This submission is intended to get you started quickly on the homework. We have provided a HW3P2 starter to help you with the preliminary

submission. You can download the starter from piazza.

Disclaimer: The starter notebook is not as elaborate as the previous homeworks. You will have to code most of the implementations yourself. This is because, after 2 homeworks we expect you to be in a position to write your own notebook from scratch. You can reuse the code from previous starter notebooks if you want. Completing the starter notebook will take you time, hence please start early.

5.2 Final Submission

You will be evaluated using Kaggle's character-level string edit distance. Since we mapped each phoneme to a single character, that means you are being evaluated on phoneme edit-distance.

We are using Levenshtein distance, which counts how many additions, deletions and modifications are required to make one sequence into another.

Your submission should be a CSV file. The headers should be "id", and "predictions" - id refers to the 0-based index of utterance in the test set and predictions is the phoneme string. Please note that the headers are case sensitive.

See sample submission for details.

6 Conclusion

That's all. As always, feel free to ask on Piazza if you have any questions.

Glhf!
Psst, don't know what this means? Haha, boomer.

Appendix

The data for this homework is similar to that of HW1P2. For a more detailed description of the data refer to the HW1P2 write-up.