Homework 3 Part 2
Utterance to Phoneme Mapping

11-785: Introduction to Deep Learning (Spring 2023)

Out: **March 17, 2023, 11:59PM**
Early Deadline/MCQ Deadline: **March 25, 2023, 11:59PM**
Due: **April 7, 2023, 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.

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
  - How to identify and tabulate all the various design/architecture choices, parameters and hyperparameters that affect your solution
  - How to devise strategies to search through this space of options to find the best solution
- The process of staging the exploration
  - How to initially set up a simple solution that is easily implemented and optimized
  - How to stage your data to efficiently search through the space of solutions
  - How to subset promising configurations/settings and tune them to obtain higher performance
- To engineer the solution using your tools
  - How to use objects from the PyTorch framework to build a GRU/LSTM based model
  - How to deal with issues of data loading, memory usage, arithmetic precision etc. to maximize the time efficiency of your training and inference

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.

Multiple Choice Questions

You need to take a quiz before you start with HW3-Part 2, which 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.

Contents

1 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.
- **Automatic Speech Recognition (ASR):** Kaggle Competition
- **Data Set:** The data for 3p2 includes a list of 40 phonemes from English, a training set, a validation set, and a test set. The training and validation data comprise (sequences of feature vectors derived from) speech recordings, and the sequence of phonemes representing the transcription for each utterance. The transcriptions are not time-aligned to the speech feature vectors. You must use these to train your model. The test data only comprise speech recordings. You must use your train model to derive transcriptions for them.

1.1 Overview

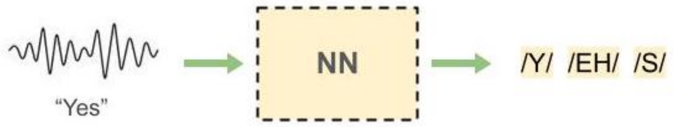


Figure 1: Audio to phonemes

In homework 3p2, we return to the topic of speech recognition by predicting the phonemes in a recording. Figure 1 illustrates the task at hand, where an audio recording of a person saying the word "yes" has been passed though a neural network, which outputs the phonetic transcription, /Y/ /EH/ /S/. In this assignment, you will be implementing RNNs and the dynamic programming algorithm, Connectionist Temporal Classification, to generate such labels.

The kind of output labels you must generate in this homework is somewhat in contrast to what we did in HW1P2, which too addressed the problem of speech recognition. In both homeworks the speech recordings are parametrized as a sequence of feature vectors (mel spectral vectors, each representing a "frame" of speech), which arrive at a rate of 100 frames per second. In HW1P2, we labeled each of these frames independently – the classification of one frame was not affected in any manner by the classification of any other frame. Consequently, the speech was labeled at a rate of 100 labels per second. In light of homework 1, if in our above example, if the word "yes" took a quarter second to utter, we would have derived 25 vectors from the recording, and our actual labeling scheme would have resulted in the label sequence /Y/ /Y/ $\cdots$ /Y/ /EH/ /EH/ $\cdots$ /EH/ /S/ $\cdots$ /S/ (with a total of 25 such labels in all). It is rather unnatural to tag *every* frame of the signal in this manner – if you were asked to label the phonemes in the recording yourself, you would be very unlikely to label it in the above manner, with a sequence of 25 phonetic symbols. Instead, you are far more likely to simply label it as /Y/ /EH/ /S/. Not only is this the more natural way of labeling the recording, it also does not require you to address the rather challenging problem of knowing exactly where one phoneme ended, and where the next one began.

In 3p2, we want you to generate more natural predictions, regardless of how words have been articulated in the speech recordings, so that the output of such an example would be /Y/ /EH/ /S/.

Figure 2 illustrates the problem. A sequence of feature vectors derived from a speech recording goes into the network. The network must output the sequence of phoneme labels for the recording.

The challenge inherent in this problem becomes apparent immediately. The output sequence, /Y/ /EH/ /S/ is shorter in length than the input sequence, and doesn't have a one-to-one correspondence with it. Thus, while the output is *order-aligned* with the input (i.e. since /Y/ occurs before /EH/ in the output, the portion of the input corresponding to /Y/ also occurs before the portion of the input corresponding to /EH/), it is is not *time-synchronous* (i.e. in one-to-one correspondence) with it. Also, it isn't immediately apparent *when* (i.e. at what times) the the symbols must be output. Thus, we must figure out how to output an order-aligned time-asynchronous label sequence that best represents the input.

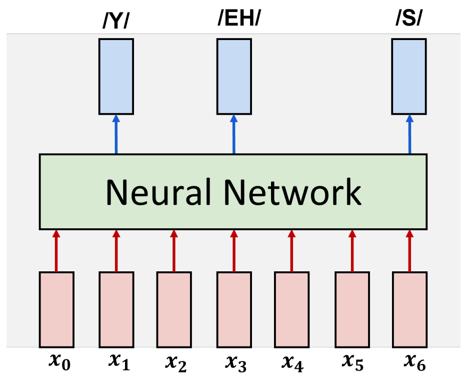


Figure 2: The network is required to take in a sequence of feature vectors ($x_1, \cdots, x_6$ in our illustration) and output a sequence of phonemes (/Y/ /EH/ /S/ in our illustration)..

Addressing this order-aligned time-asynchrony of the output is central to this homework. We must determine how to design a neural network for such inference, how to train it, and how to actually perform inference with it. We outline the solution and associated challenges below.

1.2 An Alignment Problem of Inputs and Outputs

Speech recognition is an instance of a *sequence-to-sequence* conversion (or transduction), the problem of converting (or transducing) input sequences to output sequences. Examples include machine translation, where a word sequence in one language is converted to a word sequence in another language, dialog systems, where the word sequence in a query is transduced to the word sequence in the response, and, of course speech recognition, where the sequence of vectors in the speech input is converted to the sequences of words or phonemes in the output. Sequence-to-sequence *models* are neural networks that perform sequence-to-sequence conversion. In our problem, speech recognition, the output sequence is order-synchronous with the input, though it may not be time-synchronous. Since the speech feature vectors from a recording actually form a time series, and since this homework is an exercise on *recurrent* neural networks, the neural network is an RNN.

Figure 3 re-illustrates the problem of Figure 2, explicitly showing the recurrent nature of the neural network. The input to the model is a sequence of feature vectors (shown by the orange boxes) from a speech recording. The network must analyze the input sequence and generate the output phoneme sequence (e.g. "/Y/ /EH/ /S/") that best matches the audio.

The issue is that the output is sporadic, not time-synchronous. It is unknown *a priori* which phonemes occur in the output, the length of the output sequence, or even *when* to output the phonemes the recording. How, then, do we deal with this problem?

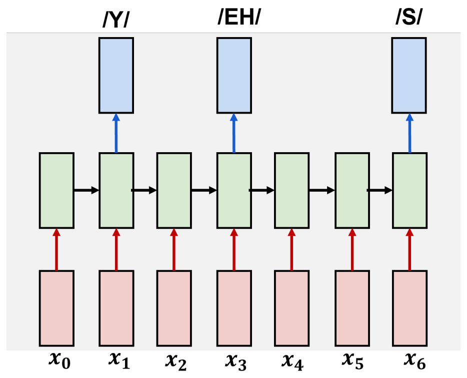


Figure 3: Illustration of non-time-synchronous output. The sequence of speech feature vectors (orange boxes) corresponds to the word "yes", pronounced as the phoneme sequence "/Y/ /EH/ /S/". The model must output the identities of each of the phonemes at the termination of the phoneme.

We tackle the problem by decomposing the inference into a two-step process. In the first step, a neural network generates outputs at *every* time step. This eliminates the uncertainty of knowing when to generate the (sporadic) outputs. In the second stage, we perform a dynamic programming (DP) search-like operation on the complete set of outputs generated by the network, to generate the actual final output. Figure 4 illustrates the process. The entire framework, including both components, is known as the "Connectionist Temporal Classification" framework or CTC (although the term CTC is sometimes also used just to refer to the DP component).

The problem of deriving an asynchronous symbol sequence from the time-synchronous output of the neural network is now transferred to the DP module. As it turns out, this is a simpler problem than having the network directly generate sporadic outputs.

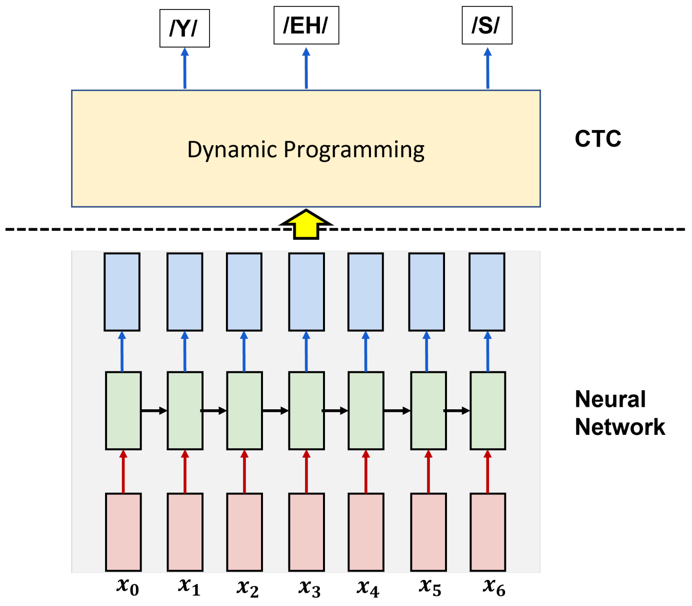


Figure 4: Solving the asynchronicity problem through a two-stage solution: a) the neural network computes outputs at *every* time step, b) a dynamic-programming block analyzes the output of the network to determine the actual final time-asynchronous output.

The key to the problem is the realization that the neural network's actual output at each time is, in fact, the vector of *probabilities* for all the symbols (phonemes in this problem). The final output layer of the network is a softmax layer that computes the probabilities for each of the symbols $P(AH)$, $P(AE)$, $P(B)$ etc at each time. This is illustrated in Figure 5. Thus the DP must, in fact, operate on these probabilities.

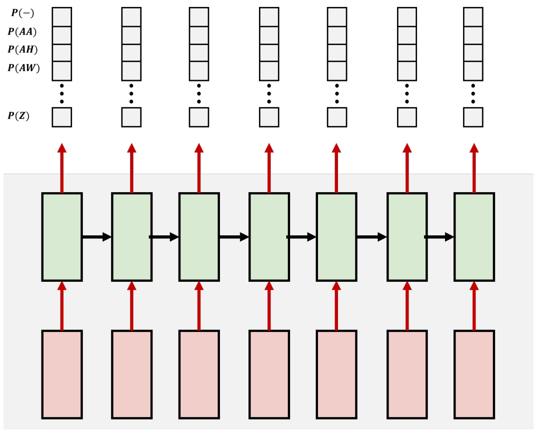
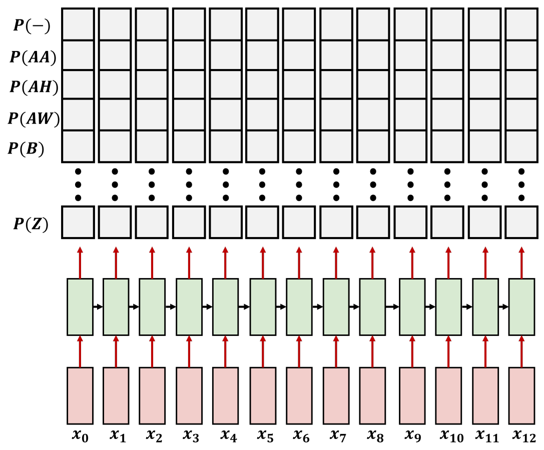


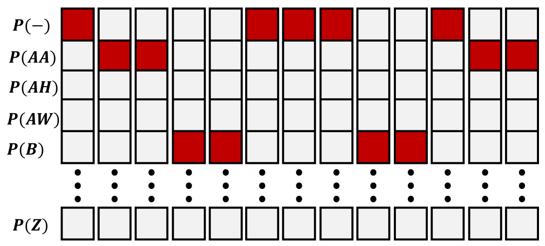
Figure 5: At each time the network actually outputs a vector of *probabilities* for the symbols (classes) in our "vocabulary".

A very simple solution for order-synchronous, time-asynchronous inference is shown in Figure 6. The neural network derives the set of symbol probabilities for each input vector. The DP stage simply greedily selects the most probable symbol at each time, and subsequently "squeezes out" all repetitions of a symbol, to derive the final asynchronous output.



$P(-)$
$P(AA)$
$P(AH)$
$P(AW)$
$P(B)$
$P(Z)$
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}

(a) At each time the network actually outputs a vector of *probabilities* for the symbols (classes) in our "vocabulary". The figure shows the generation of probabilites for a sequence of 12 input vectors.



(b) Simply selecting the most probable symbol at each time (shown by the red box) results in a symbol sequence ("- /AA/ /AA/ /B/ /B/ - - - /B/ /B/ - /AA/ /AA/" in our example). "Squeezing" out repetitions results in a compressed sequence ("- /AA/ /B/ - /B/ - /AA/" in our example), and subsequently removing blanks from this compressed sequence gives us our result ("/AA/ /B/ /B/ /AA/" in our example).

Figure 6: Simple two-stage solution, including (a) the derivation of probabilities from the neural network, followed by (b) a greedy search over the probabilities and compression of the resulting sequence, to obtain the asynchronous ouptput.

A critical component of this framework is the "blank" symbol, which must be included as part of the symbol set output by the network (even though it is not actually a part of the phoneme list for the language). The blank symbol represents an *invisible*, but real symbol which behaves like any other symbol, but is removed from the final output sequence (because it is invisible).

The greedy selection approach originally results in a sequence of symbols with many repetitions. For instance, in the example of Figure 6, the greedy output is "- /AA/ /AA/ /B/ /B/ - - - /B/ /B/ - /AA/ /AA/". Eliminating repetitions gives us "- /AA/ /B/ - /B/ - /AA/". The blank symbol must be removed from this sequence to obtain the final output "/AA/ /B/ /B/ /AA/". The need for the blank symbol becomes apparent from this example: if we had not had the blank as part of the network's vocabulary, then the greedy output would have been something like "/AA/ /AA/ /AA/ /B/ /B/ /B/ /B/ /B/ /B/ /B/ /AA/ /AA/ /AA/" (there would be no blanks). Compressing it would simply produce an "/AA/ /B/ /AA/", because the entire sequence of /B/s would compress to a single /B/ – there is no other unambiguous way of compressing the sequence of /B/s. As a result, we could not obtain genuine repetitions of a symbol, like the two /B/s in /AA/ /B/ /B/ /AA/. The blank symbol resolves this problem: it enables us to generate genuine repetitions by separating out the repetitions by inserting some blanks between them. In fact, even if our inference mechanism is not greedy, the blanks remain critical for us to obtain unambiguous asynchronous output.

1.3 The challenge tasks

The actual solution to the homework must address three challenges:

1. **The structure of the neural network that computes the symbol probabilities at each time, and how to train it**

   Since this is a homework focused on recurrent neural networks, and because speech is a signal with distinct temporal structure, our neural networks must be based on recurrent architectures. This is the only requirement imposed on you; various variations on the theme described in Section 2.1 may be considered within this framework.

2. **How to set up the DP such that losses can be computed from it in a manner that is differentiable, and permits loss derivatives to be propagated to the neural network**

   The actual output is produced after performing DP on the output of the network, and not directly from the network. Losses are computed on the output of the DP module. We must determine how to propagate their derivatives back through the DP block into the network to train it.

3. **How to perform actual inference using DP. (The simple greedy approach shown above is just one solution; how do we improve on it?)**

   The greedy algorithm is only one possible approach to obtaining an asynchronous output from the network, and may not even generate the *most probable* asynchronous output sequence for an input. Better results may be obtained through inference techniques that attempt to find the symbol sequence with the maximum total probability; the *beam search* algorithm is one such. We will also investigate these.

Without further ado, let us dive into the problem.

2 Problem Specifics

For this assignment, you will be transcribing (converting) each speech recording into a sequence of phonemes from the list below. You will be evaluated on the accuracy of the transcriptions you generate.

```
"[SIL]", "NG", "F", "M", "AE", "R", "UW", "N", "IY", "AW", "V", "UH", "OW", "AA",
"ER", "HH", "Z", "K", "CH", "W", "EY", "ZH", "T", "EH", "Y", "AH", "B", "P", "TH",
"DH", "AO", "G", "L", "JH", "OY", "SH", "D", "AY", "S", "IH"
```

This list of 41 phonemes (40 actual phonemes and the `[SIL]` symbol representing silence) is not in alphabetic order, and needs to be alphabetized and made into a dictionary. The list of phonemes also does not include the BLANK symbol – you must explicitly include it to this list to complete it. Note that `[SIL]` is a distinct symbol representing silence, and is *not* a

BLANK – the BLANK is distinct from `[SIL]`. Including the BLANK, you will have a list of 42 symbols (41 phonemes and the BLANK). Also, ***after rearranging the phonemes into alphabetic order, introduce BLANK at the top of the list, at the 0 index***. Some of the routines we use later in this homework assume that BLANK is the zeroth symbol.

Thus, your model will be a recurrent neural network that produces 42 outputs at each time.

In the following sections, we will discuss optimal neural network structures for the task at hand, how this model should be trained, and how to perform inference.

2.1 The Neural Network

As mentioned earlier, the neural network must be a recurrent network. This is the only requirement imposed on you; however we may consider any of the following in designing the actual network:

- As explained in the lecture, long-term memory is best retained by LSTMs (or LSTM-like structures, such as GRUs), so our recurrent models may be based on LSTMs or GRUs.
- For this homework we will assume that the *entire* speech recording is available before we attempt recognition (i.e. we are not restricted to a “streaming” setup where speech must be recognized as it arrives), so our recurrent networks can, in fact be *bidirectional*.
- Also, although the explanation in the previous section assumed that symbol probabilities are computed for *every* input vector, this may not in fact be required, since the asynchronous output symbol sequence typically has far fewer symbols than the number of vectors in the input. This means we could potentially *downsample* the input sequence within the network to reduce its length by a uniform factor.

Other options may also be explored. Within these confines, let us consider how a typical neural network may be setup.

The model will take in an input sequence of vectors $X_0, \cdots, X_{N-1}$ and produce a sequence of probability vectors. The input sequence of speech feature vectors exhibits both short-term structural relations, and long-term contextual dependence. The model must try to capture these dependencies. It can consist of one or more the following components:

- 1-D CNN layer. 1D CNNs capture the structural dependence between adjacent vectors in the input. They may also have a stride greater than 1 (usually 2) in order to reduce the feature rate by a factor equal to the stride.
- Bidirectional LSTM layers (Bi-LSTMS) to capture long-term contextual dependencies.
- A new variant of Bi-LSTMs called *pyramidal Bi-LSTMs* or pBLSTMs. pBLSTMs, like CNNs with stride=2, reduce the time-resolution of the input by a factor of 2. See Appendix A for a detailed description of pBLSTMs.

Here is a typical RNN model for speech recognition. It includes a pBLSTM for illustrative purposes. For purposes of convenience (and as a reference for a future homework – HW4P2),

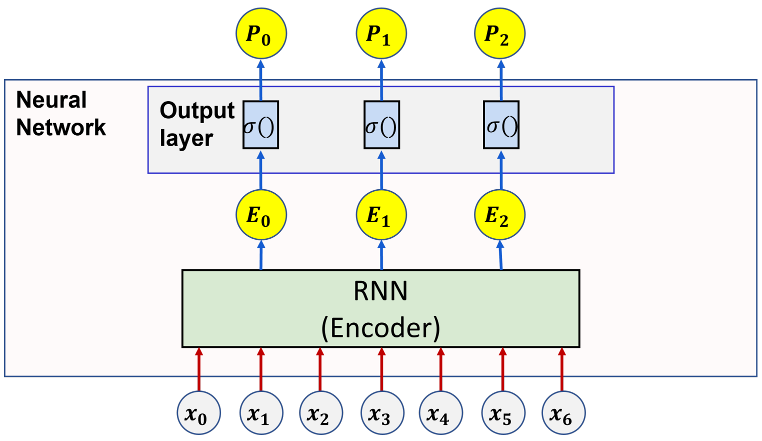


Figure 7: The neural network comprises two distinct components – an *encoder*, which converts the sequence of inputs $x_0, \cdots, x_N$ into a sequence of latent *embeddings* $E_0, \cdots, E_N$, and an output probability layer, comprising a linear layer followed by a softmax, which is applied separately to each embedding to generate a probability vector P_t at each time t.

we have decomposed it into two parts, as shown in Figure 7. The first part of the network is an "encoder", which takes in the input sequence and produces a sequence of *embeddings* – a sequence of vectors that represent "latent" features derived from the input. Comparing this to a conventional RNN, the encoder is just the portion of the network until the final output layer (i.e. until the penultimate layer). Note that if the encoder includes downsampling layers (e.g. CNNs with stride greater than 1, or pBLSTM layers), the number of embedding vectors output by it will be proportionally smaller than the length of the input sequence.

These embeddings are then passed to the output layer (which comprises a linear layer followed by a softmax *at each time*, to generate a probability vector at each time.

The Encoder: A typical instance of an encoder comprises one or more initial layers of 1D CNNs, followed by one or more layers of bidirectional LSTMs (Bi-LSTMs), and finally by one or more layers of pBLSTMs. Note that each pBLSTM layer reduces the time resolution by a factor of 2. The output of the final layer of the network is the sequence of embeddings.

Here is pseudocode for an example encoder that reduces the rate by a factor of 2. This is only an example and the details of how you implement the model up to you. You must devise your own architecture, and each of the modules (BiLSTM, pBLSTM, optionally CNNs or ResNets, and the overall encoder) will be classes.

```
# X = (batchsize, length, dim) array, which is a minibatch of input sequences
# The output is minibatch of embedding sequences (for the entire input
   minibatch).
function E = Encoder(X)
    O1 = 1DCNN(X, stride=1, CNNparams)
    O2 = BiLSTM(O1, BLSTMwidth, BLSTMparams)
    E = pBLSTM(O2, pBLSTMwidth, pBLSTMparams) # Will reduce length by 2
    return E
end
```

Listing 1 Encoder

There are many variants you could try. The 1DCNNs could be replaced by ResNets. Also, the example code only downsamples the input by 2. To downsample it further you could either include strides in CNN layers or have more pBLSTM layers. (As mentioned in the introduction, the pBLSTM is optional for this homework.)

You can follow these links to find Pytorch's implementation of LSTMs and GRUs.

Overall network: The overall network passes the embeddings to a final output layer to compute a sequence of probability vectors. The overeall pseudocode might look as follows:

```
# X = (batchsize, length, dim) array, which is a minibatch of input sequences
# The output is minibatch of probability sequences (for the entire input
   minibatch).
function LogP = NetworkComputeLogProbabilityTable(X)
   E = Encoder(X)
   Z = linear(E)
   LogP = LogSoftmax(Z)
   return LogP
end
```

Listing 2 Overall network

Here the linear layer is *applied individually to each vector in the sequence of embeddings E*. Thus Z is a (batchsize, length, nsymbol) array for the minibatch. In theory, the softmax is applied to each vector in Z to compute a probability vector. In practice, we usually compute a *log*softmax (the log of the softmax), because (1) the max probability entry stays the same; (2) the log operation turns multiplication into addition, which is more numerically stable; (3) pytorch CTCLoss expects log softmax probabilities as input.

Thus, the network results in a (batchsize, length, nsymbol) array representing a batchsize $\times$ length array of log-probability vectors.

2.2 Training the network

The training data provided to train the network will comprise speech recordings, and their transcriptions. In data terms, each training instance will comprise the pair $(x_0, x_1, \cdots, x_N, /PH_1/, \cdots, /PH_K/)$, where $x_0, x_1, \cdots, x_N$ represents the sequence of feaure vectors derived from the speech signal, and $/PH_1/, \cdots, /PH_K/$ represents the phoneme sequence for the recording (e.g. /Y/ /EH/ /S/). We must train the network from these training data.

As mentioned earlier, the actual phoneme sequence for any input speech may be much shorter than the input itself and is not time-synchronous with it, although it is order-aligned. To deal with this asynchrony we use the two-stage solution of Section 1.2, where we decompose the problem by having the network output a sequence of probability vectors in time-synchrony (possibly downsampled) with the input, and use a DP framework to derive the actual outputs from these probabilities.

While the output of the system is now no longer time-synchronous with the input, the *network itself continues to produce time-synchronous* (though possibly downsampled) outputs (which are probability vectors). In order to train the network, we will require a *target* corresponding to *every* probability vector output by the network, as illustrated by Figure 8.

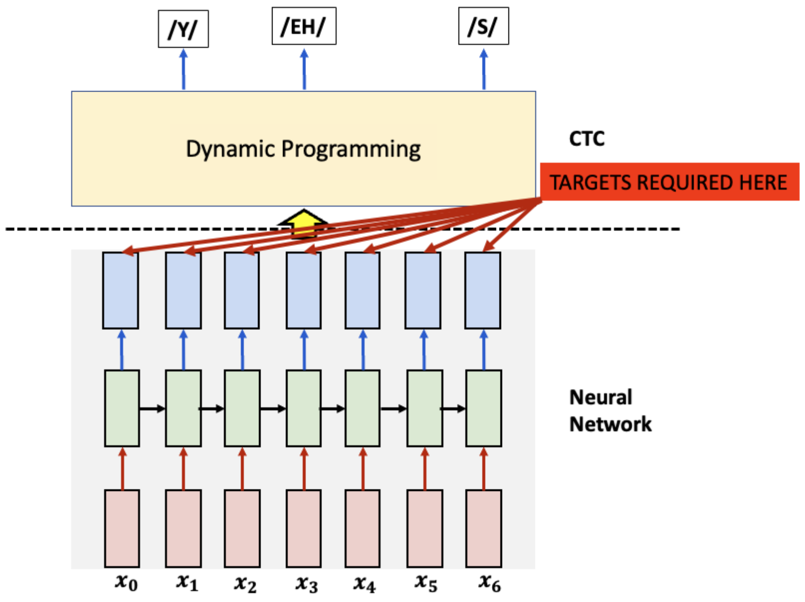


Figure 8: On training data, only the final output label sequence, e.g. /Y/ /EH/ /S/ is provided. These are what are expected to be output by the DP module that operates on the output of the network. However, to *train* the network, targets are required corresponding to *every* output by the network, as shown by the arrows.

In other words, we require an output label corresponding to each probability vector output by the network. Had we had such a time-synchronous target label sequence, we could have easily used Cross Entropy loss over the model's output and the target, and this loss could have been sent back to update the model's parameters for training. However the actual label sequence we have is compressed and not time-synchronous, although it is order-aligned.

Hence, in order to compute the the loss, we must *expand* our label sequence to align it to the input. For instance, consider a training recording that has been parameterized into a sequence of (for illustration) twelve feature vectors $x_0, \cdots, x_{11}$, and is given the label sequence /HH/ /AY/. Consider that our encoder downsamples by 2 to result in generate six time-synchronous probability vectors[1] $P_0, \cdots, P_5$. The label sequence /HH/ /AY/ must now be expanded to length 6, such as to /HH/ /HH/ /HH/ $-$ /AY/ /AY/, to match the output sequence of probability vectors from the neural network. Note that in expanding the

[1]The encoder will actually produce six time-synchronous embeddings, from which we will obtain six time-synchronous probability vectors

$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
/HH/	/HH/	/HH/	/HH/	/HH/	/AY/
–	/HH/	/HH/	/HH/	/HH/	/AY/
–	–	/HH/	/HH/	/HH/	/AY/
–	–	/HH/	/HH/	/AY/	–
/HH/	/HH/	/HH/	/HH/	–	/AY/
/HH/	/HH/	/HH/	–	/AY/	/AY/
/HH/	/HH/	–	–/	/AY/	/AY/
–	/HH/	/HH/	–	–	/AY/
–	/HH/	–	/AY/	–	–

Table 1: Examples of ways of aligning /HH/ /AY/ to six time steps. Any of these could be used as the target sequence for Figure 8.

label sequence to the desired length, we must also consider the blank symbol (shown as – here).

The way to expand the label sequence to match the length of the input is not unique. Table 1 shows just some of the ways in which /HH/ /AY/ can be aligned against six outputs from the encoder.

It is clear from the above example that there can be a *very large* number of ways to align the phoneme sequence to the input, any one of which could potentially be used as the target sequence. Which of them do we choose, to train our network? We have a couple of solutions – to choose the most probable (using the Viterbi algorithm, as discussed in class), or to simply consider *all* of them.

2.2.1 *Viterbi* training: Training with the best alignment

In the first approach, which we call “Viterbi training”, we first find the *most likely* alignment of the target phoneme sequence to the input. This can be done using the Viterbi algorithm. We have encountered the Viterbi algorithm in class. Pseudocode for it can be found in Appendix B.

Once the best alignment is found, we can use that with a standard cross-entropy loss. The pseudocode below shows how you could compute the loss for a single training sequence. We leave the generalization to a minibatch to you.

```
# X = (length, dim) array, which is a input sequence
# Y = (labellength) array, which is the label corresponding to X
# Performs a forward pass, aligns the labels, and computes the loss
function L = Viterbi(X, Y)
    LogP = NetworkComputeLogProbabilityTable(X)
    Target = ViterbiAlign(LogP, Y)
    L = XentLoss(LogP, Target)
end
```

Listing 3 Viterbi Training

Note that the ViterbiAlign expects log probabilities, rather than probabilities. The pseudocode above reflects this. ViterbiAlign also expects BLANK as the first symbol (0th index) in the symbol (phoneme) list.

2.2.2 Training using *all* alignments

Our preferred approach is the second approach: instead of choosing any specific alignment, we compute the loss against *every one* of these alignments, and the overall loss is a weighted sum of all these losses, where the weight for any alignment is the probability of that alignment. In probability terms, our loss is the *expected loss* over all possible alignments.

There are an exponential number of possible alignments, so explicitly computing the losses for all of them, and summing them up is not going to be possible. However, we can do so efficiently using a dynamic programming method, namely the forward-backward algorithm, as we saw in class.

You can implement the forward-backward algorithm yourself. However, fortunately, pytorch already provides this as a standard loss called CTCloss. In Pytorch, you can use nn.CTCLoss. Pay close attention to the documentation to determine the input shape and format required to compute CTC Loss.

CTCLoss computes the expected loss over all possible alignments. The CTC loss function is differentiable with respect to the per time-step output probabilities (i.e. w.r.t. each of P_0, P_1, $P_2, \cdots$), and can hence compute the gradients w.r.t these, and subsequently used to update the model parameters through backpropagation.

The overall code for computing the CTC Loss might look like this:

```
# X = (batchsize, length, dim) array, which is a minibatch of input sequences
# Y = (batchsize, labellength) array, which is the labels corresponding to X
# Performs a forward pass and computes the CTC Loss
function L = NetworkCTCLoss(X, Y)
    LogP = NetworkComputeLogProbabilityTable(X)
    L = CTCLoss(LogP, Y)
end
```

Listing 4 Training with CTC Loss

In the pseudocode above, as mentioned before, we are using log softmax to match the expected input of pytorch CTC loss. In addition, CTC Loss expects an argument for the index of BLANK in the symbol (phoneme) list.

After calculating either the Viterbi loss or the CTC loss, we can perform the backward pass and get the gradients using `L.backward()`.

2.3 Inference

Once we have a model that produces a vector of probabilities for every time-step of the input, we ask the following question: How do we best convert this output table of probabilities into a sequence?

To come up with a sequence of symbols, there are a couple of strategies.

2.3.1 Greedy Search

Greedy Search is the technique illustrated in Figure 6b. It is easy to implement and selects the most probable output at each time-step. Essentially, this strategy will go through our generated probability table and *argmax* along the phoneme probabilities at each time, and finally compress the output squeezing out repetitions and removing the blanks. The pseudocode below explains greedy decoding for a single test instance.

```
# X = (length, dim) array, which is a minibatch of input sequences
# symbols = dictionary of phonemes. Must include BLANK
# Performs forward pass to compute probability table, followed by greedy decode.
function O = GreedyDecode(X, symbols)
    # Returns a length x nclass probability table, where nclass includes the blank
    P = NetworkComputeProbabilityTable(X)
    # ArgMax returns length x 1 array of argmaxes of each probability vector in P
    A = ArgMax(P, dim=1)
    O = []
    # Eliminate repetitions and remove blanks.
    for i = 0:length-1
        if i == 0 || (A[i] != A[i-1] && symbols[A[i]] != BLANK)
            O.attach(symbols[A[i]])
        end
    end
    return O
end
```

Listing 5 Greedy decoding for a *single* test instance

However, this method is suboptimal – it chooses the most likely *time-synchronous* output, rather than the most likely *order-aligned* output as discussed in class.

2.3.2 Beam Search

As an alternative to Greedy Search, we may attempt to explicitly find the most likely *order-aligned* sequence. However, there are an exponential number of possible sequences, all of which must be evaluated in order to find the most likely one. This is clearly infeasible.

As a compromise, we can use the *beam-search* strategy that evaluates all possible hypotheseis (i.e. all potential symbol sequences) within a narrow beam of hypotheses.

The main idea behind Beam Search is to limit the scope of your exhaustive exponential search by an upper bound of k "beams" or sequences, by keeping a beam of top-k scored sub-sequences at each time step (`BeamWidth`). There is a real chance of missing the absolute best path using this strategy. However, the sub-optimality of Beam Search Decoding is much better than Greedy Decoding. For more details on beam search, please refer to the lecture slides.

If you have already implemented Beam Search in BeamSearch.py in your 3p1, you can use that implementation here. The Beam Search implementation of 3p1 outputs 2 arguments, one of which can be used to predict your sequence of phonemes. Otherwise, for 3p2, you can use the decoder you get from the `ctcdecode`library in python, which you will find in the starter notebook.

Note: You will likely be using beam search on your validation data, while you train. Increasing the beam width can exponentially increase the training time. Please stick to a smaller beam width (3) for your validation data, while training. You may increase the beam width while making predictions on test data.

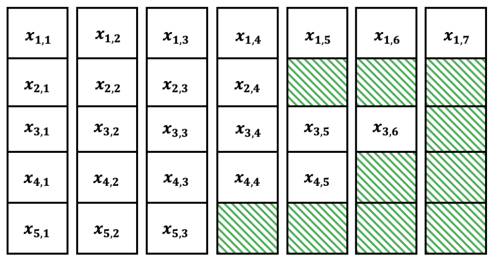
2.3.3 Caveats

We will usually work with *minibatches* of input, rather than individual inputs, for computational efficiency (and, in the case of training, better convergence). This gives rise to a problem in the sequence-to-sequence setting: the different input sequences in a mini-batch may all be of different lengths. However, for efficient processing we would like the data to be organized in a rectangular (or, more generally, *cuboidal*) structure. Vector processing units such as GPUs operate on the entire cuboid of data.

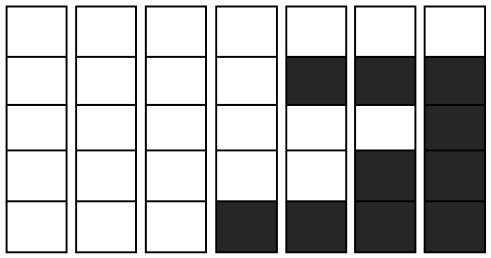
When the sequences in the minibatch are all of different length, we must *pad* the shorter sequences such that all of them are the same length (as illustrated in Figure 9a). The padded regions are usually 0-valued and carry no useful information. However, in the process of vector computation the processsor will nevertheless operate on these invalid regions of the input (the green regions of Figure 9a) and generate potentially meaningless outputs from them. This is a problem not merely during inference; it can also cause problems when we *train* the model.

To deal with this issue, we must *mask* the meaningless regions of the minibatch cuboid, using a binary mask that zeroes out the influence of the meaningless regions from the computation. Such a mask is illustrated in Figure 9b.

All of this leads to complications: (a) *how* do we select and pad inputs so that their lengths match up within a minibatch, (b) how do we construct the correspondingd masks, and, most importantly, (c) how do we use these masks to null out the contribution of the meaningless regions to training and inference. In our discussion of actual implementation details in Section **??**, we will also touch upon these issues.



(a) A minibatch of variable-length sequences. The green regions represent the padding.



(b) The masks associated with the inputs in the batch.

Figure 9: An example minibatch of size 5 consisting of variable length sequences of lengths 7, 4, 6, 5, 3 respectively. (b) An example binary mask for a minibatch is a one-hot matrix that has 0s in positions of padding and 1s elsewhere.

3 Dataset

In this homework we aim to train a sequence to sequence model that takes Mel-Frequency Cepstral Coefficients MFCCs. The data for this homework is similar to that of HW1P2, and you will be provided with mel-spectrograms that have 27 band frequencies for each time step of the speech data. You can refer to Recitation 0J to revisit MFCCs, and HW1P2 to revisit a detailed description of the data.

The feature data is an array of utterances, and each utterance is of dimension $(T_{in}, 27)$, where T_{in} refers to the number of frames in the utterance. Each utterance has a corresponding transcript (label) of length T_{out}, which refers to the length of the output sequence. This naturally keeps order synchrony but breaks time synchrony. For the purpose of this competition, we provide a look-up that maps each phoneme to a single character. (Refer to `phonetics.py`, downloadable from Kaggle.) **Note:** The mapping provided in a cell in the starter notebook is slightly modified and overloaded.

3.1 File Structure

- train-clean-360 : Full training set
- train-clean-100 : Small training set
- dev-clean : Dev (Validation) set
- test-clean : Testing set
- */mfcc/* : Same as HW1P2 for train, dev and test datasets
- */transcripts/* : 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.

The dataset class in this homework would be similar to homework except for 2 small changes:

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

4 Padding and Packing Sequences

As mentioned in Section 2.3.3, we outline a few PyTorch methods that help us work with sequences of variable lengths for more efficient computation.

PyTorch gives us a few functions to accomplish the same:

- `pad_sequence()`
- `pad_packed_sequence()`
- `pack_padded_sequence()`

To construct minibatches of sequences, we have to **pad** all sequences in the mini-batch to the length of the longest sequence in the given mini-batch with a constant padding value (typically 0). Lets use an example to understand this: say, given an input mini-batch `x`, you have six sequences of variable lengths, ranging from 2 to 9 as shown in Figure 10. For us to work with mini-batches, we would have to pad all sequences to have the same length as the longest sequence, which is of length 9. `pad_sequence(x)` returns a tensor of padded sequences with a constant padding_value, resulting in a mini-batch of shape (6, 9).

Packing is not as immediately intuitive. Consider the same example of a batch with variable length sequences: After padding, we have a mini-batch of shape (6,9). In order to perform a matrix multiplication with a weight matrix W, say of shape (9,3) - we will need $6 \times 9 \times 3 = 162$ multiplications. However, as mentioned in Section 2.3.3, we would be generating meaningless outputs on padded regions of the input. In reality, you would only need $(9 \times 3) + (8 \times 3) + (6 \times 3) + (4 \times 3) + (2 \times 3) = 96$ multiplications. Packing allows PyTorch to optimize computation by flattening the sequences by timestep (sorted in order of sequence lengths) while keeping track of the *effective* batch-size at each time-step. If you pass your padded sequence through the `pack_padded_sequence()` function, you would get back a tuple of outputs, where one would contain the packed sequences of tensors and the other would contain the actual batch size at each time step (ignoring the padding). (Figure 11).

4.1 Forward Pass - RNNs

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. 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

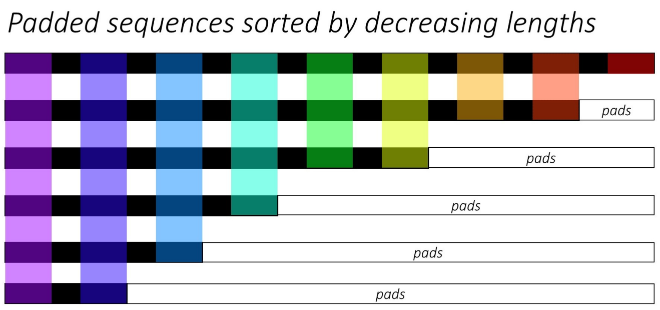


Figure 10: Batch with variable length sequences

In order to perform operations in batch, we must and pack and unpack the data every time when passing data through an RNN. It is important to *pack a padded sequence* using `pack_padded_sequence()` before passing it through an RNN, and to *pad a packed sequence* using `pad_packed_sequence()` after computation from the RNN.

The below two functions show that RNNs use a Packed Sequence object as input; they performs all computations required for each input at a given time-step before moving onto the next, enabling faster processing. Padding all inputs would consolidate all input MFCCs 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.

pack_padded_sequence:

- Input Parameters:
  1. data sequence
  2. lengths of each data sequence (as a list)
- Output:
  PackedSequence(data=tensor([]), batch_sizes=tensor([]), sorted_indices=tensor([]), unsorted_indices=tensor([]))
- Example:
  `packed = pack_padded_sequence(recordings, length_of_each_recording, batch_first=True, enforce_sorted=False)`

pad_packed_sequence:

- Input Parameters:
  1. takes in a packed data sequence as input
- Output Parameters:
  1. original data sequence

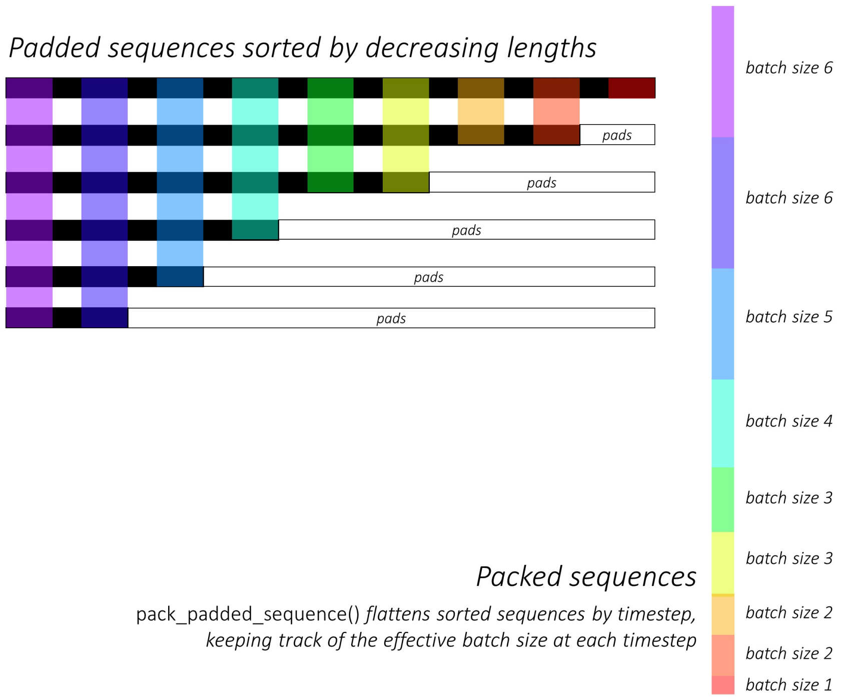

Padded sequences sorted by decreasing lengths
pads
pads
pads
pads
pads
batch size 6
batch size 6
batch size 5
batch size 4
batch size 3
batch size 3
batch size 2
batch size 2
batch size 1
Packed sequences
pack_padded_sequence() flattens sorted sequences by timestep, keeping track of the effective batch size at each timestep

Figure 11: Packed padded sequence

2. actual (unpadded) length of each sequence

- Example:
`seq_unpacked, lens_unpacked = pad_packed_sequence(packed, batch_first=True)`

5 Evaluation

You will be evaluated using Kaggle's character-level string edit distance; since we map each phoneme to a single character, you will be evaluated on phoneme edit-distance. Specifically, we will be using the Levenshtein Distance, which counts how many additions, deletions, and modifications are required to make one sequence into another (library).

Levenshtein Distance takes as input two strings, and the computed distance is the number of character differences between the strings. It is calculated in an order synchronous fashion as such: `Levenshtein.distance(target, prediction)`

In this case, the Levenshtein Distance = 2.

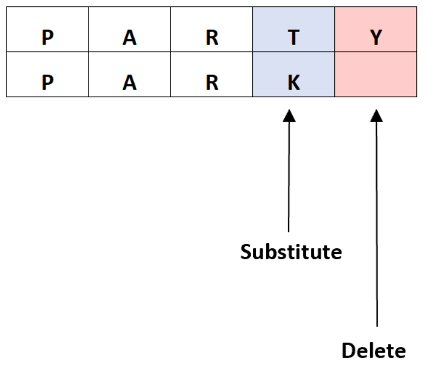


Figure 12: Calculation of Levenshtein Distance

6 Submission

6.1 Preliminary Submission

There is a mandatory preliminary submission and an associated MCQ that, together, are worth 10% of the points for the homework. This submission is intended to get you started quickly on the homework. The deadline for this preliminary submission is **March 25, 2023, 11:59PM**. We have provided a HW3P2 starter to help you with the preliminary submission, which you can download from piazza.

****Disclaimer**: The starter notebook is not as elaborate as it has been for previous homeworks. After 2 homeworks, we are expecting you to be in a position to write your own notebook from scratch. Please start early, completing the starter notebook will take time. You can reuse the code from previous starter notebooks if you want, but you will have to code most of the implementations yourself.

6.2 Final Submission

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. (See sample submission for details.) Note: the headers are case sensitive.

7 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, zoomer.

Appendices

A pBLSTM

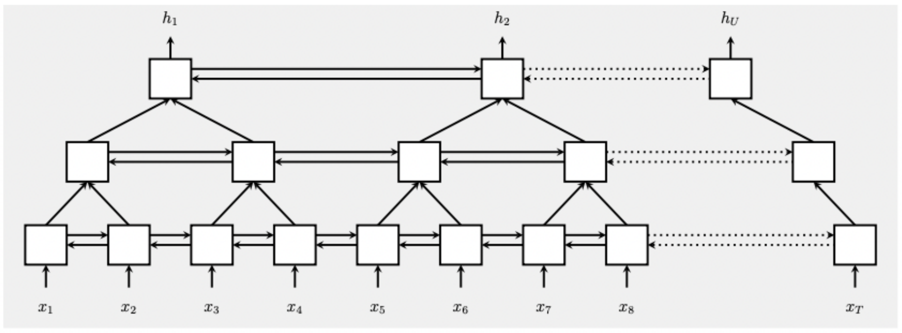


Figure 13: Taken from 'LAS', an illustration of the pBLSTM layers in the encoder with only 3 LSTM layers (and 2 time reductions).

The pBLSTM is a variant of Bi-LSTMs that downsamples sequences by a factor of 2 by *concatenating* adjacent pairs of inputs before running a conventional Bi-LSTM on the reduced-length sequence. So, given an input vector sequence $X_0, X_1, X_2, X_3, \ldots X_{N-1}$, the pBLSTM first concatenates adjacent pairs of vectors as $[X_0, X_1], [X_2, X_3], \ldots [X_{N-2}, X_{N-1}]$, and then computes a regular BiLSTM on the reshaped input.

If the length of the original input sequence is odd, then it is either padded out by appending an extra vector of zeros at the end, before reshaping, or trimmed to an even length by deleting the final vector. Note that we are *concatenating* adjacent vectors instead of averaging them to retain all the information in the sequence while also reducing the rate. You can also try pooling operations instead.

The following pseudocode shows how you could implement a simple pBLSTM. The *Params* in the code refer to the full set of BiLSTM parameters. Obviously, this is just pseudocode intended to describe the logic, and is not at all python-like; the actual code you write would look very different, and you would actually implement a pBLSTM class.

```
# X = (batchsize, length, dim) is a minibatch of sequences, possibly from a
    previous layer
# Assumes dataloader ensures all input sequences in the batch are the same length
function O = pBLSTM(X, LSTMwidth, Params)
    # Reshape inputs to have half the length, but twice the dimensionality
    X_downsampled = reshape(X,B,L/2,2*D)
    output = BiLSTM(X_downsampled, LSTMwidth, Params)
    return output
end
```

Listing 6 pBLSTM

Here BiLSTM() is a Bi-directional LSTM layer that takes in a minibatch of input sequences and generates a minibatch of output sequences of the same length. You could also replace the Bi-directional LSTM by a Bi-directional GRU.

If you are not enamoured of bidirectionality, the LSTM (or GRU) layer could be uni-directional, while maintaining the pyramidal structure. The key here is the combination of downsampling with an RNN.

B Viterbi Alignment

The Viterbi alignment algorithm finds the most probable alignment of a symbol sequence to an input. In doing so, it must also consider the invisible "blank" symbol, which can optionally occur between symbols. So, for example, given a symbol sequence /A/ /B/, it also considers that there might be invisible blanks between the symbols, i.e. that the sequence might actually be any of the following:

A/ /B/
/A/ /B/ –
– /A/ /B/
– /A/ /B/ –
/A/ – /B/
– /A/ – /B/
/A/ – /B/ –
– /A/ – /B/ –

The actual alignment might be an expansion of *any* of these sequences. The algorithm must determine *which* of these sequences best matches the input, and further what *expansion* of it best matches the input.

The algorithm proceeds as follows: Given a symbol sequence, say /A/ /B/ /C/, it first converts it to a directed graph that also represents optional blanks. Every variant of the sequence is a path from source to sink of this graph. This is done by introducing optional (skippable) blanks at the beginning and end of the sequence, as well as blanks between symbols, which may only be skipped if the symbols are non-identical (i.e. a blank is mandatory between repeating symbols). Figure 14 illustrates this with an example.

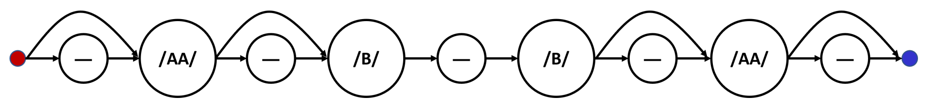


Figure 14: Graph representing all possible variants of the sequence /AA/ /B/ /B/ /AA/ with the optional blanks. Any path from the source (red) to the sink (blue) node represents a valid variant. Note that blanks are optional at the beginning and end of the sequence. They are also optional between two symbols, unless the two symbols are identical.

Subsequently the algorithm composes a *trellis* – a graph that "unrolls" the graph over time, for as many time steps as the length of the input. This results in a rectangular graph, with as many rows as the number of nodes in the graph. There in an edge from a node in the i^{th} row at time t to a node in the j^{th} row at the next time $t+1$ if there is an edge from the i^{th} node to the j^{th} node in the original graph itself. Figure 15 shows an example of such a graph.

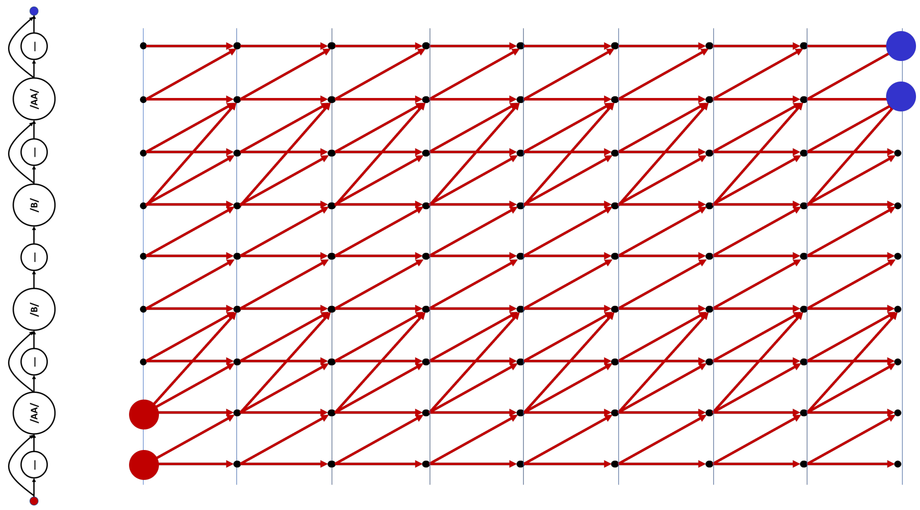


Figure 15: The trellis aligning the graph of Figure 14 against an sequence of 9 inputs. Any path from one of the red nodes (sources) to one of the blue nodes (sinks) is a valid alignment of /AA/ /B/ /B/ /AA/ to the input.

Each node in the graph is assigned a score equal to the log probability of the symbol it corresponds to, computed at the time it corresponds to. For instance, the node in the fourth row (corresponding to /B/ in our example) at time $t = 3$ would contain the log probability $log(P(/B/, t = 3))$, i.e. the log probability assigned to /L/ by the network at $t = 3$. Similarly, the node in the third row (corresponding to $-$) at $t = 4$ carries the log probability $log(P(-, t = 4))$. All edges have a score of 0.

The pseudocode below, lifted from the slides, explains how to compose the graph and find the best path through it. The code may be buggy, so please verify the logic first. Also, it is not very python, so you will have to pythonize it.

```
# S is a sequence of symbols in the target output
function Yext = ExtendedSequenceWithBlanks(Y)
    j = 1
    # loop through target symbol output
    for i = 1:N
        # add BLANK symbol before each symbol in target ouput
        Yext(j) = BLANK
        j = j + 1
        Yext(j) = Y(i)
        j = j + 1
    end
    # append BLANK symbol in the end
    Yext(j) = BLANK
    return Yext
end
```

Listing 7 Viberti Algorithm: Compose Graph

```
function AlignedSymbol = ViterbiAlign(LogP, Y)
    # extend the sequence with BLANK symbol
    Yext = ExtendedSequenceWithBlanks(Y)
    N = length(Yext) # length of extended sequence
    # BP(i, j) represents the best parent symbol when t=i, symbol=j
    # Bscr(i, j) represents the best probability at t=i and symbol=j
    # initialize best parents and scores at t=1
    BP(1,1) = BP(1,2) = -1
    Bscr(1,1) = LogP(1,Yext(1))
    Bscr(1,2) = LogP(1,Yext(2))
    Bscr(1,3:N) = -inf
    # loop through each timestamp
    for t = 2:T
        # assign the best parent and score for the first symbol in the target
           output
        BP(t,1) = BP(t-1,1)
        Bscr(t,1) = Bscr(t-1,1) + LogP(t,Yext(1))
        # loop through the symbols in the extended target output
        for i = 2:N
            if (i > 2 && Yext(i) != Yext(i-2))
                # check paths extended both previous symbol (if not duplicate) and
                   BLANK symbol
                BP(t,i) = argmax_i(Bscr(t-1,i), Bscr(t-1,i-1), Bscr(t-1,i-2))
            else
                # check only paths extended from BLANK symbol
                BP(t,i) = argmax_i(Bscr(t-1,i), Bscr(t-1,i-1))
            end
            # update the score by adding the probability of current symbol
            Bscr(t,i) = Bscr(t-1,BP(t,i)) + LogP(t,Yext(i))
        end
    end
    # Get the best aligned symbol sequence by backtracing best parents
    AlignedSymbol(T) = Bscr(T,N) > Bscr(T,N-1) ? N, N-1;
    for t = T downto 1
        AlignedSymbol(t-1) = BP(t,AlignedSymbol(t))
    return AlignedSymbol
end
```

Listing 8 Viterbi Algorithm