Homework 4 Part 2
Attention-based End-to-End Speech-to-Text Deep Neural Network

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

OUT: **Nov 18, 2022**
EARLY SUBMISSION BONUS: **Nov 26, 2022, at 11:59 PM ET**
DUE: **Dec 9, 2022, at 11:59 PM ET**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / 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.

- **Submission:**
  - You only need to implement what is mentioned in the write-up (although you are free and encouraged to explore further) and submit your results to Kaggle . We will share a Google form or an Autolab link after the Kaggle competition ends for you to submit your code.
- **TL; DR:**
  - In this homework you will work on a sequence-to-sequence conversion problem, in which you must train models to transcribe speech recordings into word sequences, spelled out alphabetically.
  - As in previous HWP2s, this homework too is conducted as a Kaggle competition. You can access the kaggle for the homework at – https://www.kaggle.com/competitions/11-785-f22-hw4p2/
  - You can download the training and test data directly from Kaggle using their API or the data tab in the competition. Your download will include the following:
    - ∗ The training data will be found in the train folder. It will include data pairs comprising sequences of audio feature vectors and their transcriptions, in mfcc and transcripts folders respectively.
    - ∗ The validation data will be similarly found in the dev folder.
    - ∗ The test data will be found in the test folder. It will only contain an mfcc folder.
    - ∗ We also have a toy dataset folder that is to be used for debugging only.
    - ∗ There is also a file called “submission.csv” to give you an idea about a sample submission.
  - You must use the train data to train the model. Data in the dev folder are to be used only for validation and not for training.
  - You must transcribe the utterances in the test folder and write them out into submission.csv following the format already specified in the file, and upload your results to Kaggle.
  - You will be graded by your performance. Additional details will be posted on piazza.

Homework objectives

If you complete this homework successfully, you would ideally have learned:

- To implement an attention-based system to solve a sequence-to-sequence problem.
  - How to setup an encoder
    - You will have an idea of synchrony/rate principles to consider in setting up the encoder
    - How to setup the (Bidirectional) Pyramidal LSTMs/LSTMCell
  - How to setup a decoder
  - How to setup attention
- You would have learned how to address some of the implementation details
  - How to setup a teacher forcing schedule
  - How to pad-pack the variable length data
- 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 effectively explore the search space and strategically finding the best solution
- As a side benefit you would also have learned how to construct a speech recognition system using the attention-based encoder-decoder architecture

Important: A note on presentation style and pseudocode

The following writeup is intended to be reasonably detailed and explains several concepts through pseudocode. However there is a caveat; the pseudocode is intended to be *illustrative*, not *exemplary*. It conveys the concepts, but you cannot simply convert it directly to python code. For instance, most of the provided pseudocode is in the form of functions, whereas your python code would use classes (in fact, we've written the pseudocode as functions with the express reason that you *cannot* simply translate it to your code. Function-based DL code will be terribly inefficient. You *must* use classes).

Nonetheless, we hope that when you read the entire writeup, you will get a reasonable understanding of how to implement (at least the baseline architecture of) the homework.

Table of Contents

Checklist

Here is a checklist page that you can use to keep track of your progress as you go through the writeup and implement the corresponding sections in your starter notebook. As you complete your unit tests for each component to verify your progress, you can check the corresponding boxes aligned with each section and go through this writeup and starter notebook simultaneously step by step.

1. Read Introduction
2. Dataset and Dataloader
   - ☐ Dataset description
   - ☐ Refer to HW0 and implement Dataset class
3. Encoder
   - ☐ Listener description
   - ☐ Implement pBLSTM
   - ☐ Implement Encoder
4. Attention
   - ☐ Attention description
   - ☐ Implement Attention computation
   - ☐ Mask Attention
5. Decoder
   - ☐ Speller description
   - ☐ Implement Decoder
6. Training
   - ☐ Training description
   - ☐ Implement Teacher Forcing
   - ☐ Cross-entropy Loss with padding
7. Inference
   - ☐ Implement Greedy Decoding
   - ☐ Implement Random Decoding
   - ☐ Implement Beam Decoding
   - ☐ Implement Levenshtein Distance

1 Introduction

Key new concepts: Sequence-to-sequence conversion, Encoder-decoder architectures, Attention.

Mandatory implementation: Encoder-decoder architecture, Attention. Your solution *must* implement an encoder-decoder architecture *with* attention. While it may be possible to solve the homework using other architectures, you will not get marks for implementing those.

Restrictions: You may not use any data besides that provided as part of this homework. You may not use the validation data for training. You cannot use an implementation from Pytorch or any other third party package for implementing any type of Attention (and optionally beam search).

1.1 Overview

In this homework you will learn to build neural networks for sequence-to-sequence conversion or transduction.

"Sequence-to-sequence" conversion refers to problems where a sequence of inputs goes into the system, which emits a sequence of outputs in response. The need for such conversion arises in many problems, e.g.

- Speech recognition: A sequence of speech feature vectors is input. The output is a sequence of characters writing out what was spoken.
- Machine translation: A sequence of words in one language is input. The output is a sequence of words in a different language.
- Dialog systems: A user's input goes in. The output is the system's response.

For the homework, we will work on the speech recognition problem, since we're already familiar with the data from HW1P2 and HW3P2. There is one difference though – in HW1P2 and HW3P2 we tried to predict the *phonemes* in the speech. Here, we learn to directly spell the spoken sentence out in the English alphabet.

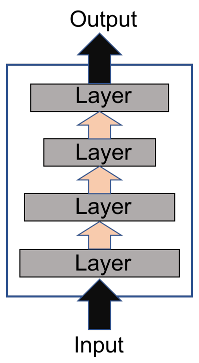
A key characteristic of such problems is that there may be no obvious correspondence between the input and the output sequences. For instance, in a dialog system, a user input "my screen is blank" may elicit the output "check the power switch". Or, more related to the problem of this homework, in a speech recognition system the input may be the (feature vector sequence for the) spoken recording for the phrase "know how". The system output must be the character sequence "k", "n", "o", "w", " ", "h", "o", "w" (note the blank space " " between know and how – the output is supposed to be readable, and must include blank spaces as characters). There is no obvious audio corresponding to the silent characters "k" and "w" in *know*, or the blank space character between *know* and *how*, yet these characters must nevertheless be output.

As a consequence of the absence of correspondence between the input and output sequences, the usual "vertical" architectures that we have used so far in our prior homeworks (Figure 1a), where the input is sequentially processed by layers of neurons until the output is finally computed, can no longer be used.

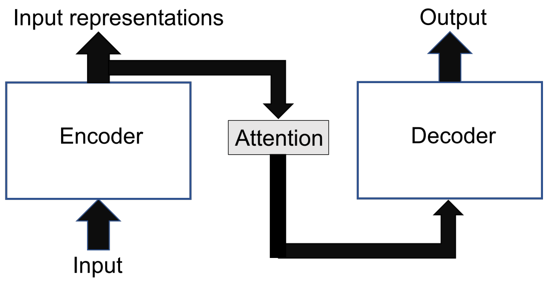
Instead, we must use a two-component model, comprising two *separate* network blocks, one to process the input, and another to compute the output. Such architectures are called *encoder-decoder* architectures. The encoder processes the input sequence to compute feature representations (embeddings) of the input sequence. The decoder subsequently uses the input representations computed by the encoder to *sequentially* compute the output *de novo*, *i.e.* from scratch (Figure 1b).

The most important part of the model is the *attention mechanism.* Although the decoder's output may not have a direct correspondence with the input, each individual output symbol (character) nonetheless relates more to some parts of the input than others, *e.g.* the blank character " " in our above speech recognition example corresponds primarily to the audio at the boundary between the words "know" and "how" in the recording. When outputting the " ", the decoder must somehow know to "focus" on, or "pay attention" to this region of the input.

In this homework you will learn how to build an encoder to effectively extract features from a speech signal, how to construct a decoder that can sequentially spell out the transcription of the audio, and how to implement an attention mechanism between the decoder and the encoder.



(a) Vertical architecture



(b) Encoder-decoder architecture

Figure 1: (a) Standard "vertical" architectures we have seen in previous homeworks. The input is sequentially passed through several layers until the output is computed from the final layer. *This kind of architecture is not useful for sequence-to-sequence problems with no input-output correspondence, such as the problem we deal with in this homework.* (b) Encoder-decoder architecture, comprising two separate modules, an encoder to process the input and a decoder to generate the output. The two are linked through an attention module.

Although the specifics of the homework will be targeted towards speech recognition, please note that the general framework (with appropriate modification of the encoder and decoder architectures) should also apply to other types of sequence-to-sequence problems, or even to image or video captioning (you would only need to modify the encoder to derive features from images or video respectively).

1.2 Problem specifics

In this problem of speech recognition you will learn how to convert a sequence of feature vectors of speech to a sequence of characters that spells out its transcription. For this, you will have to implement the following:

- **An encoder that can derive feature vectors from the input sequence of speech vectors**. The encoder must consider both, the specific nature of the input and its temporal relationship to the output.

  The input speech feature vectors have strong *structural* continuity with adjacent vectors (which can be exploited, as we saw in HW1P2 and HW3P2), as well as longer-term *contextual* dependencies to more distant vectors (as a result of long-distance context dependencies in speech production). Also the rates and regularity of input and output sequences are very mismatched – speech vectors are computed at the the uniform rate of 100 frames (vectors) per second, whereas their transcriptions average only about 15 characters per second of speech (including the blanks) and can vary widely from second to second.

  The encoder must ideally both model the context-dependencies in the speech vectors, and account for the rate differences of the input and outputs. Consequently, you will be exploring architectures that use a combination of components such as Convolutional Neural Networks (to capture correlations with immediate neighbors) and Recurrent Neural Networks (to capture longer-term dependencies), and strategies that attempt to modify the rate of input features to approximately match those of the output.

- **A decoder to produce the actual output sequence**. The output is a series of characters that is a function of the input. Unlike the input, we don't expect to see structural correlations between adjacent outputs; however we *will* expect long-term dependencies between them. Also, as mentioned earlier, the output is generated *de novo*, *i.e.* from scratch. This is done sequentially, from the first symbol to the last. When generating any output symbol, only the *previous* outputs can be expected to be available.

  The decoder must consider these characteristics and constraints. You will be exploring recurrent, *autoregressive* architectures (the actual output feeds back as inputs for subsequent timesteps) for the decoder.

  Decoding is generally probabilistic – output symbols are not deterministically generated, but are *drawn* from the probability distributions computed by the decoder. You will explore different mechanisms of performing inference to produce the most likely output for any input.

- **Attention**. Each output character (in the output sequence) relates to some portion of the original input audio. To generate each output, the decoder receives, as input, a "context", comprising a weighted sum of the sequence of representation vectors computed by the encoder. The weights with which the vectors are combined must be such that the context pays most "attention" to the most relevant part of the input.

  You will explore ways of computing these attention weights such that they automatically learn to pay attention to the most relevant parts of the input for each output.

- **Training**. And last, but not the least you will learn how to train a complicated model that has all of the above components to do your bidding. One of the consequences of the probabilistic nature of decoding is asynchrony between the actual and target outputs of the decoder. You will also explore ways of dealing with this through a mechanism called "teacher forcing", and the effect of various optimization strategies and schedules on your models.

This may seem like a lot, but believe it or not, you will manage to get all this done in the course of this homework. In the following sections we will outline how.

We provide you a baseline architecture based on *"Listen, Attend and Spell", William Chan, Navdeep Jaitly, Quoc V. Le, Oriol Vinyals* (which we recommend that you read), and describe the key components of it later in Section 3. But you are also welcome to attempt other variations (for any of the three components – encoder, decoder or attention) provided they fall within the *Encoder-Decoder+Attention* framework.

1.3 Getting Started Early

This homework includes several novel concepts not previously encountered in any homework. As a result, it can, and most likely will take a significant amount of your time compared to previous homeworks, so please start early. Not only is implementing this HW harder than previous ones (with more moving parts, you are more likely to encounter errors, which will take significant time to identify and debug), the models also typically take longer to converge and produce good results.

The rest of this writeup is structured as follows – Section 2 describes the dataset and dataloader. In Section 3 we outline the baseline model, including the individual components (encoder, decoder and attention), how to perform inference/a forward pass with it, and how to train it. Section A and the Appendices outline debugging and specific implementation details.

2 Dataset

You will be working on a similar dataset to HW1P2 and HW3P2. The training set will comprise input-output pairs, where each input consists of a sequence of 15-dimensional mel-spectral vectors derived from a spoken recording, and the corresponding output consists of the spelled out text transcription. The transcriptions are in terms of 30 characters (the 26 characters in the English alphabet, blank space, "," [comma] and two special characters: "<sos>" and "<eos>", representing the start and end of a sentence respectively).

The audio in the different training instances are of different lengths, and the output sequences too are (obviously) of different lengths. We also provide you a Python array `VOCABULARY` (refer to the starter notebook) which contains a mapping from the letters to numeric indices. *You must use this list to convert letters from transcriptions to their indices* for the purposes of the competition.

2.1 Dataloader

You will need to implement the Dataset class for this HW by your own. Please refer to the dataloader section from HW0 for the exercise on implementing it. We also recommend using cepstral normalization to normalize input cepstral vector sequences for this HW as provided in HW1P2 (although this is not mandatory). Although no output labels are provided for test data during inference, we still have to worry about length – when test data are batched, the outputs for the different inputs in a batch of test data will all also be of different lengths. We recommend using a maximum length of 600 for generating your outputs for test and validation data, although you are free to experiment with it. More details on how to handle variable data length are in Appendix D.

In addition, we provide a toy dataset for debugging purposes to ensure you implement your model correctly and obtain the correct outputs for the toy data. More information on this is provided in Appendix A.

3 Baseline Approach

The baseline model for this assignment is derived from the paper "Listen, Attend and Spell" (LAS). The paper describes an encoder-decoder approach. The encoder in this model is called a *Listener* since it "listens" to the audio, the decoder, which spells out the transcription, is called the *Speller*, and *Attend* of course refers to the attention module. The objective is to learn all components jointly.

The Listener consists of a *Pyramidal Bi-LSTM Network* structure that takes in the audio feature vector sequences of the given utterances and outputs sequence of high-level representation vectors that occur at approximately the same rate (vectors per second) as the expected (spelled out) output of the Speller network.

The Speller takes in the high-level feature output from the Listener network and uses it to compute a probability distribution over sequences of characters using the attention mechanism.

Attention can be intuitively understood as trying to learn a mapping from a word vector to some areas of the utterance map. The Listener produces a high-level representation of the given utterance and the Speller uses parts of that representation using Attention to predict the next word in the sequence.

For this HW, we elaborate on a variant of LAS you can implement as your model. We believe that the LAS paper has some unclear explanations, and hence we will explain and provide additional information on such areas in the writeup. We recommend following this approach for the HW, as it is in line with the starter code template provided. However, you are free to develop your own model, or write a notebook from scratch.

We outline the model components, inference and training procedures below.

3.1 Model

The model has three components – the Listener (encoder), the attention module, and the Speller (decoder).

3.1.1 Listener (Encoder)

The encoder takes in an input sequence of vectors $X_0, \cdots, X_{N-1}$ and produces a sequence of latent representations (or *embeddings*), $E_0, \cdots, E_{M-1}$. The length of the embedding sequence M need not be the same as that of the input sequence, N, but each embedding vector E_i effectively represents a section of the input.

The listener in the LAS model is an acoustic model encoder that takes into consideration the nature of the input feature vector sequence and rate difference between the input and output of the speech recognizer.

The input sequence of speech feature vectors exhibit both short-term structural relations, and long-term contextual dependence. The listener typically tries to capture these dependencies. There is also an approximate factor of 8 difference between the rate of the feature vector sequences in the audio (100 vectors per second) and their spelling (on average 12-15 characters per second). So the listener includes components that effectively lower the rate of the latent representations, typically by a factor of 8 to match the ouput.

A listener typically consists 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 (typically 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.

In the following subheadings we briefly outline the implementation of a pBLSTM, and a typical instantiation of a listener. It is up to you to experiment with other variations.

3.1.1.1 pBLSTM

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 as described in the paper.

Listing 1 pBLSTM

```
# X = (batchsize, length, dim) is a minibatch of input sequences, possibly from a previous layer
# Assuming dataloader ensures that 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
```

The *Params* in the code refer to the set of BiLSTM parameters, including all weights and initial state values. Obviously, this is just pseudocode intended to describe the logic, and is not at all python-like. The actual code you write will look very different, and you will actually implement a pBLSTM class.

Figure 2, taken from the paper, illustrates the working of the pBLSTM layers in the Listener.

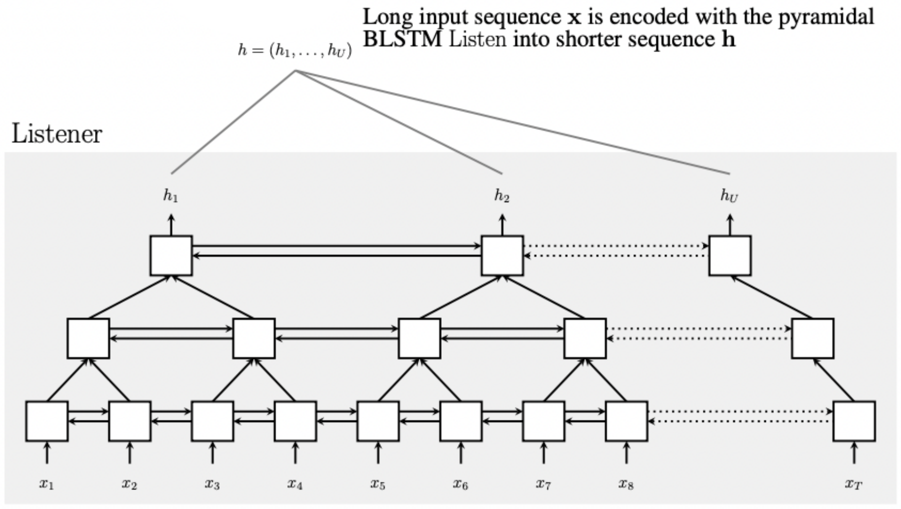

Long input sequence **x** is encoded with the pyramidal BLSTM Listen into shorter sequence **h**
$h = (h_1, \ldots, h_U)$
Listener
h_1
h_2
h_U
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_T

Figure 2: The listener in Figure 2 has only 3 LSTM layers (and 2 time reductions). The paper actually says to use 4 (specifically, one initial Bi-LSTM followed by 3 pBLSTM layers each with a time reduction)

3.1.1.2 A typical listener

Here is pseudocode for a typical complete Listener that reduces the rate by a factor of 2:

Listing 2 Listener

```
# 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 = Listener(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
```

This is only an example; you must devise your own architecture. There are many variants you could try. The 1DCNNs could be replaced by ResNets. Also, the above code only downsamples the input by 2. To downsample it further you could either include strides in CNN layers or have more pBLSTM layers.

For reference, in the original LAS paper, there is no 1DCNN layer, one BiLSTM layer (of width 256) and three pBLSTM layers. Please refer to the paper for more details.

3.1.2 Attention (Attend)

The *key* feature of this homework (and the LAS model in general) is the *attention* mechanism.

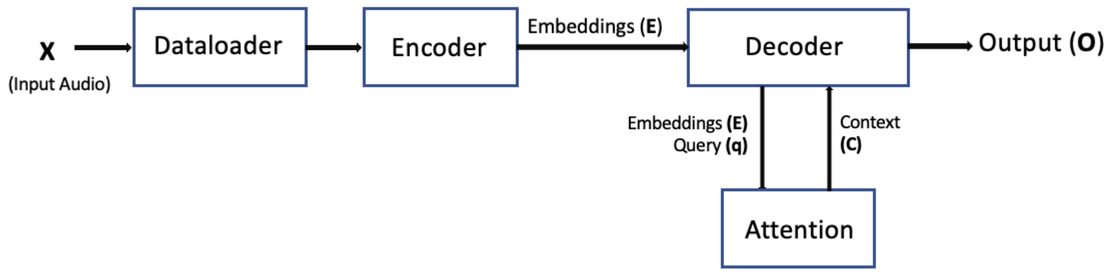


Figure 3

Conceptually, attention works as follows. Each of the encoder-derived embeddings $E_0, \cdots, E_{M-1}$ effectively represents a section of the input. As mentioned in Section 1, each symbol output by the *decoder* refers to a different portion of the input. Information on which portion of the input to refer (or pay *attention*) to, and what features to derive from it must both be decided from the encoder-derived embeddings. Both of these are typically derived through *linear* functions of the embeddings.

In a typical implementation of attention, from the embeddings E_i the attention module computes two terms, a *key* K_i and a *value* V_i, through linear transforms: $K_i = E_i\mathbf{W}_K$, and $V_i = E_i\mathbf{W}_V$ (assuming the E_is to be row vectors), where $\mathbf{W}_K$ and $\mathbf{W}_V$ are learnable parameters. Key K_i is used to compute the *relevance* of E_i to any output, and value V_i is the actual feature E_i contributes to it.

For each output O_k, the decoder computes a *query* $\mathbf{q}_k$ to query the attention module. The module uses the query $\mathbf{q}_k$ and and the keys K_j to compute a set of *attention weights* w_{kj}. Attention weights are strictly non-negative and must sum to 1.0. They are typically computed by a softmax applied to a normalized innerproduct of the queries and the keys. Ideally the w_{kj} values for embeddings E_j corresponding to inputs that are most relevant to O_k must be high, while others are low. This is expected if $\mathbf{W}_K$ and $\mathbf{W}_V$ are well learned. Finally a *context* vector C_k is computed as $C_k = \sum_j w_{kj}V_j$, which is used to compute O_k.

3.1.2.1 Computing attention for the LAS model

In the LAS model, the actual attention is computed by the *attend* module as follows. Assuming all the embeddings E_i to be *row* vectors (in the standard python format), the set of embedding vectors can be vertically stacked into a matrix $\mathbf{E}$.

The *attend* module accepts a query $\mathbf{q}$ from the decoder, and returns a context vector C computed from the embeddings $\mathbf{E}$ as follows. Here $\mathbf{K}$ and $\mathbf{V}$ represent matrices formed by a vertical stacking of the key and value vectors derived for all the embedding (the j^{th} row of $\mathbf{K}$ and $\mathbf{V}$ represent K_j and V_j, the key and value derived from the j^{th} embedding E_j). $\mathbf{w}$ is a row vectors whose j^{th} component is the attention weight for E_j.

$$\begin{aligned}
\mathbf{K} &= \mathbf{E}\mathbf{W}_K \\
\mathbf{V} &= \mathbf{E}\mathbf{W}_V \\
\mathbf{w} &= softmax\left(\frac{1}{\sqrt{length(\mathbf{q})}}\mathbf{q}\mathbf{K}^\top\right) \\
C &= \mathbf{w}\mathbf{V}
\end{aligned}$$

The following pseudocode explains how a context vector is computed in response to a query.

Listing 3 Attention (single sequence)

```
# E = sequence_length x dim array of embeddings for \textit{single} input
# query = query vector (assumed to be a (1,Dq) row vector)
# context = context vector computed (1,Dv) row vector)
function Attention = Attend(E, query)
    # WK and WV must also be specified in actual code
    key = E * WK        # WK is a (dim,Dq) matrix
    value = E * WV        # WV is a (dim,Dv) matrix
    query_length = length(q)
    raw_weights = (1/sqrt(query_length)) * query * transpose(key)
    attention_weights = softmax(raw_weights)
    context = attention_weights * value
    return context, attention_weights
end
```

The above pseudocode only shows how to compute *one* context vector for a *single* output symbol, computed from a *single* input sequence. In practice, you would be performing computations on entire minibatches of inputs, pseudocode for which can be found in Appendix F

The attention computed above, which derives a *single* context vector C_k to compute an output O_k is sometimes known as "single-headed" attention. A simple extension that derives multiple such context vectors is known as "multi-headed attention". Details of how to implement multi-headed attention can be found in Appendix H. While properly implemented multi-headed attention can provide performance benefits, for the baseline implementation of the LAS paper, however, single-headed attention should be sufficient.

3.1.3 Speller (Decoder)

The Speller is the decoder in the LAS model. It's task is to output the most likely output character sequence for the given audio, from the sequence of high-level representations derived for it by the Listener.

The Speller is a *probabilistic autoregressive* model that *sequentially* generates the output from start to end, using context information from the encoder. The term *autoregressive* here refers to the fact that in order to generate an output $O(t)$, the model explicitly uses previous outputs $O(t-1), O(t-2)$ etc. as inputs. This is to account for the fact that the probability distribution computed at any time depends explicitly on the *actual outputs* at previous times. For instance, if the decoder draws and outputs the sequence " an", at the next output the probabilities of vowels must be high (since "an" is usually followed by nouns starting with vowels), whereas if the output sequence drawn is " a", the probabilities of consonants should be high next.

The dependency on the past symbols (characters or words) in language is actually very deep – the next character in sentence depends not only on the *most recent* character, but actually on all the characters (or words) so far in the sentence. Explicitly modelling this into an autoregressive model will require a large number of parameters; also the order of the autoregression (how many past outputs are considered) limits its ability to model dependency on past outputs.

Instead, the decoder typically only uses a low-order autoregression – generally limited to *first-order* autoregression, *i.e.* it explicitly only considers $O(t-1)$ in predicting $O(t)$ (although we could certainly use higher-order autoregression) to capture dependence on the immediate past outputs, and uses *recurrence* to model dependence on the more distant past. Recurrence is generally modeled by an LSTM.

Also, since the objective is to output the *most likely* transcription for an input, the decoder is *probabilistic* – it generates *probability distributions* over the symbol vocabulary at each time. The actual output is produced by drawing an output from this distribution.

Moreover, each decoder step does not depend on the *entire* sequence of embedding vectors $\mathbf{E}$, but rather only on an *attended* context vector derived from it. In particular, in the listener described in the LAS paper, the context vector at any timestep is used *twice*, once to compute character probabilities at that timestep,and as input to the decoder at the *next* timestep. So the sequence of operations for any step of the decoder, as followed by (our variant of) the LAS paper, can be written as:

$$\begin{aligned}
S(t) &= LSTMstep(S(t-1), O(t-1), C(t-1)) \\
query &= S(t) \\
C(t) &= Attend(\mathbf{E}, query) \\
P_O(t) &= CharacterDistributionNet(C(t), S(t)) \\
O(t) &= DrawFrom(P_O(t))
\end{aligned}$$

where $LSTMStep()$ is one step of the decoding process, $\mathbf{E}$ is the set of high-level input representations (embeddings) obtained from the encoder, $S(t)$ is the recurrent hidden state of the decoder, $P_O(t)$ is the probability distribution over output symbols computed for the t^{th} output symbol, and $DrawFrom()$ is some process that *draws* an output symbol from the distribution $P_O(t)$ (more on this later). The above example assumes first-order autoregression; you may choose to use higher orders. Note the similarity of this process to the language modeling problem of HW4P1 – decoding is an extension of the language generation problem.

Figure 4 illustrates this process. $LSTMStep()$ is a single step of LSTM computation in the Decoder. Here $CharacterDistributionNet()$ is a module that computes the probability distribution $P_O(t)$ from the context $C(t)$ and state $S(t)$. In its simplest form it is just implemented by a linear layer followed by a softmax. In a more complex implementation, it could be an entire MLP with a final softmax output layer.

There are a few problems to resolve with the above autoregressive recurrence:

- What is the initial value $O(-1)$ required to generate the first output symbol? We use a special "start of sentence" `<sos>` symbol.
- What is the initial state $S(-1)$? We can initialize it as a vector of 0s, a vector of 1s, or make it a *learned* parameter. There are other options as well. In general a vector of zeros does the job.
- What is the initial context $C(-1)$? This is typically set to simply be either the initial value vector $V(0)$, or the mean of value vectors, which can be obtained by invoking the attention module by an all-zero query.
- When do we *stop* generating outputs? We include a special "end of sentence" `<eos>` symbol in our vocabulary. Output generation stops when $O(t) =$ `<eos>`.

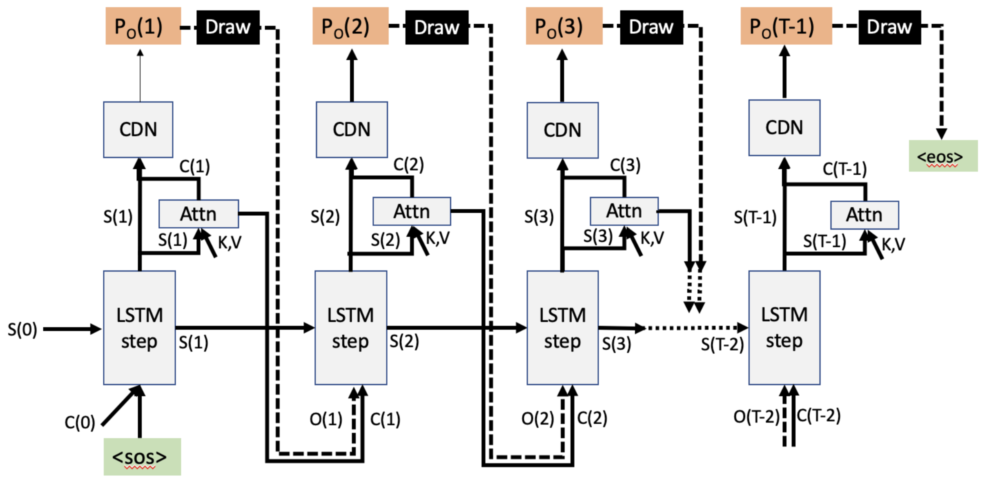


Figure 4: At timestep t, $S(t)$ represents the hidden representation obtained after LSTM step. Keys and values obtained from the encoder are represented by K,V. $C(t)$ represents the context vector obtained from the attend module (Attn) using K,V and S(t). CDN represents the Character Distribution Network. $P_O(t)$ represents the probability distribution over characters. $O(t)$ is the output symbol drawn from $P_O(t)$ obtained from CDN.

3.1.3.1 A typical speller

The pseudocode for the entire Speller (single instance) is thus as follows:

Listing 4 Speller

```
function O = Speller(E)
   t = 0, hidden_states(t) = 0, output_symbols(t) = <sos>
   query = [0,0, ...] # A vector of zeros of the same length as S(t)
   context(t) = attention(E, query) #or, alternately, as V[0]
   do
      t = t+1
      hidden_states(t) = LSTMstep(output_symbols(t-1), hidden_states(t-1), context(t-1)) # S(t-1)
          is the recurrent LSTM state(s)
      query = hidden_states(t)
      context(t) = Attend(E,query)
      prob_dist(t) = Character_Distribution_Net([context(t), hidden_states(t)])
      output_symbols(t) = DrawFromDistribution(prob_dist(t))
   until (output_symbols(t) == <eos>)
end
```

Listing 5 LSTMStep

```
functionS = LSTMStep(output_symbols, hidden_states, context)
    word_embedding = embedding(output_symbols) # We actually work with embeddings of characters
    lstm_input = [word_embedding, context]
    lstm1_hidden_state = lstm1(lstm_input)
    lstm2_hidden_state = lstm2(lstm1_hidden_state)
    hidden_states(t) = lstm2_hidden_state
   return hidden_states(t)
end
```

Equivalent pseudocode for minibatches can be found in Appenidx I.

In order to implement the Speller using Pytorch, you cannot use the standard LSTM layer implementation directly as a consequence of invoking the attention module at each timestep for generating output symbols. You will have to explicitly use LSTM cells (instead of LSTM layers), at each time step. In the pseudocode we have used two LSTM cells, but you could use more or less, depending on your setup.

We have now described nearly all the components of the LAS model (except for `DrawFromDistribution()`), and are now ready to explain the two key procedures: inference with the model, and training the model. The one unexplained component, `DrawFromDistribution()` will be explained as part of inference.

3.2 Inference

Inference in a sequence-to-sequence model attempts to find the most likely transcription for the given audio. Given the model and an input (audio), the process for obtaining the output (transcription) is very straightforward: first encode the audio,and then run the decoder on the derived hidden representations. So the entire process can be encapsulated by the following pseudocode:

Listing 6 Inference

```
# X is the input audio feature vector sequence
# O is the output transcription
function O = Inference(X)
    E = Listener(X)
    O = Speller(E)
end
```

(The code above is for a single instance; we leave it to you to work out the generalization to a minibatch).

However, the above code still requires specification of a key component of the Speller – the operation `DrawFromDistribution(P)`. Recall that the objective of the decoder is to find *the most likely* output sequence for any given input. This will depend on the specific implementation of `DrawFromDistribution(P)`. There are multiple possible ways of implementing this.

3.2.1 Greedy Search

The naive approach is greedy search, also the easiest decoding method for inference. All you would have to do is to draw the character with the highest probability at each time step from the output probability distribution for the entire batch of predictions you get from your model. Pseudocode would look like this -

Listing 7 Greedy Decoding

```
function O = DrawFromDistribution(P)
        return argmax(P)
```

While this usually generates reasonable results, due to the autoregressive nature of the model, the greedy method does not however guarantee that the final output sequence is the most likely one. Hence, in order to get the most likely final output, exploring other techniques like random search and beam search, are elaborated in Appendix J and Appendix K respectively.

3.2.2 Random Search

An alternate strategy is to simply draw outputs randomly according to the distribution P :

Listing 8 Random Decoding

```
function O = DrawFromDistribution(P)
        return Sample(P)
```

Here `Sample()` is a function that randomly draws an output according to a distribution P. Once again, random sampling this does not guarantee that the final output sequence is the most likely one, although, strangely enough, random sampling can sometimes result in sequences that are more probable overall than greedy outputs.

In practice, when using the random sampling procedure, one would generate *several* decodes (100, or even 1000), by rerunning the decoder multiple times. Since only the decoder is being run (the encoder only needs to be run once), this is relatively inexpensive. We would then choose the decode with the highest probability.

3.2.3 Beam Search

To actually obtain the most likely output sequence, we need a rather more complex decoder than the one described above in the pseudocode – we need beam search. This is the method chosen in the LAS paper.

However, implementing this will involve not using the basic decoder and have a completely different approach to the decoding strategy given above. To select the most probable output sequence we must, ideally, now explicitly evaluate all possible sequences and select the most likely. This is infeasible. Instead, we use a beam search, which iteratively expands out the K most probable paths, until it finds a most likely path that ends in a `<eos>`. Pseudocode for beam search can be found in Appendix K.

3.3 Training

Finally, we come to training. In the type of neural networks we dealt with previously, the process of training was simple. To compute parameter gradients from an input, we ran an initial inference – a "forward pass" – over the input to get an output, and computed a loss that compared the network output to the target output. The derivatives of the loss were propagated backward through the net to compute parameter gradients and update the model.

In the sequence-to-sequence setting, however, this is challenging or even infeasible for a simple reason – there may be no correspondence at all between the output of the inference and the target output. For instance, a training instance may have the ground-truth transcription of "know how", whereas the decoder output may be "it it it it it it it it it it". This sort of absurd output is not at all unlikely, particularly in the initial stages of training when the model is poor. The two sentences "know how" and "it it it it it it it it it" are not comparable in length, and have no obvious alignment that could be used to compute a loss that can be differentiated.

In order to be able to compute a differentiable loss, we would ideally require the forward pass to generate an output that is not only comparable in length to the ground truth, must also have *character-wise correspondence* with the ground truth character sequence. In order to achieve this, the forward pass itself must be modified for training. Instead of making it autoregressive, we will explicitly use the ground truth as input in the decoder. This is what is referred to as *teacher forcing.*

3.3.1 Teacher Forcing

In the modified form of the decoder that we use for training, the pseudocode will change to the following.

```
# Y(1:Tlen) are the sequence of characters in the ground-truth transcriptions
# We are assuming that Y(0) has been set to <sos> and Y(Tlen+1) has been set to <eos>
function prob_dist = TeacherForcedSpeller(E, Y)
   t = 0
   hidden_states(t) = 0
   query = [0,0, ...] # A vector of zeros of the same length as hidden_states(t)
   context(t) = attention(E, query) #or, alternately, as V[0]
   for t = 1:Tlen
      hidden_states(t) = DecoderLSTMstep(Y(t-1), hidden_states(t-1), context(t-1)) #
          hidden_states(t-1) is the recurrent LSTM state(s)
      query = hidden_states(t)
      context(t) = Attend(E,query)
```

```
        prob_dist(t) = Character_Distribution_Net([context(t), hidden_states(t)])
    end
end
```

Note that this has actually changed only marginally from the decoder in Section 3.1.3.

- The Speller now also takes in the ground truth `Y`.
- We no longer draw `output_symbol(t)` from `prob_dist(t)`. Instead, `DecoderLSTMstep()` is directly fed the ground truth `Y(t)`. This is the "teacher forcing" – the "teacher", i.e. the ground truth, is being forced as input to the decoder, instead of an output that would usually be drawn from `prob_dist(t)`.
- Instead of looping until the drawn sample is `<eos>`, we now simply loop until we run out of characters in the ground truth.

Figure 5a illustrates this process.

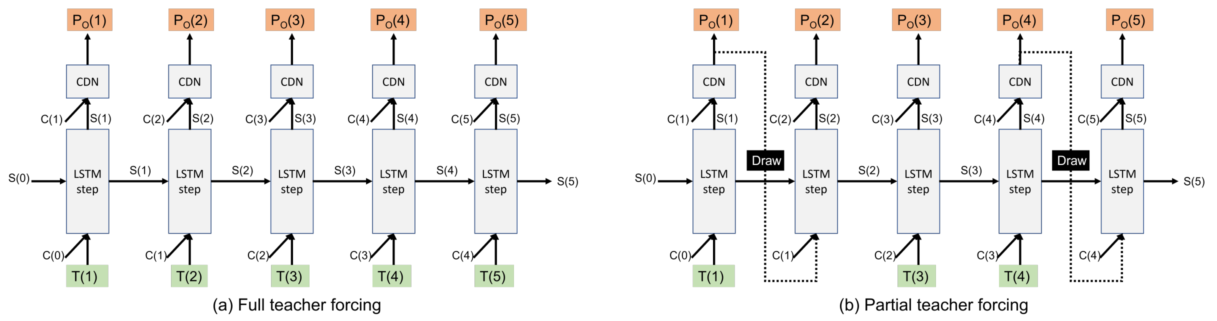


Figure 5: (a) Full teacher forcing. "CDN" is the Character_Distribution_Net. The `T(i)`s are teacher-forced inputs. (b) Partial teacher forcing. "Draw" is DrawFromDistribution. Dotted lines represent drawn inputs.

Note how, because the decoder is now only actually predicting the *next* symbol at any time (since the inputs are deterministically the ground truth), at any time, there is strict correspondence between the predicted output and the target output at that time. The loss is now computed between the sequence outout probability distributions `PO(t)` and the ground truth sequence. Loss computation is described in Section 3.3.2.

The approach in the above pseudocode *inputs* the ground truth to the decoder at every time step (as opposed to autoregressively feeding in the drawn output as mentioned earlier). This process is very different from actual inference on test data, where such ground truth will obviously not be available to provide as input. As a result, the process can result in suboptimal models.

An intermediate option between full inference for the forward pass (which cannot be used for training as we saw) and a fully teacher-forced decode in the forward pass is *partial* teacher forcing, where, for *some* of the time we actually provide a drawn sample instead of the ground truth as input to the decoder. This still maintains one-to-one correspondence between the decoder output and the ground truth, and permits loss computation. The modified pseudocode for this operation is given below. Here the variable `tf` in the code is the *teacher-forcing ratio*, which is the fraction of time for which the input is teacher forced. When `tf = 1`, every input is teacher-forced and we get back the situation in the pseudocode above (Figure 5a).

```
# Y(1:Tlen) are the sequence of characters in the ground-truth transcriptions
# We are assuming that Y(0) has been set to <sos> and Y(Tlen+1) has been set to <eos>
function prob_dist = TeacherForcedSpeller(E, Y, tf)
    t = 0
    hidden_states(t) = 0
    query = [0,0, ...] # A vector of zeros of the same length as hidden_states(t)
    context(t) = attention(E, query) #or, alternately, as V[0]
    for t = 1:Tlen
        if (t > 1) # At the first time step the input symbol is always <sos>
```

```
            p = generateuniformrandomnumber()
            if (p <= tf)
                inputsymbol = Y(t-1)
            else
                inputsymbol = DrawFromDistribution(prob_dist(t-1))
            end
        hidden_states(t) = DecoderLSTMstep(inputsymbol, hidden_states(t-1), context(t-1)) #
             hidden_states(t-1) is the recurrent LSTM state(s)
        query = hidden_states(t)
        context(t) = attention(E,query)
        prob_dist(t) = Character_Distribution_Net([context(t), hidden_states(t)])
    end
    return prob_dist
end
```

(The above code is written for a single instance. You must adapt this to work over minibatches.)

Figure 5b illustrates this process. The `DrawFromDistribution()` may be an argmax, or a random sample (the latter is generally preferred, however the former is easier to implement). The loss is now computed as before between the sequence of distributions `prob_dist(t)` and the ground truth sequence `Y`.

It is important to note that any teacher-forcing ratio above 0 introduces a mismatch between the decodes performed in training and those in inference. However, equally importantly, teacher-forcing ratios below 1.0 introduce a *misalignment* between the decoder output and ground truth output (i.e. the decoder may want to now output a longer or shorter sequence than the ground truth, and thus the output distribution may end up being compared to the wrong ground-truth character) and training will not converge.

So, we typically set tf initially to 1.0, and as the model converges and becomes more likely to generate an aligned output, the tf is gradually reduced, although reducing it below 0.5 will result in poor convergence and bad models.

The manner in which the tf is reduced with iterations is called teacher-force scheduling. Some pointers regarding teacher force scheduling:

- Avoid reducing your teach forcing rate below 0.5. Based on our observations, it does not work well most of the time.
- Reducing your teach forcing rate by small amounts consecutively seems to give better performance than reducing it by a larger amount.
- A low teach force rate with a high learning rate can cause your model to diverge suddenly. Monitor both parameters if you experience sudden divergence in the middle of training.
- Explore different scheduling methods - use learning rate schedulers as inspiration and have fun!

3.3.2 Cross-Entropy Loss with Padding

We use the cross-entropy loss to optimize the model. Since, with teacher forcing, there is one-to-one correspondence between the network output and ground truth, the X-ent loss is just the sum of the losses for individual symbols and can be computed using the following code:

Listing 9 Loss Computation

```
#prob_dist is a sequence of probability distribution vectors.
#Y(1:Tlen+1) is the ground truth sequence *including* the final <eos>. Y(0) is not considered
function Loss = XEntLoss(prob_dist, Y)
    loss = 0
    for t = 1:Tlen+1
        # The Xent loss is just the negative of the log probability of the target symbol
        loss = loss - log(prob_dist(Y(t))
    end
    return loss
end
```

The above code only computes the loss for a single instance, and is presented only as an explanation of loss computation. In practice, you will be training over minibatches, and the output will be a 3D array (a minibatch of arrays of probability distributions). Since you will use Pytorch's CrossEntropyLoss, you'll need to modify your output and label data as such to accommodate to these dimensions by reshaping. Make sure to study the Pytorch CrossEntropyLoss documentation for more information.

Next, it is important to note that your inputs will be of different lengths and will be padded to uniform length in a minibatch. The predictions and ground truth transcripts consist of padding indices that we created in our dataloader. Hence to calculate the loss, we need to mask out the padding indices so that they do not affect your gradients. We have multiple ways of doing this, which can be found in Appendix G.

3.3.3 Overall training

The overall training procedure is simple now:

```
# X is the input audio, T is its ground-truth transcription
# tf is the current setting of teacher-force ratio according to your scheduler
function gradient = ComputeGradient(X,T,tf)
    E = Listen(X)
    PO = TeacherForcedSpeller(E,T,tf)
    L = XEntLoss(PO, T)
    gradient = BackPropagate(L)
end
```

`BackPropagate()` is the gradient packpropagation that Pytorch's autograd will compute. The gradient is used to update the models. Observe how `TeacherForcedSpeller` is used instead of the regular speller, in the forward pass for training.

There is, however, a few more issues to factor in (we did promise you that this was a tricky homework).

- When the teacher-force ratio is less than 1, some of the inputs to the decoder are obtained by `DrawFromDistribution()` – these are shown by the dotted line in Figure 5(b). When backpropagating gradients, the derivatives must ideally also be backpropagated through this operation. However, `DrawFromDistribution(P)` is typically either an argmax or random sampling from `P`, neither of which is a differentiable operation if performed the usual way. In order to address that, we must modify these operations – by replacing embedding(argmax(P)) with embedding(softmax(P)), or by using the Gumbel-noise reparametrization as a replacement for sampling. We discuss this in Appendix O.
- While simply training as given above would work, it is generally better to *pretrain* the listener and speller separately before including them in the overall training. The lisetner can be pretrained either using a CTC criteron (as in HW3P2) or as an autoencoder. The speller can be pretrained as a language model as in HW4P1. These are discussed in Appendix Q.

4 Practical Implementation Details

You will have to implement the baseline (LAS) model to clear the early submission cutoff. Using the regularization methods suggested in M will help you clear the medium cutoff comfortably. Using data augmentation techniques as done previously in HW1P2 and HW3P2, with a good teacher forcing schedule will also help you reach the high cutoff.

You can also try using different types of attention - additive, scaled-dot product, cosine, multi-head or self attention. You are also welcome to try things such as language models to rescore your output as mentioned in Appendix Q, use CNNs/CNN blocks from HW2P2, or a transformer.

5 Conclusion

This homework is a true litmus test for your understanding of concepts in Deep Learning. It is not easy to implement, and the competition is going to be tough. But we guarantee you, that if you manage to complete this homework by yourself, you can confidently claim that you are now a **Deep Learning Specialist**!

Good luck and enjoy the challenge!

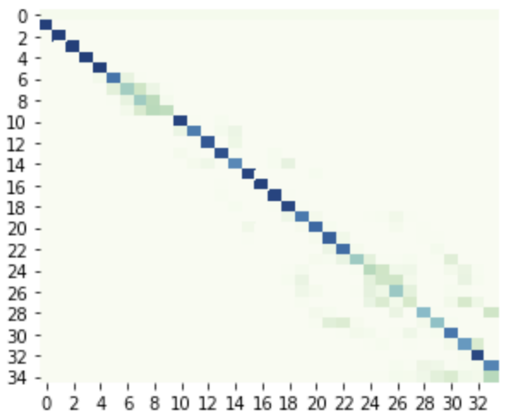
Appendices

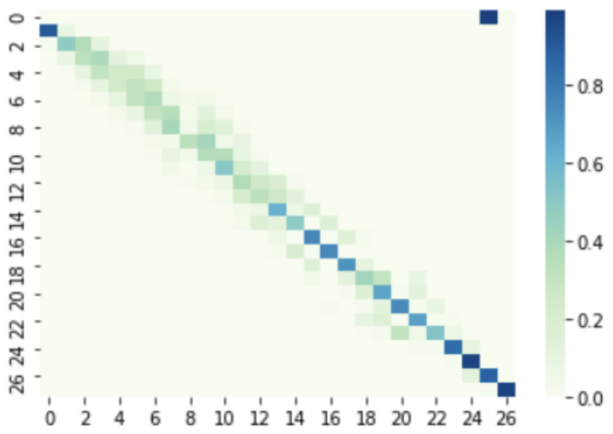
A Debugging

A.1 Toy Dataset

In addition to the real datasets, we provide a toy dataset - for debugging your attention implementation. The code for downloading it and the dataset will be given in the starter notebook. It has fixed length sequences, and the transcripts are phonemes, not characters. It is meant to give you an idea of how you can use attention for phonetic transcription, a step above HW3P2. Using the toy dataset as a test for convergence and ensuring you get the right attention plots will be extremely useful for your HW.

We attach two examples of attention plots you can expect from the toy dataset. Note that all toy utterances from a given split are stored in a single file as opposed to multiple files like the real training data.





Note that some portions of the plots are fuzzy and do not show clean lines (the right plot has more fuzzy areas than the left one). A correct attention implementation can result in both plots as well as fuzzier ones.

The toy dataset is simple and not reflective of the actual dataset. Please be mindful that the knowledge (e.g. hyperparameters, model architectures) you gain from playing around with the toy dataset may not transfer to the actual dataset. We recommend you use this to check for the correctness of the attention implementation and move on as soon as you see a diagonal shape that is close to the provided plots.

A.2 Mixed Precision

When implementing mixed precision, students typically overlook a common error - scaling and unscaling the optimizer when using gradient clipping. Please refer to this article by Pytorch that has code examples. Not unscaling the optimizer before gradient clipping can result in convergence delays.

B Character Based vs Word Based

We are giving you raw text in this homework, so you have the option to build a character-based or word-based model. Word-based models won't have incorrect spelling and are very quick in training because the sample size decreases drastically. The problem is, it cannot predict rare words.

The paper describes a character-based model. Character-based models are known to be able to predict some really rare words but at the same time they are slow to train because the model needs to predict character by character. In this homework, we **strongly recommend** you to implement a character based model, since:

- According to the statistic from last semester, almost all students will implement LAS or its variants, and LAS is a character based model.
- Most TAs are familiar with the character-based model, so you can receive more help for debugging.

C Transcript Processing

HW4P2 transcripts contain letters. You should make use of the `LETTER_LIST` array (consisting of all characters/letters in the vocabulary) provided in the starter notebook and convert the transcripts into their corresponding numerical indices to train your model.

Each transcript/utterance is a separate sample that is a variable length. We want to predict all characters, so we need a [start] and [end] character added to our vocabulary, which are `<sos>` and `<eos>`, respectively.

Hints/Food for thought:

- Think about special characters that are included in the vocabulary.
- You need to build a mapping from char to index, and vice versa. Reverse mapping is required to convert output, i.e., index, back to char.
- How do you deal with the length of the final output when evaluating on validation/test sets?

D Variable Length Inputs

Similar to HW3P2, you will be dealing with variable length transcripts as well as variable length utterances. We list one way you **should** deal with this problem, and another way that is useful for **debugging only**.

D.1 Built-in pack padded sequence and pad packed sequence

Idea: PyTorch already has functions you can use to pack your data. **Use this!**

- All the RNN modules directly support packed sequences.
- The slicing optimization mentioned in the previous item is already done for you! For LSTM's at least.
- Probably the fastest possible implementation.
- IMPORTANT: There might be issues if the sequences in a batch are not sorted by length. If you do not want to go through the pain of sorting each batch, make sure you put in the parameter 'enforce_sorted = False' in 'pack_padded_sequence'. Read the docs for more info.

D.2 Batch size one training instances

Idea: Give up on using mini-batches. **Only use this for debugging!**

- Trivial to implement with basic tools in the framework
- Helps you focus on implementing and testing the functionality of your modules
- Is not a bad choice for validation and testing since those aren't as performance critical.
- IMPORTANT: Once you decide to allow non-1 batch sizes, your code will be broken until you make the update for all modules.

E CNNs over pBLSTMs

As mentioned in earlier sections, some architectural variations you can try include using CNN layers with pBLSTMs, or replace pBLSTMs with CNNs, residual connections, etc. You can also try using a variation of your HW2P2 architectural blocks from networks such as ResNets, ConvNeXt, MobileNet, etc. Feel free to experiment and potentially enlighten us with an architecture that works well with CNNs over/with pBLSTMs! Again, keep in mind that we don't recommend downsampling by a factor higher than 8.

F Single Head Attention (Minibatch)

The following pseudocode describes context computation, as outlined in Section 3.1.2, for a minibatch. We use loops in the pseudocode – actual python code would be smarter and use tensor operations.

Listing 10 Attention (minibatch)

```
# E(:,:,:) = batchsize x max_sequence_length x dim array of embeddings for a minibatch
# mask(:,:) = batchsize x max_sequence_length array of binary masks (1 for valid, 0 for invalid)
# q(:,:) = batchsize x Dq matrix of query vectors
# C(:,:) = batchsize x Dv matrix of computed context vectors
# attention_weights = batchsize x max_sequence_length array of attention weights
function C, attention_weights = Attend(E, q, mask)
    # WK and WV must also be specified in actual code
    query_length = length(q(1,:))
    for b = 0:batchsize-1 # Goes over the entire minibatch
        instance_embeddings = E(b,:,:)
        K = instance_embedding * WK      # WK is a dim x Dq matrix
        V = instance_embedding * WV      # WV is a dim x Dv matrix
        raw_weights = (1/sqrt(query_length)) * q(b,:) * transpose(K)
        unmasked_attention_weights = softmax(raw_weights)
        # Attention for masked-out embeddings must be set to 0
        masked_unnormalized_weights = unmasked_attention_weights * mask
        # Masking will result in weights that dont sum to 1. Renormalize to sum to 1.
        masked_attention_weights = unmasked_attention_weights / sum(unmasked_attention_weights)
        attention_weights(b,:) = masked_attention_weights
        C(b,:) = masked_attention_weights * V
    end
    return C, attention_weights
end
```

G Masking Padding Indices

Here we provide some more detail on different ways to come up with a mask to discard padding indices in your predictions/attention.

G.1 Method 1 - For Loop

You can use a nested for loop to create a mask by looping through the batch, retrieving the label length for the instance, and setting all time instances before the label length as true and all time instances after label length as false. If you use this approach, your code should look something like this (again, this is psuedocode!):

Listing 11 Mask using For Loop

```
#let B be batch size, let T be time step length
mask = array of size B*T
for b in range(B)
    length = label length of instance
    for i in range(T):
        if i < length:
            #you guys can figure this out :)
        else:
            # ????
```

G.2 Method 2 - Boolean Mask

This method involves creating a boolean mask over all the sequences in a minibatch such that for each sequence, the corresponding mask has values that are True for all positions that aren't padding, and False for all padding indices (vice-versa also works). If you take this approach, your code should look something like this (this is pseudocode!):

Listing 12 Boolean Mask using broadcasting

```
#let B be batch size, T be time step length
#let sequenceLength be a list of sequence lengths of each transcript in the batch of size B
lst = #how can we make an list of length T that starts at 0 and counts up to T-1?
mask = lst < sequenceLength #or lst >= sequenceLength
#The code above by itself will cause dimension issues - how can you use squeeze/unsqueeze lst
    and/or sequenceLength to fix this?
```

For masking your loss calculation, you have a few options:

- Use the mask to index the relevant parts of your inputs and targets, and use those only for loss.
- Send your inputs and targets to the loss calculation (set reduction='none'), then use the Boolean mask to zero out anything that you don't care about.
- Use the ignore index parameter in Xentropy Loss and set all of your padding to a specific value.

For masking your attention, we suggest you use the `masked_fill` function from Pytorch to replace the masked indices with a value of choice. Remember, you want to make sure your padding has no impact when performing the softmax operation. (Hint: Think about what value to use for `masked_fill`, such that the softmax of padding indices becomes 0).

G.3 Method 3 - PackedSequence

The most straightforward and easy way to mask your loss is to use PackedSequence. If you remember how we have used packing and padding so far, we pad sequences to a fixed length and pack the output sequences from the model with the respective array of sequence lengths. Following this line of thought, we can pack the predictions obtained from our model, and the ground truth transcripts from the dataloader, and use those to calculate the loss. If your inputs and outputs are Packed-Sequence, then you can run your loss function on sequence.data directly. Refer to Pytorch's PackedSequence documentation for more information.

```
# ly is the lengths array from the dataloder
predictions = model(input)
packed_predictions = PackPaddedSequence(predictions, ly, enforce_sorted=False)
packed_transcripts = PackPaddedSequence(y, ly, enforce_sorted=False)
loss = XentropyLoss(packed_predictions.data, packed_transcripts.data)
```

Typically, we will use the sum over the sequence and the mean over the batch. That means take the sum of all of the losses (modified by the mask) and divide by batch size. If you're wondering "why" consider this: if your target is 0 and your predicted logits are a uniform 0, is your loss 0 or something else?

The concept to keep in mind is that your mask should be reflective of the corresponding lengths of the ground truth transcripts, i.e your mask should negate all values from the padding that comes after the length of the ground truth transcripts. Easiest way to check if you have the right mask would be to make sure the sum of your mask (or the negated mask, depending on how you implemented it) along the sequence length dimension adds up to the lengths of the ground truth transcripts.

H Multi-Head Attention

The attention computed in (Section 3.1.2) computes a *single* context vector C_k to compute output O_k. C_k derives and focuses on a single specific aspect of the input that is most relevant to O_k. It is reasonable to assume that there are *multiple* separate aspects of the input that must all be considered to compute O_k.

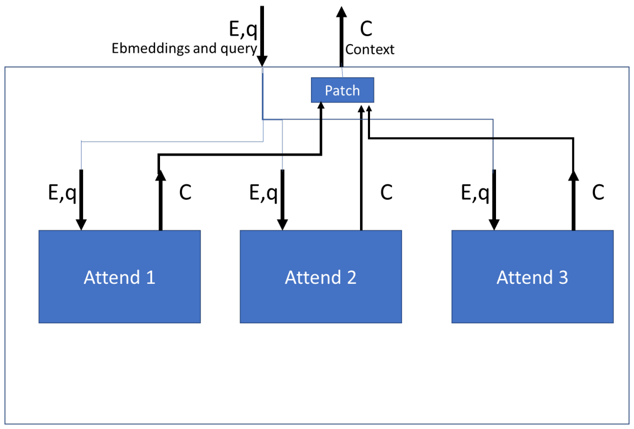


Figure 6

Multi-head attention works on this principle to derive multiple context vectors $C_k^1, C_k^2, \cdots, C_K^L$. Each context vector C_k^l is computed using a separate attention module of the form explained above (Section 3.1.2), with its own learnable parameters $\mathbf{W}_K^l, \mathbf{W}_V^l, and \mathbf{W}_Q^l$. The complete module that computes an individual context vector C_k^l is called an *attention head*, thus the overall attention framework that computes multiple context vectors is referred to as *multi-head* attention.

The final context vector C_k that is used to compute O_k is obtained by concatenating the individual context vectors: $C_k = [C_k^1, C_k^2, \cdots, C_k^L]$ (where L is the number of attention heads).

Following is the pseudocode for multi-head attention:

Listing 13 Multihead Attention

```
# E = (sequence_length, dim) array of embeddings for \textit{single} input
# q = query vector (assumed to be a (1,Dq) row vector)
# context = context vector computed (1,Dv) row vector)
# no_of_heads = number of attention heads
function Attention = MultiHead(E, q)
    key = E * WK # WK is a (dim, Dk) matrix
    value = E * WV # WV is a (dim, Dv) matrix
    query = query * WQ # WQ is a (dim, Dq) matrix
    # Reshape Q,K,V to compute attention in parallel for each head
    query = reshape(query, (1, no_of_heads, Dq//no_of_heads))
    key = reshape(key, (sequence_length, no_of_heads, Dk//no_of_heads))
    value = reshape(value, (sequence_length, no_of_heads, Dv//no_of_heads))
    # Transpose to (no_of_heads, 1, Dkv//no_of_heads)
    query = transpose(query)
    # Transpose to (no_of_heads, sequence_length, Dkv//no_of_heads)
    key = transpose(key)
    value = transpose(value)
    mask = ? #Similar to single head attention, but now have one for each head
    context, attention_weights = Attend(query,key,value,mask) #Assuming Attend works on minibatches
    context = reshape(context, (1, Dv))
    context = context * W_O #W_O is a (Dv,Dv) matrix
    return context, attention_weights
end
```

I Speller (minibatch)

The following pseudocode generalizes the implementation of Speller found in 3.1.3 for minibatches.

Listing 14 Speller (minibatch)

```
# hidden_states(:,:,:) = (batchsize,timestep,lstm_hidden_size) array of lstm hidden states
# query(:,:) = (batch_size, Dq) matrix of query vectors
# context(:,:,:) = (batchsize, timestep,Dv) matrix of computed context vectors
# prob_dist(:,:,:) = (batch_size,timestep,vocab_size) array of probability distributions
# output_symbols(:,:,:) = batch_size x timestep x 1 array of output symbols for each sequence
# predictions = (batch_size,) list of variable length sequences of predicted transcripts
function predictions = Speller(E)
    hidden_states(:,0,:) = [[0,0 ...], ... [0,0,...]]
    output_symbols(:,0,:) = [<sos>, <sos> ... ]
    prob_dist = []
    query = hidden_states(:,0,:)
    context(:,t,:) = attention(E, query) #or, alternaetely, as V[0]
    for t = 0:Tmax
        hidden_states(:,t,:) = LSTMstep(output_symbols(:,t-1,:), hidden_states(:,t-1,:),
            context(:,t-1,:))
        query = hidden_states(:,t,:)
        context(:,t,:) = attention(E,query)
        prob_dist(:,t,:) = Character_Distribution_Net([context(:,t,:), hidden_states(:,t,:)])
        output_symbols(:,t,:) = DrawFromDistribution(prob_dist(:,t,:))
        predictions = concat(predictions, output_symbols(:,t,:))
    return predictions
end
```

J Inference - Random Search

As we saw in Section 3.2.2, random search would involve generating several decodes by rerunning the decoder multiple times. We outline a procedure for you to do it as follows -

- Pass only the utterance and the `<sos>` character to your model.
- Generate text from your model by sampling from the predicted distribution for some number of steps.
- Generate many samples in this manner for each test utterance (100s or 1000s). You only do this on the test set to generate the Kaggle submission so the run time shouldn't matter.
- Calculate the sequence lengths for each generated sequence by finding the first `<eos>` character.
- Now run each of these generated samples back through your model to give each a loss value.
- Take the randomly generated sample with the best loss value, optionally re-weighted or modified in some way like the paper.

K Inference - Beam Search

Ideally, to select the most probable output sequence we must explicitly evaluate all possible sequences and select the most likely. This is infeasible. Instead, we use a beam search, which iteratively expands out the K most probable paths, until it finds a most likely path that ends in a `<eos>`. This is the method chosen in the LAS paper. Note that if you want to implement this, you are not allowed to use any third party packages. Implementing this will involve not using the basic decoder and have a completely different approach to the decoding strategy given in 3.1.3.

We provide pseudocode for beam search here, for you to get started:

Listing 15 LSTM Beam Search

```
# k = beam_width
function beam_decoder(E):
    O(t) = <sos>, query = [0,0,...]
    log_probs = 0.0, init_hidden_states = 0
    out_seq = [] #List to maintain output sequence of symbols for path
    path_end = False #Boolean to signify if we path reached <eos>
    initial_context = attention(E, query) #or, alternatively as V[0] or zeros
    beams = [[log_probs, out_seq, hidden_states, initial_context, path_end]] #Super list of all
         paths

    for t = 1:Tmaxlength-1: #Loop over timesteps till max length
        new_beams = [] #Temporary list of paths for current sequence at timestep t

        for path in beams: #Loop over all paths in your beams list
            #Each path has log_probs, out_seq, hidden_states, context and path_end as attributes
            if path has encountered <eos>:
                #Save path, continue to the next
            else:
                #Search not concluded if all paths have not encountered <eos> symbol yet.
                #How can you maintain a check for this condition?

            S(t-1) = hidden_states.copy() #Copy of hidden_states from current path list
            #We do this to use the last hidden state of path so far to get the next symbol from
                 LSTMStep, in our search

            S(t) = LSTMStep(O(t-1), S(t-1), C(t-1))
            query = S(t)
            C(t) = attention(E, query)
            PO(t) = Character_Distribution_Net([C(t), S(t)])
            candidate_paths = topkpaths(log_softmax(PO(t))) #Topkpaths gets the top beam_width
                 possible output symbols from the log softmax of the probabilities in the
                 distribution

            for i = 0:k #Loop over each possible output symbol until beam_width
                path_symbol = #Get output symbol for path i from candidate paths
                path_log_prob = #Get log probabilities of path i from candidate paths

                #Add new path to new_beams as follows:
                #update path log probabilities, path sequence with new output path symbol,
                #update hidden states and path_end if search for this path has ended with <eos>

        beams = prune(sort(new_beams)) #Sort new beams by probability score and prune top k

        #if all paths have encountered <eos>, break here

    #return concatenated output sequence of top path in beams
```

L Levenshtein Distance

Kaggle will evaluate your outputs with the Levenshtein distance, aka edit distance : number of character modifications needed to change the sequence to the gold sequence. To use that on the validation set, you can use the python-Levenshtein package.

Implementing it is fairly similar to how you did it in HW3P2. You would first need to create the predicted transcript based on your inference method for each item in the batch. Calculate and sum the Levenshtein distance between the predicted transcript and the ground truth for each pair, during your inference loop. Finally, average the sum to obtain the levenshtein distance for the entire batch. Make sure you do not include `<sos>` and `<eos>` tokens in your transcripts when calculating the distance and test predictions.

M Weight Initialization and Regularization

As you may have noticed in HW3, a good initialization significantly improves training time and the final validation error. We recommend referring to the LAS paper for initialization strategies they have mentioned and using the same for this task. You are welcome to explore different initialization strategies either way.

As you have done in HW4P1, you will have to use regularization methods including Locked Dropout, Weight tying, Embedding Dropout, etc. Using these techniques is key to achieving great performance on your model. Using Data Augmentation is also a key way to achieving better performance. We recommend using time masking and frequency masking as you may have used in previous homeworks.

N Optimizer and Learning Rate

From our empirical studies, we recommend you use Adam or AdamW and start with a learning rate of $1e^{-3}$, and experiment with the scheduling methods. On the other hand, the LAS paper uses ASGD as the optimizer with a learning rate of 0.2. The learning schedule involved a geometric decay of 0.98 per 3M utterances (i.e., 1/20-th of an epoch). Feel free to try that if it works for you.

O Gumbel Noise

One way to deal with the problem addressed in Section 3.3.3, is by using the Gumbel Noise Reparameterization Trick. We reparameterize our output samples, as coming from a sum of the mean of a probability distribution (gumbel distribution) plus some stochastic noise. We can then use argmax to draw the outputs. The gumbel distribution is specifically used as it is stable under max operations. This is also equivalent to computing the softmax over a set of stochastically sampled outputs. You can implement gumbel noise easily using Pytorch (refer to the gumbel softmax function).

P Pretraining Listener

You can pretrain the listener in the baseline model with an autoencoder architecture with a separate encoder and decoder block. You would separately train the Listener with the audio input to get hidden representations and use a decoder block consisting of 1-d transpose CNNs that dilate the embeddings back to the input dimensions, using L2 loss as your divergence metric. It should help make training faster in terms of iterations required for convergence.

Q Language Models and Pretraining Speller

In HW4P2, you can train a language model to model the transcripts (similar to just HW4P1).One strategy is to pre-train the speller (decoder) before starting the real training. You can use the train transcripts as a train data and train your model like you would train a language model (e.g. HW4P1). During the pre-training stage, you can set the attended context to zeros or random values, concatenate with your input embeddings, and train with a language modeling objective.

Another strategy is to train LAS and add its outputs to your trained Language Model outputs at each time-step. LAS is then only trying to learn how utterances change the posterior over transcripts, and the prior over transcripts is already learned. It should make training much faster (not in terms of processing speed obviously, but in terms of iterations required for convergence).