Homework 4 Part 2

Attention-based End-to-End Speech-to-Text Deep Neural Network

11-785: Introduction to Deep Learning (Fall 2021)

OUT: **November 05, 2021**
DUE: **November 29, 2021, 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:**
  - **Part 1**: All of the problems in Part 1 will be graded on Autolab. You can download the starter code from Autolab as well. Refer to the Part 1 write-up for more details.
  - **Part 2**: You only need to implement what is mentioned in the write-up 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.

1 Introduction

In the last Kaggle homework, you should have understood how to predict the next phoneme in the sequence given the corresponding utterances. In this part, we will be solving a very similar problem, except, you do not have the phonemes. You are ONLY given utterances and their corresponding transcripts.

In short, you will be using a combination of Recurrent Neural Networks (RNNs) / Convolutional Neural Networks (CNNs) and Dense Networks to design a system for speech-to-text transcription. End-to-end, your system should be able to transcribe a given speech utterance to its corresponding transcript.

2 Getting Started Early

This homework can take a significant amount of your time, so please start early. The baseline is already given, so you know what will work. **Note that using attention is mandatory.** We are aware of certain model architectures that give a high performance without using attention, and this is explicitly prohibited as implementing attention is a crucial part of this assignment. However, you are not required to use the same kind of attention as the one we presented. For those familiar with self-attention, multi-headed attention, the transformer network, or whatever else, you are free to implement any version you want. Except for adding data from an extra source, you are allowed to explore any architecture.

3 Dataset

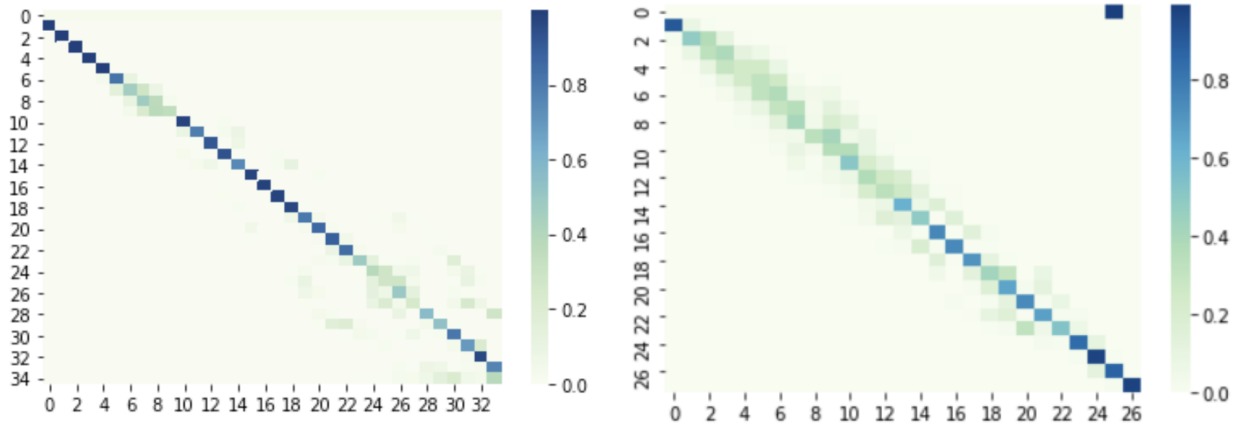
You will be working on a similar dataset again. You are given a set of 6 files:

- `train.npz`: The training set contains training utterances each of variable duration and 40 frequency bands.

- `dev.npz`: The development set contains validation utterances each of variable duration and 40 frequency bands.
- `test.npz`: The test set contains test utterances each of variable duration and 40 frequency bands. There are no labels given for the test set.
- `train_transcripts.npz`: These are the transcripts corresponding to the utterances in train.npz. These are arranged in the same order as the utterances.
- `dev_transcripts.npz`: These are the transcripts corresponding to the utterances in dev.npz. These are arranged in the same order as the utterances.
- `hw4p2_toy_dataset.zip`: In addition to the data above, we provide a toy dataset you could use for debugging purpose. You are encouraged to use it to check the correctness of your attention implementation. Please refer to Section 3.1 for more details.

3.1 Debugging Tool

In addition to the real datasets, we provide a toy dataset - `hw4p2_toy_dataset.zip` - for debugging your attention implementation (can be downloaded from Kaggle). This is a small and simple dataset, which has a random sequence of characters as its input and the same as an output. To test with this dataset, we encourage you to first strip down your model to its base format, i.e., no dropout, no batch-norm, and a 1-layer LSTM in both encoder and decoder. We attach two examples of attention plots you can expect from the toy datasets.



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.

4 Approach

There are many ways to approach this problem. In any methodology you choose, we require you to use an attention-based system like the one mentioned in the baseline (or another kind of attention) so that you achieve good results. Attention Mechanisms are widely used for various applications these days. More often than not, speech tasks can also be extended to images. If you want to understand more about attention, please read the following papers (you can click the texts):

- "Listen, Attend and Spell"
- "Show, Attend and Tell (Optional)"
- Annotated "Attention is All You Need"

4.1 LAS

The baseline model for this assignment is described in the "Listen, Attend and Spell paper". The idea is to learn all components of a speech recognizer jointly. The paper describes an encoder-decoder approach, called Listener and Speller respectively.

The Listener consists of a Pyramidal Bi-LSTM Network structure that takes in the given utterances and compresses it to produce high-level representations for 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.

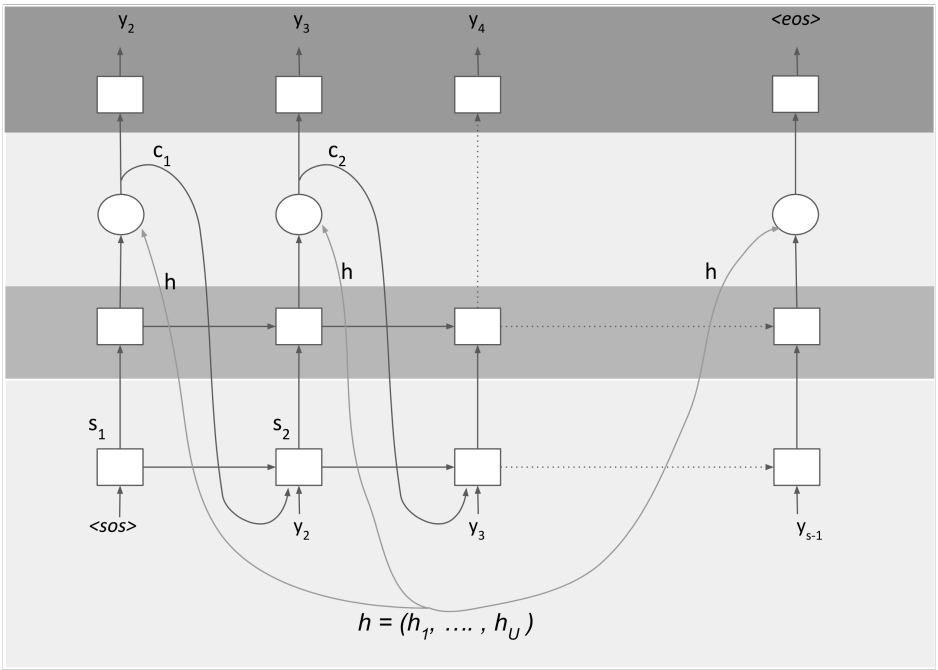
Attention intuitively can be understood as trying to learn a mapping from a word vector to some areas of the utterance map. The Listener produces a high-level representation of the given utterance and the Speller uses parts of the representation (produced from the Listener) to predict the next word in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

This system in itself is powerful enough to get you to the top of the leader-board once you apply the beam search algorithm (no third-party packages, you implement it yourself).

Warning : You may note that the Figure 1 of the LAS paper isn't completely clear w.r.t what the paper says to do, or even contradictory. In these cases, follow the formulas and the fiture we provided in this write-up. The ambiguities are :

- In the speller, the context c_{i-1} should be used as an extra input to the RNN's next step : $s_i = RNN(s_{i-1}, y_{i-1}, c_{i-1})$. If you use PyTorch's LSTMCell, the simplest is to concatenate the context with the input : $s_i = \text{LSTMCell}(s_{i-1}, [y_{i-1}, c_{i-1}])$. The figure seems to concatenate s and c instead, which makes less sense.
- As the paper says, the context c_i is generated from the output of the (2-layer) LSTM and the Listener states, and then directly used for the character distribution, i.e. in the final linear layer. The figure makes it look like the context is generated after the first LSTM layer, then used as input in the second layer. Do not do that.
- The listener in the figure has only 3 LSTM layers (and 2 time reductions). The paper says to use 4 (specifically, one initial Bi-LSTM followed by 3 pLSTM layers each with a time reduction). We recommend that you use the 4-layer version - at least, it is important that you reduce the time resolution by roughly 8 to have a relevant number of encoded states.

We provide a more accurate figure to make things clearer :



Additionally, if the computation of energy, i.e.,

$$e_{i,u} = \langle \phi(s_i), \phi(h_u) \rangle$$

does not seem clear to you, it refers to a scalar product between vectors :

$$\langle U, V \rangle = \sum_{k=1}^{n} U_i V_i$$

You will have to perform that operation on entire batches with sequences of listener states; try to find an efficient way to do so. You are free to explore any other forms of attention as per the annotation in the notebook we provided.

4.2 LAS - Variant

It is interesting to see that the LAS model only uses a single projection from the Listener network. We could instead take two projections and use them as an Attention Key and an Attention Value. It's actually recommended.

Your encoder network over the utterance features should produce two outputs, an attention value and a key. In addition, your decoder network will produce an attention query.

We will call the product between that query and the key the energy of the attention. We then need to apply a mask to the energy to reduce the impact of padding in our softmax. There are multiple ways to go about creating the mask - you can used a nested for loop or broadcasting (study section 5.7 for pseudocode - the less than or equal sign in the broadcasting example may or may not be correct to use in this mask....). To apply the mask to the energy, we suggest you look up the masked fill function in pytorch.

Feed the masked energy into a softmax, the result of which will be our attention. Finally, compute the product of the attention and the attention value. This final result is called our context, which is fed back into your transcript network.

This model has shown to give amazing results, we **strongly recommend** you to implement this in place of the vanilla LAS baseline model.

4.3 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 it 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.

5 Implementation Details

5.1 Variable Length Inputs

This would have been a simple problem to solve if all inputs were of the same length. You will be dealing with variable length transcripts as well as variable length utterances. There are many ways in which you can deal with this problem. Below we list down one way you **shouldn't use** and another way **you should**.

5.1.1 Batch size one training instances

Idea: Give up on using mini-batches.

Pros:

- 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.

Cons:

- Once you decide to allow non-1 batch sizes, your code will be broken until you make the update for all modules. Only good for debugging.

5.1.2 Use the built-in pack padded sequence and pad packed sequence

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

Pros:

- 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.

5.2 Transcript Processing

HW4P2 transcripts are a lot like HW4P1, except we did the processing for you in HW4P1. That means you are responsible for reading the text, creating a vocabulary, mapping your text to NumPy arrays of ints, etc. Ideally, you should process your data from what you are given into a format similar to HW4P1.

Also, 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. You can also make them both the same number, like 0, to make things easier. If the utterance is "hello", then:

inputs=[start]hello
outputs=hello[end]

Hints:

- You may want to decide between character-based and word-based (Section 4.3). Depending on your choice, your vocabulary will be different.
- Think about special characters that need to be included in the vocabulary.
- Whichever vocabulary you choose, you need to build a mapping from char/word to index, and vice versa. Reverse mapping is required to convert output, i.e., index, back to char/words.
- Think how you want to deal with the length of the final output when evaluating on validation/test sets.

5.3 Listener - Encoder

Your encoder is the part that runs over your utterances to produce attention values and keys. This should be straight forward to implement. You have a batch of utterances, you just use a layer of Bi-LSTMs to obtain the features, then you perform a pooling like operation by concatenating outputs. Do this three times as mentioned in the paper and lastly project the final layer output into an attention key and value pair.

pBLSTM Implementation:

This is just like strides in a CNN. Think of it like pooling or anything else. The difference is that the paper chooses to pool by concatenating, instead of mean or max.

You need to transpose your input data to (batch-size, length, dim). Then you can reshape to (batch-size, length/2, dim*2). Then transpose back to (length/2, batch-size, dim*2).

All that does is reshape data a little bit so instead of frames 1,2,3,4,5,6, you now have (1,2),(3,4),(5,6).

Alternatives you might want to try are reshaping to (batch-size, length/2, 2, dim) and then performing a mean or max over dimension 3. You could also transpose your data and use traditional CNN pooling layers like you have used before. This would probably be better than the concatenation in the paper.

Two common questions:

- What to do about the sequence length? You pooled everything by 2 so just divide the length array by 2. Easy.
- What to do about odd numbers? Doesn't actually matter. Either pad or chop off the extra. Out of 2000 frames one more or less shouldn't really matter and the recordings don't normally go all the way to the end anyways (they aren't tightly cropped).

5.4 Speller - Decoder

Your decoder is an LSTM that takes word[t] as input and produces word[t+1] as output on each time-step. The decoder is similar to hw4p1, except it also receives additional information through the attention context mechanism. As a consequence, you cannot use the LSTM implementation in PyTorch directly, you would

instead have to use LSTMCell to run each time-step in a for loop. To reiterate, you run the time-step, get the attention context, then feed that in to the next time-step.

5.5 Teacher Forcing

One problem you will encounter in this setting is the incapability of your model at the early training stage. At the very beginning, your model is likely to produce all wrong predictions. Since the previous prediction will be fed as input to the next time step, if it is wrong, the next prediction might be also wrong. This is a vicious circle. To alleviate this issue, we can set a probability to reveal the ground-truth of each time step to the model to accelerate its learning. This is known as teacher forcing.

What happens if we are at the very beginning of a sequence and we want to reveal the "ground-truth" of the previous time step? This might sound confusing since there aren't any other previous time steps. To simplify this dilemma, we'll just pass in ¡sos¿ - there is no other ground truth that would make any logical sense to pass in!

You can start with .90 teacher forcing rate in your training. This means that with .90 probability you pass in the ground truth char/word from the previous time step, and with .10 probability you will pass in the generated char/word from the previous time step.

5.6 Gumbel Noise

Another problem you will be facing is that given a particular state as input to your model, the model will always generate the same next state output, this is because once trained, the model will give a fixed set of outputs for a given input state with no randomness. To introduce randomness in your prediction, you will want to add some noise into your prediction (only during generation time) specifically the Gumbel noise.

5.7 Cross-Entropy Loss with Padding

First, you have to generate a Boolean mask indicating which values are padding and which are real data. If you have an array of sequence lengths, you can generate the mask on the fly. The comparison range(L) (shaped L, 1) < sequence lengths (shaped 1, N), (in other words, range(L) < seq-lens) will produce a mask of true and false of the shape (L,N), which is what you want. That should make sense to everybody. If you have the numbers from 0-L and you check which are less than the sequence length, then that is true for every position until the sequence length and false afterwards. If you take this approach, your code should look something like this (this is pseudocode!):

```
#let B be batch size
#let T be time step length
#let sequenceLength be a list of the real sequence lengths of each instance 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
#The code above by itself will cause dimension issues - how can you use squeeze/unsqueeze lst
    and/or sequenceLength to fix this?
```

You can also 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!):

```
#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:
        # ????
```

Now you have at least three options:

- Use the Boolean mask to index the relevant parts of your inputs and targets, and send just those to the loss calculation
- 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 and set all of your padding to a specific value.

There is one final interesting option, which is using PackedSequence. If your inputs and outputs are PackedSequence, then you can run your loss function on sequence.data directly. Note sequence.data is a variable but variable.data is a tensor.

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?

Another point to consider is the dimensions of your model's output. Your output will come out as a 3D array, unlike the 2D array we are used to! You'll need to modify your output and label data as such to accommodate to these dimensions, either by reshaping or transposing. Make sure to study the Pytorch CrossEntropyLoss documentation for more information.

5.8 Inference - Greedy Search

This is the easiest decoding method. It is trivial to implement and will promise you a decent enough result. Please using greedy search to make sure you system is working before attempting other advanced decoding methods.

5.9 Inference - Random Search

If you have implemented the baseline and wish to improve the performance further, you could try random search. How to do that?

- Pass only the utterance and the [start] 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 [end] 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

Much easier than a beam search and results are also pretty good as long as you generate enough samples which shouldn't be a problem if you code it efficiently and only run it once at the end.

But if you really want to squeeze every bit of value you can, implement beam search yourself, it is a bit tricky, but guaranteed to give good results. Note that you are not allowed to use any third party packages.

5.10 Character based - Implementation

Create a list of every character in the dataset and sort it. Convert all of your transcripts to NumPy arrays using that character set. For Kaggle submissions, just convert everything back to text using the same character set.

5.11 Good Initialization

As you may have noticed in HW3, a good initialization significantly improves training time and the final validation error. In general, you should try to apply that idea to any task you are given. In HW4P2, you can train a language model to model the transcripts (similar to just HW4P1). You can then train LAS and add its outputs to you 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).

An alternative with similar effects 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 some random values, concatenate with your input embeddings, and train with a language modeling objective.

5.12 Layer Sizes

The values provided in the LAS model (listener of hidden size 256 per direction, speller of hidden size 512) are good, no need to try something larger. The other sizes should be smaller (for example, attention key/query/value of size 128, embedding of size 256). Adding tricks such as random search, pre-training, locked dropout, will have more effect than increasing sizes higher than that.

Attention models are hard to converge, so it's recommended to start with a much smaller model than that in your debugging phase, to be sure your model is well-built before trying to train it. For example, only one layer in the speller, 2 in the listener, and layer sizes twice smaller.

6 Evaluation

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.

During training, it's good to use perplexity (exponential of the loss-per-word) as a metric, both on the train and validation set, to quantify how well your model is learning.

7 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 walk up anywhere and claim that you are now a Deep Learning Specialist!

Good luck and enjoy the challenge!