Homework 4 Part 2

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

11-785: Introduction to Deep Learning (Fall 2019)

OUT: **Nov 10, 2019**
DUE: **Dec 5, 2019, 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 Dataset

You will be working on the WSJ dataset again. You are given a set of 5 files train.npy (2.4 GB), dev.npy (105.81 MB compressed), test.npy (58.27 MB), train_transcripts.npy (5.68 MB), and dev_transcripts.npy (247.16 kB).

- `train.npy`: The training set contains training utterances each of variable duration and 40 frequency bands.
- `dev.npy`: The development set contains validation utterances each of variable duration and 40 frequency bands.
- `test.npy`: 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.npy`: These are the transcripts corresponding to the utterances in train.npy. These are arranged in the same order as the utterances.

- `dev_transcripts.npy`: These are the transcripts corresponding to the utterances in dev.npy. These are arranged in the same order as the utterances.

3 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:

- Listen, Attend and Spell
- Show, Attend and Tell (Optional)

3.1 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.

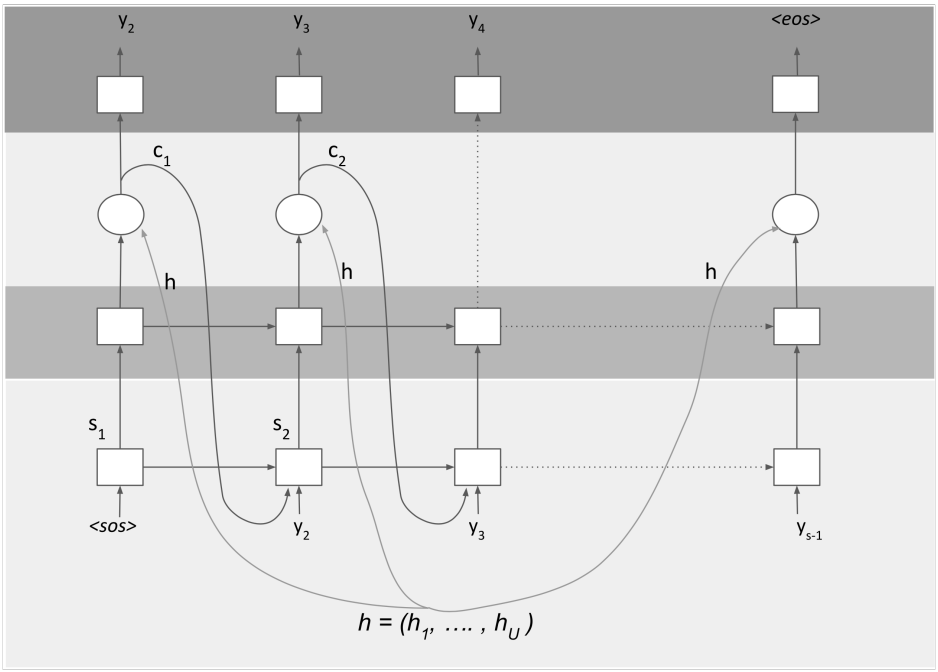
Attention intuitively can be understood as trying to learn a mapping from a word vector to some areas of the utterance map. The Listener produces a high-level representation of the given utterance and the Speller uses parts of the representation (produced from the Listener) to predict the next word in 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, not the figure. 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 : $s_i = LSTMCell(s_{i-1}, [y_{i-1}, c_{i-1}])$. 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 BLSTM 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 operation (9)

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

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

$$\langle U, V \rangle = \sum_{k=0}^{n} U_i V_i$$

You will have to perform that operation on entire batches with sequences of listener states; try to find an efficient way to do so.

3.2 LAS - Variant 1

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 and your decoder network over the transcripts will produce an attention query. We are calling the dot product between that query and the key the energy of the attention. Feed that energy into a Softmax, and use that Softmax distribution as a mask to take a weighted sum from the attention value (apply the attention mask on the values from the encoder). That is now called attention 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.

3.3 LAS - Variant 2

The baseline model implements a pyramidal Bi-LSTM to compute the features from the utterances. You can conveniently swap the entire Listener block with any combination of LSTM/CNN/Linear networks. This model is interesting to try once you have the baseline working.

3.4 Character Based vs Word Based

We are giving you raw text in this homework. You are free to build a character-based or word-based model. LAS model is, however, a character-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.

4 Implementation Details

4.1 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. We will list down some ways ordered by the difficulty of implementation, there will be a trade-off between the implementation complexity, model performance and computational speed. We suggest you to implement 4.1.5 or 4.1.6, but you get to pick your poison.

4.1.1 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
- No need to worry about padding or three-dimensional matrix operation, or sorting by length
- Can still run pretty quickly for sequences that have high-dimensional elements, or when you matrix multiply two complete sequences.
- Is not a bad choice for validation and testing since those aren't as performance critical.

Cons:

- Runs very slowly when you're doing mostly linear-time processing with lower-dimensional elements.
- It's very easy to accidentally apply simplifications to your code that aren't valid when the batch size is more than 1, making it much harder to upgrade to a better method.
- Once you decide to allow non-1 batch sizes, your code will be broken until you make the update for all modules.

4.1.2 Pass in lists and loop over each instance

Idea: Run batch size one code in a loop across instances in a batch. Training data is a Python list and not a tensor.

Pros:

- Almost just as simple as batch size one method, and with slightly better performance.
- Custom collate function needed, but it's very easy to implement.
- When you decide to vectorize your code across batches, you can keep the for loop implementation for modules that aren't performance critical.

- Even after you implement padding, it's easy to build a list back from it for any loop-based code.

Cons:

- Still very slow for many problems.
- You still end up with code that might be very hard to rewrite into vectorized forms.

4.1.3 Use padded data and masks

Idea: Perform the full computation on padded values, and mask away the unneeded values at the end.

Pros:

- Actually uses vectorization across batches, so there is potential for very good performance.
- Computation of a mask is not very hard to do manually.

Neutral:

- Custom collate function needed to pad the data, but this is a pretty standard requirement for all the faster methods.
- It's sometimes simpler to use a list of sequence lengths instead of a binary mask. The mask is only needed if zero padding can affect the results.

Cons:

- Need to be very careful not to let padding values affect your results. Many vectorized functions are affected by zeros too. Makes it harder to use library functions.
  - Softmax breaks when some of its inputs are 0.
  - Averaging needs a dot product with the mask.
  - Backward pass of bidirectional LSTM will read lots of zeros before reading your true data.
  - You need to explicitly do 2 one-directional LSTM's, with two inputs where one has reversed data (but padding in the same locations).
- Need to pass a mask through all the modules if you have any nontrivial computations.
- When sequence lengths are very different in a batch, you end up with lots of unnecessary computation.

4.1.4 Sort all the training data by sequence length, then mask

Idea: By sorting the training data, we keep the training instances similar in length.

Pros:

- Faster computation than normal masking method.
- Sorting gives us a second benefit. For each position in the time dimension, only a consecutive range of training instances contain valid data. This lets us use vectorized operations on just the valid training instances just by slicing data, which is very useful if we needed to for-loop over the sequence dimension anyway (sound familiar?).

Cons:

- You need to be careful with the data loader. Either you randomly generate an offset and choose several consecutive training instances from that offset (requires a fully hand-written data loader), or you give up on randomization entirely.
- Don't forget to shuffle the labels in the same you way you shuffled the data! If you forget, the network will be learning garbage and it'll be really hard to figure out why.

- If you have multiple sequences with unrelated lengths, you can only choose to sort one. You'll still need to use a different approach to deal with the unsorted sequences.

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

Idea: PyTorch already has functions you can use to pack your data.

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.

Cons:

- These functions will only work if you sort the sequences by length.
  - You can choose to sort the whole dataset, which would require crafting your own data loader logic.
  - You can sort each batch separately, remembering to shuffle the labels everytime as well.
- It's still annoying to deal with multiple sequences of unrelated lengths. We can't use pack/unpack for all but one of the sequences.
  - You'd have to be really careful to work around the sorting of one of the sequences. Sorting-related bugs can be very nasty and hard to find.
- If you index directly into the packed sequence without knowing what you're doing (e.g. without unpacking), you will be confused by the bad performance of your network.

4.1.6 Use a custom wrapper around pack padded sequence and pad packed sequence

Idea: Enough with sorting! I don't want to deal with sorting or argsort or inverse permutations or any of that nonsense. My LSTM's make up 90% of my running time so I don't care if I just use a for loop for all the non-LSTM stuff. Just let me pack and unpack anything I want.

Implementation: Your custom pack function sorts by sequence length, packs the data then returns a tuple (packed data, permutation used for sort). Your custom unpack function will take those two values, unpacks the data and then undoes the permutation used for sorting. Therefore, custom unpack(custom pack(data, sequence lengths)) = (data unchanged, sequence lengths unchanged).

Pros:

- Make these functions work correctly and no more sorting related bugs.
- LSTM's will still run very quickly.
- Can handle multiple sequences of unrelated lengths much better. All the order-related bookmarking is done for you.

Cons:

- Can't use the sorted data optimization manually anywhere else. You're stuck with either masks or for-loops.
- Slightly slowed than the built-in pack and unpack, because of all the shuffling that happens.

4.2 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.

In terms of inputs to your network, this is another difference between hw4 and hw4p1. 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 make them both the same number, like 0, to make things easier.

If the utterance is "hello", then:

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

4.3 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).

4.4 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.

4.5 Teacher Forcing

One problem you will encounter in this setting is the difference of training time and evaluation time: at test time you pass in the generated character/word from your model, when your network is used to having perfect labels passed in during training. One way to help your network be better at accounting for this noise

is to actually pass in your generated chars/words during training, rather than the true chars/words, with some probability. This is known as teacher forcing.

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

4.6 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.

4.7 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.

Now you have at least three options:

- Use the Boolean mask to index just 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 reduce=False), 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 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?

4.8 Inference - Random Search

The easiest thing for decoding would be to just perform a 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.

4.9 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.

4.10 Word based - Implementation

Split every utterance on spaces and create a list of every unique word. Build your arrays on integer values for training. For Kaggle submission, just join your outputs using spaces.

4.11 Good initialization

As you may have noticed in hw3p1, 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 to be a language model before starting the real training. During that pre-training, you can just set the attended context to some random value.

4.12 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.

5 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.

5.1 Some numbers

Here are figures that may answer some of your questions. They are valid for character-based models.

- Your initial perplexity should start around 30
- A language model not using the audio (aka what you should get immediately after pre-training), should be able to get your perplexity at least below 4

- You may have trouble crossing some threshold around perplexity 18 in your training. This corresponds to simple predictions, like only the space character. If that happens, persists several epochs, or change optimizer/learning rate/other optimization parameters.
- The perplexity corresponding to good outputs should be below 1.3.
- The LAS model with the tricks mentioned, here and in the paper, is enough to get to any of these cutoffs. Adding more stuff to the model (like CNN layers in the listener, or ensemble different models) may help you go even further.

6 Getting Started

You are not given any template code for this homework. So, it is important that you design the code well by keeping it as simple as possible to help with debugging. 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 contrary to the previous homeworks, here all models are allowed. We impose that you use attention, but it is pretty much necessary to get good results anyway, so that's not really a constraint. 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, that's perfectly fine to use as long as you implement it yourself. No limits, except that you can't add extra data from different datasets.

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!