(520|600).666
Information Extraction from Speech and Text

Project # 3

Due May 13, 2024.

The goal of this project, again, is to build an isolated-word speech recognizer for a vocabulary of 48 words using the CTC-Objective function.

1 Warmup

1. Prove that the posterior distribution modeled by CTC is globally normalized, i.e., the probability of a sequence is computed as the score of the sequence normalized by the sum of scores over all possible sequences and can be expressed as

$$p\left(y_1^M|\mathbf{x}_1^T\right) = \frac{p\left(y_1^M\right) e^{f\left(\mathbf{x}_1^T, y_1^M\right)}}{\mathbb{E}_{p(y')}\left[e^{f\left(\mathbf{x}_1^T, y'\right)}\right]}. \tag{1}$$

   Make sure to state any assumptions used.

2. Using the result of problem above, show that maximizing the CTC objective maximizes a lower bound on the mutual information, $I\left(X;Y\right)$, between input and output sequences.

2 CTC Training

Previously, you had built a recognizer to process speech represented via *quantized* spectral features, using hidden Markov models (HMMs), one HMM to represent each letter of the alphabet. Word HMMs were composed by concatenating the letter-HMMs of their spelling.

This time, you will build two (recurrent) neural network based recognizers trained using the CTC objective function. One is based on the same quantized speech representations from project #2, and another using continuous-valued Mel-frequency cepstral coefficients (MFCCs).

- Recall that each speech sample (i.e. `*.wav` file) has been processed to extract spectral features every 10 ms, and the resulting feature-vectors have each been quantized to one of 256 discrete labels.

These will constitute your discrete-valued observation sequences $\mathbf{Y}$.

- You will first construct a neural network whose input is $\mathbf{Y}$ and output is the letter sequence $\ell_1\,\ell_2, \ldots, \ell_{|\mathbf{W}|}$ corresponding to the spelling of its word label $\mathbf{W}$.
- You will then train your neural network using all the training data and CTC loss, which maximizes the total posterior probability (given $\mathbf{Y}$) of all time-alignments of the sequence $\ell_1\,\ell_2\,\ldots\,\ell_{|\mathbf{W}|}$ to $\mathbf{Y}$.
- You will then compute the posterior probability (or CTC loss) for each of the 48 words given each test speech sample using your neural network, and choose the maximizer (resp. minimizer) as your output.
- Finally, you will calculate the system's average recognition accuracy over the test data.

The instructions below are written assuming that you will implement your recognizer(s) in `PyTorch`, using the code-skeleton provided by the TAs.

Training and Test Data Files:

There is a file enumerating the possible values of the 256 *cluster labels*, i.e. the (quantized) outputs of the acoustic processor.

`clsp.lblnames` contains 256 two-character-long label names, one per line, resulting in 256 lines, plus a title-line at the top of the file. [Total: 257 lines]

There are four training-data files as described below.

`clsp.trnscr` is the "script" that was read by the speakers whose speech comprises the training data. Each of the 48 words in the vocabulary were presented to speakers in some randomized order. The script-file contains at least 10 and up to 25 examples of each word, for a total of 798 lines of data, plus a title-line at the top of the file. [Total: 799 lines.]

`clsp.trnwav` contains the name of the speech (waveform) file corresponding to the utterance of each word in the script file described above. There are 798 lines, plus a title-line at the top of this file as well. [Total: 799 lines.]

You may want to further separate the training data into a "training" and "validation" set for neural network training.

`clsp.trnlbls` contains $\mathbf{Y}$, the processed speech corresponding to the utterance of each word in the script file described above. There is one long label-string per line, and there are 798 lines, plus a title-line at the top of this file as well. [Total: 799 lines, 106,785 labels.]

`clsp.endpts` contains "end-point" information, or the information about the leading- and trailing-silence surrounding each utterance. This information is encoded in the form of two integers per line, say, i and j, to indicate that the *last label of the leading silence*

is at position i, the *first label of the trailing silence* is at position j, and the speech corresponds to the $(i+1)$-th through $(j-1)$-th labels in the label-file. There are, again, 798 lines of data, plus a title-line. [Total: 799 lines.]

Finally, there are two test-data files:

`clsp.devwav` contains the name of the speech (waveform) file corresponding to the utterance of each word in the test set. There are 393 lines, plus a title-line at the top of this file as well. [Total: 394 lines.]

`clsp.devlbls` contains the labels $\mathbf{Y}$, one variable-length label-string per line, corresponding to an utterance of each word in the test set. There are 393 lines, plus one title-line at the top of this file as well. [Total: 394 lines, 52,812 labels.]

Proceed to build your *primary* recognizer as follows. A code-skeleton is provided.

1. **Create a `Pytorch` DataLoader**: Modify the code provided in the file `dataset.py` to load the dataset. Use the spelling of each word as the output label-sequence for its speech. Pad the spelling on each side with a "silence" symbol of your choice to let the network output a suitable label for silence frames.

2. **Create a `PyTorch` model**: Modify the code provided in the file `model.py` to create a `PyTorch` model of your choosing. An LSTM model is suggested, but you may experiment with CNNs or combinations of CNNs and LSTMs. The outputs of your model should be of dimensions `Batch` $\times$ `InputLength` $\times$ `NumLetters`.

3. **Train the `PyTorch` model**: Train your `PyTorch` model with the CTC criterion. You may wish to compute the CTC loss on some held-out (i.e. validation) set during training to ensure that the model does not overfit.

4. **Build a word recognizer**: The output of your trained system should give you the probabilities of each letter for every frame. Create a recognizer that uses these probabilities to generate the most likely word from the 48-word vocabulary.

   (a) A simple ("greedy") method is to take the most likely letter for every frame as the output symbol, and then "compress" repeated symbols to spell the output word.

   (b) A more complex method is to compute the built-in `CTCLoss` for each silence-padded word hypothesis for every test example, and choose the word with the lowest loss as the output for that example.

5. **Test the `PyTorch` model**: Check the accuracy of your model on the training data itself to ensure that it is training properly. In addition, submit your word hypothesis for each of the 393 test utterances provided.

Submit the following in your report for the *primary* system for this project.

$\Rightarrow$ A plot of the neural network training loss as a function of the number of iterations.

If you held out a validation set to check the progress of your neural network training, you may optionally include its loss in this plot.

⇒ For each utterance in the test data, the identity of the most likely word and a *confidence*.

⇒ Your source code, along with substantial documentation about exactly what files (among the 7 data files provided for the project) are needed to run each module, and the command line (usage) for running the training and testing modules.

- Your code should expect the 7 files to be in the current directory, and should run on a `x86_64` machine running a recent version of GNU/Linux and `PyTorch`.

A *tarball* containing all the above, with obvious filenames and a README are expected.

Build a *contrastive* system to explore alternatives to your primary system.

(i) Use MFCC vectors, or another continuous representation of speech, such as features extracted from the Wav2Vec2.0 model, instead of the 256 discrete features provided. You may use code based on in-built Python libraries to compute these features, as provided by the TAs. HMM-based systems typically compute 40-dimensional MFCCs with an analysis window of 25ms and a stride of 10ms (i.e. 15ms overlap between adjacent windows). However, it may help to tune these hyper-parameters to see what works best.

Submit an analogous report for your *contrastive* system (see items marked ⇒ above).