CS4420 Programming Project 2 (Points: 30, Due: Oct 9)

Weiran Wang
University of Iowa
weiran-wang@uiowa.edu

Abstract

In this project, you will implement the forced alignment algorithm for an ASR model called Connectionist Temporal Classification (CTC). This project can be done with less than 200 lines of python code.

1 Alignment

From the previous project, you have already observed that the sequence length of input is (much) larger than that of the output for speech recognition: the acoustic feature vectors are computed every 10ms, yielding 100 frames per second, whereas the average speech speaking rate is 2.5 words (less than 20 characters) per second. In this course, the tokenizer we use (as implemented in Project 1) is based on characters. In the following, we use the more general term "*tokens*" to refer to the modeling units of an ASR system. In typical ASR systems, the token set consists of not only single characters but also frequent combinations of characters (aka, word pieces). We denote the token set by $\mathcal{V}$.

Let the acoustic feature sequence be $X = [x_1, x_2, \ldots, x_T]$, where $x_t \in R^d$, and the output token sequence be $Y = [y_1, \ldots, y_L]$ where $y_i \in \mathcal{V}$. For ASR, we typically have $L < T$. The most prominent problem for ASR, from a machine learning perspective, is to resolve this length mismatch. We will be reasoning about which token is emitted at each frame. Each token may be emitted at multiple consecutive frames; this is natural for large units that take multiple frame's duration to utter. The sequence of emitted tokens for all the frames is called an *alignment*.

Different ASR models use alignments of different topology. In this course, we will be implementing a specific model called Connectionist Temporal Classification (CTC, [1]), whose alignments have the following properties.

- CTC introduces an additional special token ϕ called *blank*, to indicate non-emission of (real) tokens at a frame. Note ϕ is not the same token you use for space ('_') which denotes the word boundary. Usually we reserve the token id 0 for ϕ.
- For a input acoustic feature sequence X of length T, a CTC alignment $A = [a_1, a_2, \ldots, a_T]$ is a sequence of T tokens, where $a_t \in \mathcal{V}$ or $a_t = \phi$. The token id representation of the alignment is a sequence of T integers, each from $\{0, 1, \ldots, |\mathcal{V}|\}$.
- As mentioned above, we may have repetitions of non-blank tokens in consecutive frames. And we now have ϕ to indicate non-emission. We can then define the *collapse* operation $\mathcal{B}(\cdot)$ which maps an alignment A into a token sequence $\mathcal{B}(A)$, by *first removing the repetitions and then removing the blanks.* As an example,

$$\mathcal{B}([h, h, \phi, e, \phi, \phi, l, l, \phi, l, o]) = hello. \tag{1}$$

- Note a special case: if two consecutive output tokens of Y are the same, a valid alignment for Y must have at least one blank in between. As already demonstrated in the above alignment, we have two consecutive l's in the output sequence *hello*, and there is one ϕ that separate them in the alignment. Had the ϕ been gone, we would obtain a different output sequence:

$$\mathcal{B}([h, h, \phi, e, \phi, \phi, l, l, l, l, o]) = helo. \tag{2}$$

- There exist multiple alignments that can be collapsed into the same output token sequence. For example,

$$\mathcal{B}([h, \phi, \phi, e, \phi, \phi, l, \phi, l, o, \phi]) = hello \tag{3}$$

provides a valid alignment for *hello* different from the one in (1). For a output sequence Y, we define $\mathcal{B}^{-1}(Y)$ to be the set of all valid alignments that collapse into Y:

$$\mathcal{B}^{-1}(Y) = \{A = [a_1, \ldots, a_T] : a_t \in \mathcal{V} \cup \{\phi\},\ \mathcal{B}(A) = Y\}.$$

2 Objective function of CTC

The machine learning component of CTC is a neural network that predicts the probability of emitting each token (including blank) at each frame. Let the output distribution be $P_t(\cdot|X)$ at frame t; the distribution is conditioned on the acoustic features. For an alignment, the product of probabilities of emitting the corresponding token at each frame is the probability of the alignment. That is, for an alignment $A = [a_1, \ldots, a_T]$, we have

$$P(A|X) = \prod_{t=1}^{T} P_t(a_t|X), \tag{4}$$

where $P_t(a_t|X)$ is the neural network's output probability for token a_t at frame t. The probability of the output sequence is then computed as the *marginalization* of alignment probabilities:

$$P(Y|X) = \sum_{A \in \mathcal{B}^{-1}(Y)} P(A|X). \tag{5}$$

The standard training method for CTC is to adapt the neural network weights to maximize $P(Y|X)$. This method is called *Baum-Welch* training.

In this project, however, we will use a different training strategy. We will make the assumption that the probability mass of $P(Y|X)$ concentrates on one alignment A^*:

$$P(Y|X) \approx P(A^*|X), \qquad \text{where} \quad A^* = \operatorname*{argmax}_{A \in \mathcal{B}^{-1}(Y)} P(A|X). \tag{6}$$

And once we have found A^*, we can use the tokens at each frame as targets for training the neural network, and we turn ASR into a standard classification problem! This method is called *Viterbi* training.[1]

The task of this project is to find the optimal alignment A^* for a given output sequence Y.

3 Finding the optimal alignment

The task is, in its essence, a search problem. The optimal alignment A^* is known as the *forced alignment* in the ASR literature.

To allow for blanks in the alignments for Y, we consider a modified output sequence Y', with blanks added to the beginning and the end and inserted between every pair of labels. The length of Y' is therefore $2L+1$. In Figure 1, we illustrate the search graph for an utterance of T frames, whose transcript is $Y = [C, A, T]$. The rows of the graph corresponds to $Y' = [\phi, C, \phi, A, \phi, T, \phi]$. We start at the top-left corner of the graph, from either of the first two nodes[2], and travel along the arrows which indicate allowed transitions, and end with either of the two bottom right nodes. Every path from one of the start nodes to one of the end nodes, i.e., a solution in the AI course terminology, is a valid CTC alignment.

Consistent with the properties of CTC alignments, the following transitions are allowed when we move from one frame to the next:

[1] The terms *Baum-Welch* and *Viterbi* originate from the HMM literature. In fact, CTC can be considered as a special form of HMM, see [2].

[2] We could also have a imaginary start node at $t = 0$ with transitions to the first two nodes at $t = 1$.

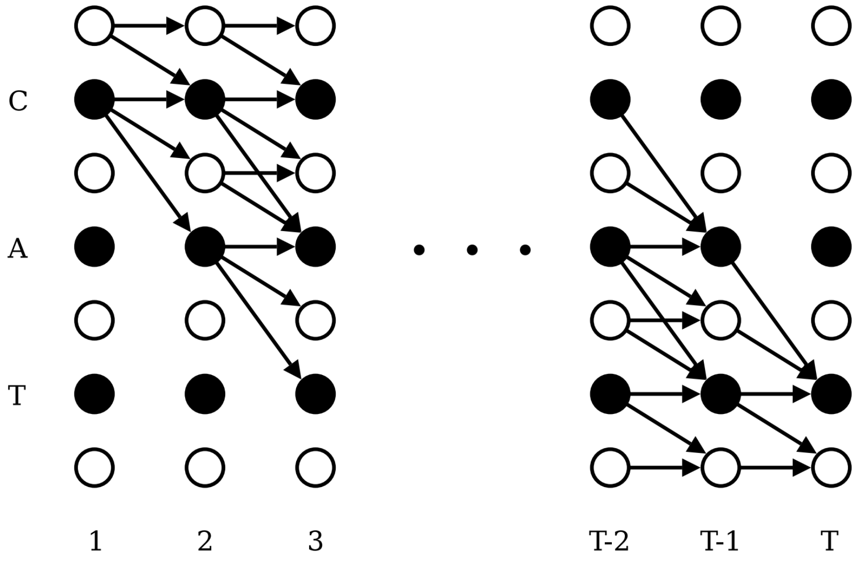

Figure 1: The CTC search graph. Black nodes represent non-blank tokens, and white nodes represent blanks.

- If we were on a blank row, we can move either horizontally to the same blank row, or diagonally to the next non-blank row.
- If we were on a non-blank row, we can move horizontally to the same non-blank row, or diagonally to the next blank row, or diagonally to the next non-blank row if the two non-blank tokens are different.

Collecting the tokens on the solution path gives the alignment. From the figure, we can verify that every alignment is collapsed into Y after removing repetitions (basically horizontal moves) and blanks,.

For an alignment $A = [a_1, \ldots, a_T]$, there is a cost associated with the transition to the node at time t with token a_t, which is the negative log-probability $-\log P_t(a_t|X)$.[3] As a result, the cost of an alignment (as a path in the search graph) is

$$cost(A) = -\sum_{t=1}^{T} \log P_t(a_t|X) = -\log\left(\prod_{t=1}^{T} P_t(a_t|X)\right) = -\log P(A|X). \tag{7}$$

The advantage of working in the log space is that it is numerically more stable: multiplication in the probability space[4] reduces to addition in the log-probability space.

Now, in view of (7), the problem reduces to finding the solution in the search graph with lowest cost, a problem we know how to solve! Using terminologies from the AI course, the specific search algorithm we use here is breadth-first graph-search, where "breadth-first" means we systematically expand nodes first at $t = 1$, then $t = 2$, and so on, while "graph-search" means we maintain the best path for reaching each state along the way.

We now define the dynamic programming procedure to find A^* for $Y' = [\phi, y_1, \phi, \ldots, \phi, y_L, \phi]$. We use the shorthand $S_{t,y} = -\log P_t(y|X)$, for $t = 1, \ldots, T$, $y \in \mathcal{V} \cup \{\phi\}$. Define $\alpha_t(l)$, $l = 1, \ldots, 2L+1$ as the cost of the best path from one of the initial nodes to row l at time t of the search graph. At $t = 1$, we have

$$\begin{aligned} \alpha_1(1) &= S_{1,\phi} && (Y'[1] = \phi) \\ \alpha_1(2) &= S_{1,y_1} && (Y'[2] = y_1) \\ \alpha_1(l) &= \infty, && l > 2. \end{aligned}$$

[3] Typically we want to maximize the likelihood, and therefore minimize the negative log-likelihood.

[4] We are talking about multiplying together many values less than 1, and the result quickly diminishes to 0.

The recursion from $t-1$ to t is computed as

$$\begin{aligned}
&\alpha_t(l) = \min\{\alpha_{t-1}(l),\ \alpha_{t-1}(l-1)\} + s_{t,Y'[l]}, && \text{if } Y'[l] = \phi \text{ or } Y'[l-2] = Y'[l],\\
&\alpha_t(l) = \min\{\alpha_{t-1}(l),\ \alpha_{t-1}(l-1),\ \alpha_{t-1}(l-2)\} + s_{t,Y'[l]}, && \text{otherwise.}
\end{aligned}$$

At $t = T$, the best cost is computed as

$$cost(A^*) = \min\{\alpha_T(2L),\ \alpha_T(2L+1)\}.$$

By storing the argmins during dynamic programming, we can perform backtracking to find the tokens of A^*.

4 Submission

You should implement the function `compute_forced_alignment()` in the file `ctc.py` and submit that file. Test your implementation by running `ctc_test.py` and passing the assertions without modifying the file.

As for the interface of the function, `compute_forced_alignment()` takes the following inputs.

- A float array S of the shape $[B, T, C]$. It contains the negative of log-probabilities output by a neural network (we are faking it in this project by providing random values) for a batch of B utterances. That is, the slice $S[b, :, :]$ contains the log-probabilities for utterance b. Note T is the maximum length of the acoustic features. If an utterance has less than T frames, we pad the log-probabilities to have length T; it does not matter what values we use for padding. $C = 1 + |\mathcal{V}|$ is the output dimension of the neural network, each integer in $[0, 1, \ldots, |\mathcal{V}|]$ indexes a token and 0 is reserved for blank.
- A integer array *input_lengths* of the shape $[B]$. It contains the actual lengths of the utterances. The actual length determines the depth of the search for each utterance.
- A integer array Y of the shape $[B, L]$. It contains the output token sequences of the utterances. Again L is maximum output length and some token sequences may contain paddings.
- A integer array *output_lengths* of the shape $[B]$. It contains the actual lengths of the outputs.

On the other hand, `compute_forced_alignment()` returns the following:

- A float array *best_costs* of the shape $[B]$. It contains the cost of optimal alignments for each utterance.
- A integer array *paths* of the shape $[B, T]$. It contains the optimal alignments for each utterance.

Padded positions of the outputs are set to 0.

References

[1] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *International Conference on Machine Learning*, 2006.

[2] A. Zeyer, E. Beck, R. Schlüter, and H. Ney. CTC in the context of generalized full-sum HMM training. In *Interspeech*, 2017.