GPU-Accelerated Forward-Backward Algorithm with Application to LF-MMI Training

Lucas Ondel*, Léa-Marie Lam-Yee-Mui*, **Martin Kocour**†, Caio Filippo Corro*, Lukáš Burget†

*LISN (ex-LIMSI), CNRS, Orsay, France
†Brno University of Technology, Brno, Czech Republic

May 22, 2022

Overview

1. Implementation challenge of the MMI loss
2. A new implementation of the Forward-Backward algorithm
3. Results

Implementation challenge of the MMI loss

The MMI loss

- SoTA for ASR (Vyas et. al)
- MMI stands for Maximum Mutual Information

$$\mathcal{L} = \log \frac{p(\mathbf{X}|\mathbb{G}_{\text{num}})}{p(\mathbf{X}|\mathbb{G}_{\text{den}})}, \tag{1}$$

- "Lattice-Free" MMI (LF-MMI) uses phone rather than words to keep a small memory footprint bypassing the need to build lattices
- Its gradient is given by:

$$\nabla\mathcal{L} = p(z_n = i|\mathbf{X}, \mathbb{G}_{\text{num}}) - p(z_n = i|\mathbf{X}, \mathbb{G}_{\text{den}}). \tag{2}$$

where z_n is the state index at time n.

[A. Vyas, S. Madikeri, H. Bourlard, *Comparing CTC and LFMMI for out-of-domain adaptation of wav2vec 2.0 acoustic model.* InterSpeech 2021]

Forward-Backward algorithm

- Algorithm to compute the probability to be in a given state at time n.

$$p(z_n = i | x_1, \ldots, x_N) \tag{3}$$

- Usage:
  - Expectation-Maximization for HMM (E-step)
  - Calculating the gradient of the losses: CTC, (LF-)MMI

Description

- Algorithm comprises of computing forward $\alpha_n(z_n)$ and backward $\beta_n(z_n)$ messages

$$\alpha_n(z_n) = p(x_n|z_n) \sum_{z_n} p(z_n|z_{n-1}) \alpha_{n-1}(z_{n-1}) \tag{4}$$

$$\beta_n(z_n) = \sum_{z_{n+1}} p(z_{n+1}|z_n) \beta_{n+1}(z_{n-1}) p(x_{n+1}|z_{n+1}) \tag{5}$$

$$p(z_n = i|\mathbf{x}) = \frac{\alpha_n(i)\beta_n(i)}{\sum_j \alpha_N(j)} \tag{6}$$

Implementation challenges

Implementation is not trivial:

- state-space is big (at the very least several toushand)
- numerically unstable, we need to do the computations in the log-domain
- transition probabilities are sparse
- it needs to run on GPU

A new implementation of the Forward-Backward algorithm

The notation

$$\mathbf{T} = \begin{bmatrix} p(z_n = 1|z_{n-1} = 1) & \dots & p(z_{n-1} = K|z_{n-1} = 1) \\ \vdots & \ddots & \vdots \\ p(z_n = 1|z_{n-1} = K) & \dots & p(z_n = K|z_{n-1} = K) \end{bmatrix}$$

$$\mathbf{v}_n = \begin{bmatrix} p(x_n|z_n = 1) \\ \vdots \\ p(x_n|z_n = K) \end{bmatrix} \boldsymbol{\alpha}_n = \begin{bmatrix} \alpha_n(\overbrace{1}^{z_n}) \\ \vdots \\ \alpha_n(K) \end{bmatrix} \boldsymbol{\beta}_n = \begin{bmatrix} \beta_n(\overbrace{1}^{z_n}) \\ \vdots \\ \beta_n(K) \end{bmatrix},$$

Naive solution

Matrix-based forward-backward:

$$\boldsymbol{\alpha}_n = \mathbf{v}_n \circ (\mathbf{T}^\top \boldsymbol{\alpha}_{n-1}) \tag{7}$$

$$\boldsymbol{\beta}_n = \mathbf{T}(\boldsymbol{\beta}_{n+1} \circ \mathbf{v}_n) \tag{8}$$

$$p(z_n|\mathbf{x}) = \frac{\alpha_n(z_n)\beta_n(z_n)}{\sum_{z_N} \alpha_N(z_N)}. \tag{9}$$

- efficient for computation on GPU
- $\mathbf{T}$ has to be sparse, i.e. 0 values are not stored
- stability issue: product of probabilities goes to zero exponentially

An equivalent version

A more generic version:

$$\boldsymbol{\alpha}_n^{\log} = \mathbf{v}_n^{\log} \circ (\mathbf{T}^{\log\top} \boldsymbol{\alpha}_{n-1}^{\log}) \tag{10}$$

$$\boldsymbol{\beta}_n^{\log} = \mathbf{T}^{\log}(\boldsymbol{\beta}_{n+1}^{\log} \circ \mathbf{v}_n^{\log}) \tag{11}$$

$$\log p(z_n|\mathbf{x}) = \frac{\alpha_n^{\log}(z_n)\beta_n^{\log}(z_n)}{\sum_{z_N} \alpha_N^{\log}(z_N)}. \tag{12}$$

where $x^{\log} = \log(x)$ and $x^{\log} \in \mathcal{S}$
and $\mathcal{S} = (\mathbb{R}, \oplus, \otimes, \oslash, \bar{0}, \bar{1})$ is the log-semiring:

$$a^{\log} \oplus b^{\log} = \log(e^{a^{\log}} + e^{b^{\log}}) \tag{13}$$

$$a^{\log} \otimes b^{\log} = a^{\log} + b^{\log} \tag{14}$$

$$a^{\log} \oslash b^{\log} = a^{\log} - b^{\log} \tag{15}$$

$$\bar{0} = -\infty \quad \bar{1} = 0 \tag{16}$$

Sparse matrix in semiring alegbra

- stability issue solved: calculations are numerically stable in the log-semiring
- GPU efficiency: $\mathbf{T}^{\log}$ should be represented as a sparse matrix which stores only the element that are not $\bar{0}$, i.e. not $-\infty$.

Dealing with batches

- when using GPU, we need to train on batches of sequence to fully exploit the hardware
- hopefully, this is very easy when our algorithm is expressed in terms of linear matrix operation...
- ... we just need to stack everything together!

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_1 & & \\ & \ddots & \\ & & \mathbf{T}_I \end{bmatrix} \qquad \mathbf{v}_n = \begin{bmatrix} \mathbf{v}_{1,n} \\ \vdots \\ \mathbf{v}_{I,n} \end{bmatrix}$$

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{1,n} \\ \vdots \\ \boldsymbol{\alpha}_{I,n} \end{bmatrix} \qquad \boldsymbol{\beta}_n = \begin{bmatrix} \boldsymbol{\beta}_{1,n} \\ \vdots \\ \boldsymbol{\beta}_{I,n} \end{bmatrix},$$

In practice

Implementation of this algorithm is trivial if:

- you have (sparse) semiring algebra library
- we have used the Julia programming language:
  - native support for semiring-algebra
  - can be used to write CUDA kernels
  - rich deep learning ecosystem
  - `https://github.com/lucasondel/MarkovModels.jl/blob/master/src/inference/algorithms.jl`

Results

State-of-the-art implementation

PyChain (`https://github.com/YiwenShaoStephen/pychain`):

- C++/CUDA implementation
- wrapped with python/pytorch for training with a neural network
- optional use of the "leaky-HMM" approximation for efficiency reason

Experiment

Comparing PyChain and our Julia implementation:

- datasets:
  - MiniLibriSpeech: subset of librispeech for software testing (5 hours)
  - Wall Street Journal (WSJ) (80 hours)
- features: high-resolution MFCC extracted with Kaldi
- model: 5-layers TD-NN (equivalent to the "Chain" model in Kaldi)
- 4-gram Kaldi decoder
- neural network backends:
  - PyChain: PyTorch
  - ours: KNet

Results I

System	Dataset	B/F	Duration	WER (%)
PyChain	MiniLS	128/1	0h42	27.17
proposed	MiniLS	64/2	**0h22**	**21.21**
PyChain	WSJ	128/1	6h48	4.74
proposed	WSJ	64/2	**3h20**	**4.37**

B is the batch-size and F is the update frequency.

Results II

System	Avg. duration (s) LF-MMI	Avg. duration (s) Neural network propagation
PyChain	4.08	0.076
proposed	0.220	0.180

To sum up

- Julia offers new abstractions to numerical computation:
  - easy to use semiring algebra
  - no need to mix languages (python/c++), direct optimization in Julia

To sum up

- Julia offers new abstractions to numerical computation:
  - easy to use semiring algebra
  - no need to mix languages (python/c++), direct optimization in Julia
- we proposed new matrix-based implementation of forward-backward
  - we make it simpler
  - we do it faster
- Reproducing the results:
  `https://lucasondel.github.io/recipes`

Thank you for your attention :) !

Comparing the implementations

Forward-Backward calculated on:

- the longest numerator graph and the denominator graph of WSJ
- simulated likelihood of 128 sequences of 14 seconds of speech

			Duration (s)	
Implementation	**Device**	**Leaky-HMM**	**Num.**	**Den.**
PyChain	CPU	no	5.696	421.447
PyChain	CPU	yes	**1.212**	**27.601**
proposed	CPU	no	3.789	226.60
PyChain	GPU	no	0.093	5.862
PyChain	GPU	yes	0.248	5.449
proposed	GPU	no	**0.058**	**1.04**