GPU-ACCELERATED FORWARD-BACKWARD ALGORITHM WITH APPLICATION TO LATTICE-FREE MMI

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

* LISN, CNRS, Université Paris-Saclay, Orsay, France
† Brno University of Technology, Faculty of Information Technology, Brno, Czech Republic
‡ Vocapia Research, Orsay, France
ondel@lisn.fr

ABSTRACT

We propose to express the forward-backward algorithm in terms of operations between sparse matrices in a specific semiring. This new perspective naturally leads to a GPU-friendly algorithm which is easy to implement in Julia or any programming languages with native support of semiring algebra. We use this new implementation to train a TDNN with the LF-MMI objective function and we compare the training time of our system with PyChain—a recently introduced C++/CUDA implementation of the LF-MMI loss. Our implementation is about two times faster while not having to use any approximation such as the "leaky-HMM".

Index Terms— Lattice-Free MMI, end-to-end ASR, Julia language, forward-backward

1. INTRODUCTION

The forward-backward algorithm is a crucial algorithm in speech recognition. It is used to compute the posterior distribution of state occupancy in the Expectation-Maximization training approach for Hidden Markov Model (HMM). Even though deep learning approaches have superseded the traditional GMM/HMM-based ASR [1, 2], the forward-backward algorithm is still used to estimate the gradient of two major sequence discriminative objective functions: Connectionist Temporal Classification [3] and Lattice-Free MMMI (LF-MMI) [4, 5].

Because state-of-the-art models are trained on GPU, having a fast and an efficient implementation of these losses (and their gradients) is essential. The implementation is usually done in C++ and then wrapped in a Python module to be integrated with popular neural network libraries [6, 7]. However, this practice is far to be satisfactory as the C++ code is usually complex, difficult to modify and, as we shall see, not necessarily optimal.

In this work, we propose a different approach: we express the forward-backward algorithms in terms of operations between matrices living in a particular semiring. This new perspective leads to a trivial implementation in the Julia language [8] which is significantly faster than a competitive C++ implementation. As a result, our proposed implementation[1] of the forward-backward is just few lines long, easy to read, and easy to extend by anyone.

This paper is organized in two parts: in Section 2, we describe the forward-backward algorithm and its representation in terms of semiring algebra and in Section 3, we conduct our numerical analysis.

Finally, we warmly encourage interested readers to look at the provided code and the Pluto notebooks[2]; we have made them with the hope to be accessible by a vast majority.

2. ALGORITHM

2.1. Description

Let $\mathbf{x} = (x_1, x_2, \dots, x_N)$ be a sequence of features and $\mathbf{z} = (z_1, z_2, \dots, z_n)$ an unobserved sequence sampled from a discrete-time Markov process defined by the transition probability $p(z_n|z_{n-1})$. Each z_n takes value in $\{1, \dots, K\}$, where $1, \dots, K$ are the index of the states of the Markov process. The forward-backward algorithm [9] calculates the marginal state posterior distribution:

$$p(z_n|\mathbf{x}) = \frac{\alpha_n(z_n)\beta_n(z_n)}{\sum_{z_N} \alpha(z_N)}, \tag{1}$$

where α (the forward pass) and β (the backward pass) are recursive functions defined as:

$$\alpha_n(z_n) = p(x_n|z_n) \sum_{z_n} p(z_n|z_{n-1}) \alpha_{n-1}(z_{n-1}) \tag{2}$$

$$\beta_n(z_n) = \sum_{z_{n+1}} p(x_{n+1}|z_{n+1}) p(z_{n+1}|z_n) \beta_{n+1}(z_{n+1}). \tag{3}$$

Whereas the algorithm is simple to state, its implementation can be quite challenging. First, because it involves multiplication of probabilities, a naive implementation would quickly underflow. Second, the state-space is usually very large leading to heavy computations. However, because it is frequent that $p(z_n|z_{n-1})$ is zero for most of the pairs z_n, z_{n-1} the amount of "useful" operations remains relatively low.

Thus, an efficient implementation of the forward-backward algorithm should address these two issues: numerical stability and using the structure of the transition probabilties to gain speed.

2.2. Matrix-based implementation

A convenient way to implement the forward-backward algorithm is to express (2) and (3) in terms of matrix operations. We introduce

[1] `https://github.com/lucasondel/MarkovModels.jl`, the forward-backward is implemented in the functions α`recursion` and β`recursion`

[2] `https://github.com/lucasondel/SpeechLab/tree/main/recipes/lfmmi`

978-1-6654-0540-9/22/$31.00 ©2022 IEEE

the following notation:

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

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

and we rewrite the forward-backward algorithm as:

$$\boldsymbol{\alpha}_n = \mathbf{v}_n \circ (\mathbf{T}^\top \boldsymbol{\alpha}_{n-1}) \tag{4}$$

$$\boldsymbol{\beta}_n = \mathbf{T}(\boldsymbol{\beta}_{n+1} \circ \mathbf{v}_n), \tag{5}$$

where $\circ$ is the Hadamard (i.e. element-wise) product. Implementing (4) and (5) is trivial using any linear algebra library. Another advantage of this implementation is that it can be easily accelerated with a GPU as matrix multiplication is a highly optimized operation on such device. Finally, we can represent the matrix $\mathbf{T}$ as a sparse matrix and avoid performing unnecessary operations.

However, despite all these benefits, this implementation remains numerically unstable.

2.3. Semiring algebra

In order to keep the advantages of the matrix-based implementation while solving the numerical stability issue, we propose to express the algorithm in terms of matrices in the log-semifield[3].

For a matrix $\mathbf{M}$ with non-negative entries we define:

$$\mathbf{M}^{\log} = \begin{bmatrix} \log M_{11} & \log M_{12} & \dots \\ \log M_{21} & \ddots & \\ \vdots & & \end{bmatrix}, \tag{6}$$

$$M_{ij}^{\log} \in \mathcal{S}(\mathbb{R}, \oplus, \otimes, \oslash, \bar{0}, \bar{1}), \tag{7}$$

where $\mathcal{S}$ is the log-semifield defined as:

$$a^{\log} \oplus b^{\log} = \log\left(e^{a^{\log}} + e^{b^{\log}}\right) \tag{8}$$

$$a^{\log} \otimes b^{\log} = a^{\log} + b^{\log} \tag{9}$$

$$a^{\log} \oslash b^{\log} = a^{\log} - b^{\log} \tag{10}$$

$$\bar{0} = -\infty \tag{11}$$

$$\bar{1} = 0. \tag{12}$$

Equipped with these new definitions, we express the forward-backward algorithm in the logarithmic domain:

$$\boldsymbol{\alpha}_n^{\log} = \mathbf{v}_n^{\log} \circ (\mathbf{T}^{\log\top} \boldsymbol{\alpha}_{n-1}^{\log}) \tag{13}$$

$$\boldsymbol{\beta}_n^{\log} = \mathbf{T}^{\log}(\boldsymbol{\beta}_{n+1}^{\log} \circ \mathbf{v}_n^{\log}) \tag{14}$$

$$\log p(z_n | \mathbf{x}) = \frac{\alpha_n^{\log}(z_n)\beta_n^{\log}(z_n)}{\sum_{z_N} \alpha_N^{\log}(z_N)}. \tag{15}$$

Altogether, (13), (14) and (15) leads to an implementation of the forward-backward algorithm which is (i) trivial to implement (if provided with a semiring algebra API) as it consists of a few matrix multiplications, (ii) numerically stable as all the computations are in the logarithmic domain, and (iii) efficient as the matrix $\mathbf{T}$ can be represented as a sparse matrix storing only the elements different than $\bar{0}$.

2.4. Dealing with batches

Thus far, we have described our new implementation for one sequence only. However, when training a neural network it is common practice to use a batch of input sequences to benefit from the GPU parallelization. Our matrix representation of the algorithm is easily extended to accommodate for multiple sequences.

In the following, we drop the superscript $^{\log}$ for the sake of clarity. We define $\mathbf{T}_i$ the transition probabilities of the ith sequence of the batch of size I. $\boldsymbol{\alpha}_{i,n}$, $\boldsymbol{\beta}_{i,n}$ and $\mathbf{v}_{i,n}$ are defined similarly. Now, we set:

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

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

and using these new variables in (13), (14) and (15), we obtain a batch version of the forward-backward algorithm. This algorithm naturally extends to batch of sequences of different length by adding to the Markov process a phony self-looping state marking the end of sequence and padding $\mathbf{v}$ with $\bar{0}$ appropriately.

2.5. Using the Julia language

The algorithm we have described so far is straightforward to implement but for one difficulty: packages dealing with sparse matrices usually store the elements that are different from 0. However, for the log-semifield, because $\bar{0} \neq 0$ and $\bar{1} = 0$, this would lead to ignore important values and store many of non-relevant ones.

Implementing in C or C++ a sparse matrix API agnostic to the semiring is not a trifle. It is perhaps easier to do it in a scripting language such as Python, but that would lead to poor performances. Fortunately, the Julia language provides an elegant solution to this problem.

Julia is a high-level language with performances comparable to other statically compiled languages. From a user perspective, it provides a programming experience close to what a Python programmer is accustomed to while allowing to write critical code without resorting to C or C++[4]. Importantly for our problem, Julia naturally allows to use arbitrary semirings [10] and, consequently, implementing a sparse matrix in the log-semifield amounts to write a few lines of code to implement (8)-(12). Moreover, Julia offers a rich landscape for GPU [11] and neural network toolkits [12, 13] allowing to integrate any new code with state-of-the-art machine learning techniques.

3. APPLICATION

We now demonstrate the practicality of our algorithm by using it to build an ASR system trained with the LF-MMI objective function.

[3] A semifield is a semiring for which all the elements but $\bar{0}$ have a multiplicative inverse. Loosely speaking, a semifield is a semiring with a division operator.

[4] It is also possible to write GPU kernels directly in Julia as was done in this work.

3.1. Objective function

The LF-MMI objective function [14, 4, 5] for one utterance is defined as:

$$\mathcal{L} = \log \frac{p(\mathbf{X}|\mathbb{G}_{\text{num}})}{p(\mathbf{X}|\mathbb{G}_{\text{den}})}, \tag{16}$$

where $\mathbb{G}_{\text{num}}$ is the *numerator graph*, i.e. an utterance-specific alignment graph, and $\mathbb{G}_{\text{den}}$ is the *denominator graph*, i.e. a phonotactic language model. If we denote $\boldsymbol{\Phi} = (\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \dots)$ the sequence output by a neural network where we interpret $\phi_{n,i} = \log p(\mathbf{x}_n|z_n = i)$, the derivatives of the loss are given by:

$$\frac{\partial \mathcal{L}}{\partial \phi_{n,i}} = p(z_n = i|\mathbf{X}, \mathbb{G}_{\text{num}}) - p(z_n = i|\mathbf{X}, \mathbb{G}_{\text{den}}). \tag{17}$$

In practice, these derivatives are estimated by running the forward-backward algorithm on the numerator and denominator graphs.

3.2. Python and Julia recipes

The baseline is the PyChain [15] package which implements the LF-MMI objective function integrated with the PyTorch [7] neural network toolkit. [15] is the latest development of the original "Kaldi chain model" and, to the best of our knowledge, it is currently the most competitive implementation of the LF-MMI training. We have used the PyChain recipe[5] provided by the authors to prepare and to train the system.

We compare the PyChain recipe against our Julia-based recipe[6] that is built on top of our implementation of the forward-backward algorithm and the KNet [12] neural network Julia package. Despite that there exists other popular Julia neural network packages, we elected to use KNet as it is technically the most similar to PyTorch.

Note that the PyChain implementation of the LF-MMI loss is not exact: the forward-backward on the denominator is done with the so-called "leaky-HMM" approximation [4] which speeds up the computations at the expense of an approximate result.

3.3. Datasets

We use two datasets:

- MiniLibrispeech, a subset of the Librispeech corpus [16] created for software testing purposes. It contains 5 hours and 2 hours of training and validation data respectively. Because, it is only for "debugging", it doesn't have a proper test set and we report the WER on the validation set.
- the Wall Street Journal (WSJ) corpus [17], where we use the standard subsets for training (si284), validating (dev93) and testing eval92)

3.4. Model and Graphs preparation

Prior training the model with the LF-MMI objective, one needs to prepare the alignments graphs (i.e. the numerator graphs) and the n-gram phonotactic language model (i.e. the denominator graph). Our preparation is identical to the PyChain recipe with one exception: whereas the PyChain recipe uses only one pronunciation for words having multiple ones, we use all the pronunciations when building the numerator graphs. In both recipes, we set the n-gram order of the phonotactic language model to 3.

The model is a Time-Delay Neural Network (TDNN) [18] with 5 convolutional layers and a final affine transformation. Each convolutional layer is composed of the following sequence:

- 1-dimensional convolution
- batch-normalization
- REctified Linear Unit activation
- dropout with a dropping probability of 0.2

For each convolutional layer, the kernel sizes are $(3, 3, 3, 3, 3)$, the strides are $(1, 1, 1, 1, 3)$ and the dilations are $(1, 1, 3, 3, 3)$.

The input features to the model are standard 40-dimensional MFCCs extracted at a 10 ms frame rate. These features are then mean and variance normalized on a per-speaker basis. The neural network output dimension is set to 84—we have 42 phones and each phone is modelled a with 2-state HMM topology [5] resulting 2×42 emission probabilities.

3.5. Training

Our recipe follows exactly the one of PyChain: we use the Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.999$ and an initial learning rate of 10^{-3}. If there is no improvement of the validation loss after one epoch, the learning rate is halved. We also apply a so-called "curriculum" training for one epoch, i.e. we sort in ascending order the utterances by their durations for the first epoch. For the rest of the training, we form batches of sequences of similar lengths and we shuffle the order of the batches. Upon completion of the training, we select the model with the best validation loss to decode the test data.

We have observed that our Julia-based recipe consumes more memory and cannot accommodate the same batch size as the PyChain recipe. To leverage this issue, we divide the batch size B by a factor F and we update the model after accumulating the gradient for F batches. In this way, the gradient calculated is the same but memory requirement is divided by F.

3.6. Results

The results reported here were run with a NVIDIA GeForce RTX 2080 Ti GPU.

The final Word Error Rate (WER) evaluated on the test set and the duration of the neural network training for both recipes are shown in Table 2. We observe that Julia-based training drastically outperforms the baseline recipe in term of training time even though it cannot use the same batch size. Regarding the WER, whereas differences for the WSJ is small, our recipe achieves a much better WER on the MiniLibrispeech corpus. Currently, we do not have a definitive explanation for this improvement. A potential cause that we will explore in future work is that the approximation used in PyChain may degrade the performance of the system when trained on small amount of data.

3.7. Analysis

In order to gain further insights about the training speed up observed in the previous experiment, we compare our proposed implementation of the forward-backward algorithm with the ones found in PyChain. PyChain ships two versions of the forward-backward both implemented in C++: the first one calculates the exact computations in the logarithmic domain whereas the second runs in the probability

[5] https://github.com/YiwenShaoStephen/pychain_example

[6] https://github.com/lucasondel/SpeechLab/tree/main/recipes/lfmmi.

			Duration (s)	
Implementation	**Device**	**Leaky-HMM**	**Numerator**	**Denominator**
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**

Table 1: Comparison between the PyChain and the proposed implementations of the forward-backward algorithm. On GPU, our implementation is significantly faster than both PyChain implementations.

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**

Table 2: Comparison between the PyChain recipe and our Julia-based recipe of LF-MMI. B is the batch size and F is the gradient update frequency. Note that we report the duration of the neural network training, not the total duration of the recipes.

domain but uses a scaling factor and the "leaky-HMM" approximation to overcome numerical stability issues.

We run our benchmark on two different graphs reflecting the typical structure of the numberator and denominator graphs in the LF-MMI function. For the numerator graph, we select the alignment graph from the training set of WSJ with the largest number of states. This alignment graph has 454 states and 1036 arcs. Then, we replicate this graph 128 times to simulate a training mini-batch and we measure the duration of the forward-backward on 128 input sequences of pseudo-loglikelihoods. We set each sequence to have 700 frames as it corresponds to the length of the largest likelihood sequence on WSJ. For the denominator graph, we repeat the same procedure replacing the alignment graph with a 3-gram phonotactic language model. The denominator graph has 3022 states and 50984 arcs.

The duration of the different implementations are shown in Table 1. First looking at the CPU version, we see that the proposed Julia implementation significantly outperfoms the logarithmic-domain PyChain implementation. The "leaky-HMM" version is drastically faster at the cost of yielding an approximate result. On GPU, the benefit of the "leaky-HMM" vanishes and the implementation is even slower in the case of the numerator graph benchmark. On the other hand, our Julia implementation fully benefits from the parallelism provided by the GPU device and shows more than 5 times speed up compared to logarithmic-domain PyChain implementation on the denominator graph.

Next, we measure the overhead induced by the forward-backward algorithm during the training. In Table 3, we plot the average duration of (i) the computation the LF-MMI loss and its gradient as described in (17), (ii) the forward and backward propagation through the neural network computation excluding the loss computation. These averages are estimated from 100 gradient computations during the WSJ training. We see that for the Julia-based version the loss overhead is severely reduced and accounts for only half the time of the total backpropagation. Interestingly, one can see that KNet, the neural network backend used in the Julia recipe, is actually slower than PyTorch, the neural network backend used in the PyChain recipe. Therefore, the speed up of the Julia-based training comes from our forward-backward algorithm rather than a faster neural network toolkit.

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

Table 3: Comparison of the average time spend in (i) the computation of the LF-MMI loss and its gradient, (ii) the forward and backward propagation through the neural network.

4. CONCLUSION AND FUTURE WORK

We have proposed a new implementation of the forward-backward algorithm relying on semiring algebra. This implementation naturally fits in the Julia language and achieves a significant speed up compared to a competitive C++ implementation.

So far, we have only explored the use of the log-semifield, however, it is trivial to extend our algorithm to other semirings. Particulary, replacing the log-semiring with the tropical-semiring would lead to a straightforward implementation of the Viterbi algorithm. It is remarkable that with the advance programming languages, the implementation of something as complex as a full-fledged speech decoder can now be done in a few dozens lines of code.

Finally, we hope that this work will sparkle the interest of the speech community to the rich capabilities offered by the Julia language ecosystem.

5. REFERENCES

[1] Nelson Morgan and Herve Bourlard, "An introduction to hybrid hmm/connectionist continuous speech recognition," *IEEE Signal Processing Magazine*, vol. 12, no. 3, pp. 25–42, 1995.

[2] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig, "Achieving human parity in conversational speech recognition," *arXiv preprint arXiv:1610.05256*, 2016.

[3] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural net-

works," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 369–376.

[4] Daniel Povey, Vijayaditya Peddinti, Daniel Galvez, Pegah Ghahremani, Vimal Manohar, Xingyu Na, Yiming Wang, and Sanjeev Khudanpur, "Purely sequence-trained neural networks for asr based on lattice-free mmi.," in *Interspeech*, 2016, pp. 2751–2755.

[5] Hossein Hadian, Hossein Sameti, Daniel Povey, and Sanjeev Khudanpur, "End-to-end speech recognition using lattice-free mmi.," in *Interspeech*, 2018, pp. 12–16.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[9] Lawrence R Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[10] Viral B Shah, Alan Edelman, Stefan Karpinski, Jeff Bezanson, and Jeremy Kepner, "Novel algebras for advanced analytics in julia," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2013, pp. 1–4.

[11] Tim Besard, Christophe Foket, and Bjorn De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[12] Deniz Yuret, "Knet: beginning deep learning with 100 lines of julia," in *Machine Learning Systems Workshop at NIPS*, 2016, vol. 2016, p. 5.

[13] Mike Innes, "Flux: Elegant machine learning with julia," *Journal of Open Source Software*, vol. 3, no. 25, pp. 602, 2018.

[14] Y-L Chow, "Maximum mutual information estimation of hmm parameters for continuous speech recognition using the n-best algorithm," in *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 1990, pp. 701–704.

[15] Yiwen Shao, Yiming Wang, Daniel Povey, and Sanjeev Khudanpur, "Pychain: A fully parallelized pytorch implementation of lf-mmi for end-to-end asr," *arXiv preprint arXiv:2005.09824*, 2020.

[16] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2015, pp. 5206–5210.

[17] Douglas B Paul and Janet Baker, "The design for the wall street journal-based csr corpus," in *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.

[18] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur, "A time delay neural network architecture for efficient modeling of long temporal contexts," in *Sixteenth annual conference of the international speech communication association*, 2015.