# A PARALLELIZABLE LATTICE RESCORING STRATEGY WITH NEURAL LANGUAGE MODELS

*Ke Li*[1]*, Daniel Povey*[3]*, Sanjeev Khudanpur*[1,2]

[1]Center for Language and Speech Processing & [2]Human Language Technology Center of Excellence
The Johns Hopkins University, Baltimore, MD 21218, USA.
[3]Xiaomi Corp., Beijing, China.
{kli26,khudanpur}@jhu.edu, dpovey@gmail.com

**ABSTRACT**

This paper proposes a parallel computation strategy and a posterior-based lattice expansion algorithm for efficient lattice rescoring with neural language models (LMs) for automatic speech recognition. First, lattices from first-pass decoding are expanded by the proposed posterior-based lattice expansion algorithm. Second, each expanded lattice is converted into a minimal list of hypotheses that covers every arc. Each hypothesis is constrained to be the best path for at least one arc it includes. For each lattice, the neural LM scores of the minimal list are computed in parallel and are then integrated back to the lattice in the rescoring stage. Experiments on the Switchboard dataset show that the proposed rescoring strategy obtains comparable recognition performance and generates more compact lattices than a competitive baseline method. Furthermore, the parallel rescoring method offers more flexibility by simplifying the integration of PyTorch-trained neural LMs for lattice rescoring with Kaldi.

***Index Terms***— lattice rescoring, Transformer, parallel computation, neural language models, automatic speech recognition

## 1. INTRODUCTION

Neural language models (LMs), including long short-term memory (LSTM) and Transformer based ones, have significantly improved performance over $n$-gram LMs in automatic speech recognition (ASR) [1, 2, 3, 4, 5, 6, 7]. Since it is challenging for one-pass decoding with a neural LM to obtain competitive performance with lower latency than a two-pass approach [8, 9, 10, 11, 12], a widely adopted way is still to use neural LMs to rescore $N$-best hypotheses (alternative word-sequences) or lattices that are decoded with an $n$-gram LM [13, 14, 15, 10, 16, 17, 18, 19]. A lattice is a compact representation of the hypothesis space for an utterance. $N$-best hypotheses only cover a small subspace. Thus, lattice rescoring usually outperforms $N$-best rescoring.

The key for lattice rescoring is to balance accuracy and efficiency since exact rescoring is not practical because it involves expanding a lattice into a linear or prefix tree structure and rescoring each hypothesis. A major speed bottleneck in lattice rescoring using a neural LM is the LM evaluation. Neural LM probabilities are usually computed on-the-fly and sequentially among hypotheses in a lattice during lattice traversal [14, 16, 19]. Though caching computed probabilities [16] or pruning-based methods [14, 19] can reduce the number of evaluations, the sequential order of LM evaluation in a lattice is inefficient. The process can be accelerated significantly by evaluating multiple hypotheses in parallel. However, given the graph structure of lattices, taking advantage of such speedup is challenging, especially for lattice rescoring methods that perform expansion and rescoring simultaneously. To enable batch computation, we convert a lattice into a minimal list of hypotheses that satisfy two conditions. First, every arc should be included in at least one hypothesis. Second, each hypothesis is the best path for at least one arc it contains. Computed neural LM scores are integrated back into the lattice for rescoring where score refer to negative log probabilities.

Lattice rescoring usually involves lattice expansion. Performing rescoring without changing the lattice structure is feasible, but it is generally not as good as with expansion [14]. To prevent expanded lattices from being too large, equivalence estimation of history states and pruning-based methods have been proposed [14, 15, 10, 16, 17, 19]. For example, an $n$-gram approximation method restricts lattice size by merging history states that share $(n-1)$ most recent words. But $n$-gram approximation based expansion method may sacrifice accuracy and waste computation on less likely paths. The general goal of lattice expansion is to make arcs on relatively probable paths have unique histories so that neural LM scores for them can be exact. To this end, we propose a new lattice expansion method that expands arcs only when their posteriors are larger than a threshold. Effectively, only more probable arcs are expanded so that arcs on sufficiently likely paths tend to have unique histories.

In summary, we propose an efficient lattice rescoring strategy that enables parallel computation of neural LM scores within a lattice. The strategy mainly involves operations such as posterior-based lattice expansion and lattice-to-list conversion using a proposed path cover algorithm. Furthermore, we experiment with a refined lattice rescoring strategy to further improve results. The proposed lattice-to-list conversion makes it easier to integrate neural LMs trained with PyTorch (or other tools) for efficient lattice rescoring in Kaldi [20]. Our code is open-source in Kaldi.

## 2. LATTICE CONVERSION AND EXPANSION

### 2.1. Lattices

A lattice is a graph representation of hypothesis space for an utterance and it can encode an exponential number of hypotheses with respect to the number of states. A lattice has one start state and a set of final states. A path in a lattice is consecutive transitions from the start state to a final state. Assuming lattices generated from first-pass decoding in a weighted finite state transducers (FST) based ASR system are determinized, each path represents a unique word-sequence.

Next, we will introduce the methods for lattice-to-list conversion, estimation of neural LM scores for each arc, and the posterior-based lattice expansion.

### 2.2. Lattice-to-List Conversion

Many lattice rescoring methods compute neural LM scores for arcs within a lattice dynamically during traversing the lattice. Considering that neural LM evaluation is a major speed bottleneck and the sequential order of traversing a lattice is inefficient, we propose a method to enable batch computation of hypotheses within a lattice. The general idea is to convert a lattice into a list of hypotheses that include every arc. Neural LM scores are then computed in parallel and merged back into the lattice.

A lattice $L$ can be viewed as a weighted directed acyclic graph $(V, E)$, where $V$ and $E$ denote the set of states (vertices) and arcs (edges), respectively. We define a *path cover* of lattice $L$ as a set of paths such that every arc in $E$ is included in at least one path in the set. A *minimal path cover* of $L$ is a path cover containing fewest possible paths. Our definition of path cover is different from the original definition in graph theory in which paths should cover states rather than arcs and they may start and end anywhere.

The size of a minimal path cover can be determined as

$$\sum_{s \in V} (\max(\deg_{\text{out}}[s] - \deg_{\text{in}}[s], 0)) \quad (1)$$

where $\deg_{\text{out}}[s]$ and $\deg_{\text{in}}[s]$ are the number of outgoing and incoming arcs of state $s$, because for each state, extra outgoing arcs should be covered by extra paths. Considering efficiency, the list of hypotheses converted from a lattice should be a minimal path cover.

However, since neural LM scores computed from the list directly affect rescoring, the quality of the hypotheses may matter more than the size. For each arc, a common choice for its history is the one in the best path that contains the arc. Therefore, a straightforward way of generating the list is: i) take the best path that contains each arc and sort them, ii) from the worst path to the best path, remove one if removing it does not cause any arc uncovered. While this method is not optimal since some generated paths are redundant and need to be removed. To fix it, we record the best path information for each arc during path generation so that it will not be regenerated if it already exists. The pesudocode for this method is shown below.

**Algorithm 1** A Constrained Path Cover Algorithm

**Input:** $L$: a lattice
**Output:** $O$: a list of paths, each is represented as a linear FST.

```
1:  procedure ConstrainedPathCover(L)
2:      ToplogicalSort(L)
3:      P ← []                          ▷ A list of pairs of a path and its cost
4:      α, β ← ViterbiForwardBackward(L)
5:      for s = 0 : S − 1 do                          ▷ Loop over states
6:          for e ∈ s.out do            ▷ Loop over outgoing arcs of s
7:              if best path including e is not generated then
8:                  p, c ← BestPathForAnArc(α,β,s,e)
9:                  P.append((p, c))
10:     Sort(P)                         ▷ Sort paths based on their costs
11:     O ← ConstructOutputLattice(P)
```

"Constrained" means that each path must be the best path for at least one arc it includes. The ViterbiForwardBackward function in Algorithm 1 computes best costs $\alpha$ and $\beta$ from the start state to every other state and from every final state to every other state, respectively. It also records the best predecessor and successor states of each state so that best paths can be found. The linear FSTs that represent the best paths are then converted into word-sequences for neural LM evaluation.

### 2.3. Estimation of Neural LM Scores

Neural LM scores of word-sequences converted from a lattice by the path cover algorithm need to be integrated back into the lattice for rescoring. If an arc in the lattice is shared by multiple paths, there are multiple neural LM scores associated with the arc. An approximation thus needs to be made to assign a single neural LM score for the arc. We experiment with three ways for obtaining the approximation: (i) by simply averaging the neural LM scores from the shared paths, (ii) obtaining a refined estimation using a weighted average, where weights are normalized values of neural LM scores of histories for the arc, and (iii) choosing the neural LM score from the lowest-cost path among the shared paths. Note, the lowest-cost path is not guaranteed to be the best path including the arc in the lattice because of the way the list of word-sequences generated. We thus refer to the third estimation as "semi-Viterbi" in the experiment.

### 2.4. Posterior-based Lattice Expansion

Lattice rescoring usually involves lattice expansion. To prevent the expanded lattices from blowing up in size, a commonly adopted approach is an $n$-gram approximation of history states. It merges history states with the same $(n-1)$ most recent words. However, it sacrifices accuracy since histories for computing neural LM scores may not be unique for many arcs. It also may expand out many paths with low probability, which is not optimal. To alleviate the problems of $n$-gram approximation, we propose a new expansion method. It expands arcs with posteriors higher than a threshold $\epsilon \in (0, 1)$. This method aims to make arcs on relatively probable paths have unique histories so that neural LM scores can be exactly computed. We refer to this method as posterior-based lattice expansion as present below.

**Algorithm 2** A Posterior-based Lattice Expansion Algorithm

**Input:** $L_{in}$: a lattice; $\epsilon$: a threshold for arc posteriors
**Output:** $L_{out}$: an expanded lattice

```
1:  procedure PosteriorExpansion(L_in, ε)
2:      ToplogicalSort(L_in)
3:      α ← []          ▷ Initialize forward logprobs for states in L_out
4:      β ← BackwardCosts(L_in)          ▷ β[0] is the total logprob
5:      L_out.SetStart(0)                ▷ Add start state 0 in L_out
6:      M[(0, 0)] ← 0          ▷ Initialize state map from a state pair
    (s_in, s_out) to a state s_out, where s_in ∈ L_in, s_out ∈ L_out
7:      Q.push((0, 0))                   ▷ Initialize the queue of state pairs
8:      while !Q.empty() do
9:          s_in, s_out ← DeQueue(Q)
10:         for e ∈ s_in.out do          ▷ Loop over outgoing arcs of s_in
11:             s_in^next ← e.nextstate
12:             a_e ← α[s_out] + e.weight          ▷ weight is −logprob
13:             e_post ← exp(a_e + β[s_in^next] − β[0])          ▷ arc posterior
14:             if e_post > ε then
15:                 s_out^next ← L_out.AddState()
16:                 Q.push((s_in^next, s_out^next))
17:                 M[(s_in^next, s_out^next)] ← s_out^next
18:             else if s_in^next is never copied to L_out then
19:                 Repeat line 15-17 and mark s_in^next as copied
20:             else
21:                 s_out^next ← GetCopyState(s_in^next)
22:             L_out.CreateArc(M[(s_in, s_out)], e, s_out^next)
23:             α[s_out^next] += a_e          ▷ Update forward logprobs
```

Algorithm 2 is a composition-type algorithm mainly implemented with a queue of state pairs. Each pair represents a state in

the input lattice and its copy state in the expanded lattice. The basic question for lattice expansion is whether an incoming arc should be split off from the rest of the incoming arcs to its destination state. The rule is to allocate a new copy of the destination state if the arc posterior is larger than $\epsilon$, otherwise transition to the original destination state. The threshold $\epsilon$ controls the size of expanded lattices such that larger $\epsilon$ results in smaller lattices. The backward costs $\beta$ are computed in advance while the forward costs $\alpha$ are computed dynamically during creating the expanded lattice.

## 3. LATTICE RESCORING STRATEGY

### 3.1. Non-iterative Lattice Rescoring

Combining the posterior-based lattice expansion algorithm and constrained path cover method, we propose an efficient lattice rescoring strategy that enables batch computation for words within a lattice. It mainly consists of two steps. First, lattices from first-pass decoding are expanded by the posterior-based lattice expansion method. We apply beam pruning before lattice expansion since in practice, we observe that it can reduce lattice size without hurting performance. Second, each expanded lattice is converted into a list of hypotheses. The neural LM scores that are computed in parallel are approximated when necessary and merged back into the expanded lattices for rescoring. We then find the best path in each rescored lattice and compute WERs. The proposed lattice rescoring strategy is referred to as "non-iterative" to distinguish it from a two-pass rescoring method introduced in section 3.2.

When neural LM scores are put back to lattices, they are interpolated with LM scores of the original $n$-gram LMThe interpolation involves removing a portion of the original LM scores from the lattice, which is implemented by FST composition.

### 3.2. Iterative Lattice Rescoring

To further improve the performance of the non-iterative lattice rescoring method described above, we propose a refined approach which introduces an extra rescoring stage. First, the original $n$-gram LM scores on decoded lattices are replaced with neural LM scores while the lattice structure is fixed. The proposed non-iterative rescoring strategy is then applied to the resulting lattices. We expect the integrated neural LM scores from the score replacement stage can result in better path cover lists and thus more accurate recognition results than the non-iterative method alone.

We refer to the refined rescoring approach as "iterative" since rescoring are executed twice. An alternative way is to perform the non-iterative rescoring twice. But it complicates the rescoring procedure and slows the rescoring speed by introducing an extra lattice expansion operation.

## 4. EXPERIMENTS

### 4.1. Datasets and Setups

We conduct experiments on the telephone speech corpus Switchboard (SWBD) which consists of approximately 260 hours of speech. We use Kaldi for acoustic model training and decoding. The acoustic model is factorized TDNNs [21] trained with the LF-MMI objective [22]. The audio data of English Fisher corpus is not included. We use Kaldi RNNLM [3] for text data preprocessing. There are a total of 34M words in the training dataset.

We experiment with both LSTM and Transformer. They are word-level LMs with vocabulary around 30K and trained with PyTorch. We use a 2-layer LSTM model with hidden dimension 650, and a 6-layer Transformer model with 8 heads and 512 hidden dimension. The LSTM and Transformer LMs have a total of 26.5M and 25M parameters respectively, both with parameter tying. We refer the readers to [23] for further details about the models.

Besides, we train an LSTM LM with Kaldi to compare the proposed rescoring method with the pruned lattice rescoring algorithm [19]. We do not use PyTorch-trained LMs for comparison since integrating them into the pruned rescoring algorithm is relatively complicated. That is also a motivation to develop the new lattice rescoring strategy.

### 4.2. Effect of Estimation Methods

We evaluate the performance of the three approximation methods for neural LM scores with both LSTM and Transformer models using the non-iterative lattice rescoring strategy. WERs in Table 1 show that the semi-Viterbi estimation consistently outperforms the other two. It is thus used in all the remaining experiments.

**Table 1**: WERs (%) on Hub5'00 (full set) of SWBD from the non-iterative lattice rescoring strategy with three estimation methods.

| Model | $\epsilon$ | Average | Weighted Average | Semi-Viterbi |
|---|---|---|---|---|
| LSTM | 0.5 | 10.8 | 10.8 | **10.7** |
| | 0.05 | 10.7 | 10.7 | **10.6** |
| Transformer | 0.5 | 10.7 | 10.7 | **10.6** |
| | 0.05 | 10.6 | 10.6 | **10.5** |

### 4.3. Analysis of Iterative Rescoring

We compare the non-iterative and iterative lattice rescoring methods using a Transformer LM. The parameter $\epsilon$ was set to 0.5 for the non-iterative method and 0.1 for the iterative one. The results of the iterative rescoring approach are in the last row in Table 2, and "Score replacement" refers to the refined operation of replacing $n$-gram LM scores on decoded lattices with neural LM ones. The 0.3% absolute WER reduction on Hub5'00 from score replacement shows the benefit of neural LM over the original $n$-gram LM. The observation that score replacement performs worse than the non-iterative rescoring approach alone indicates the value of lattice expansion.

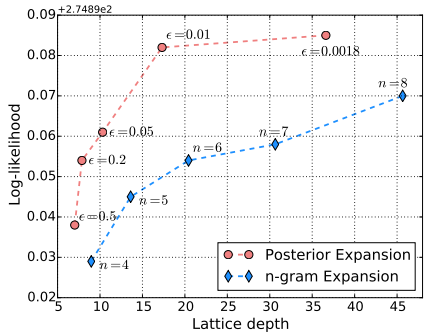
**Table 2**: WERs (%) from proposed lattice rescoring strategies with a Transformer LM.

| Rescoring Method | Hub5'00 | Swb | Callhm |
|---|---|---|---|
| Non-iterative ($\epsilon$ = 0.5) | 10.6 | 6.8 | 14.3 |
| Score replacement | 10.8 | 6.8 | 14.6 |
| Score replacement + Non-iterative | **10.3** | **6.6** | **14.0** |

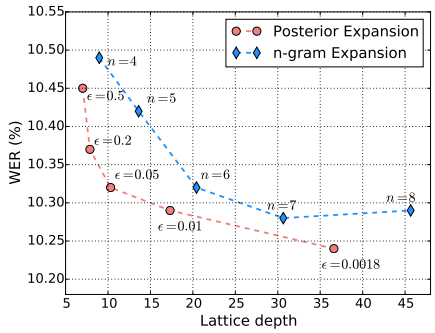
### 4.4. Comparison with $n$-gram Expansion

The performance of the proposed lattice expansion is compared with $n$-gram approximation based expansion using iterative rescoring strategy and a Transformer LM. The same pruning beam is used for a fair comparison. We evaluate the two methods through average log-likelihood of best paths and WER from rescored lattices in the Hub5'00 test set. Fig. 1(a) summarizes the log-likelihoods and lattice depths measured in frame-level average number of arcs for different $\epsilon$ values and $n$-gram orders. We can observe that the posterior-based expansion method results in higher log-likelihoods than $n$-gram expansion. The corresponding WERs in Fig. 1(b) show that the proposed expansion method generates more compact lattices with better recognition performance. Though WER is a more

noisy metric, it essentially reflects the tendency of the log-likelihood curve. We can infer from the results that lattice rescoring with the new expansion method can be faster for even better recognition performance than $n$-gram expansion.




(a) Log-likelihoods and lattice depths.




(b) WERs and lattice depths.

**Fig. 1**: Log-likelihoods, WERs, and lattice depths for different $\epsilon$ values and $n$-gram orders, respectively.

### 4.5. Comparison with Pruned Lattice Rescoring

We compare the proposed non-iterative lattice rescoring method with the pruned lattice rescoring algorithm [19] in Kaldi. An LSTM LM trained with Kaldi RNNLM toolkit is used for experiments. WERs and lattice depths (measured as average number of arcs cross a frame on rescored lattices of Hub5'00 full set) are present in Table 3. The WERs from using a Kneser–Ney (KN) smoothed 4-gram LM and $N$-best rescoring are shown for reference.

**Table 3**: WERs (%) and lattice depths from pruned lattice rescoring and the proposed non-iterative lattice rescoring.

| Method | WER | | | Lattice Depth |
|---|---|---|---|---|
| | Hub5'00 | Swb | Callhm | |
| 4-gram KN | 12.8 | 8.6 | 17.0 | 31.5 |
| $N$-best | 11.3 | 7.5 | 15.0 | - |
| Pruned (4-gram approx.) | 11.2 | **7.3** | 15.0 | 15.1 |
| Non-iterative ($\epsilon$ = 0.5) | **11.1** | 7.4 | **14.9** | **6.4** |

For both lattice rescoring methods, the interpolation weight of the LSTM LM (with the 4-gram LM) is 0.8. Compared with the pruned lattice rescoring, the non-iterative rescoring strategy obtains competitive performance and generates smaller lattices.

### 4.6. Speedup

In the proposed lattice rescoring strategies, the main speedups are from beam pruning and parallel computation of neural LM scores. Beam pruning reduces size of lattices by 3-4 times without degrading WERs in our experiments. The speedup by the batch computation varies with batch size and models. Compared with sequential evaluation, batch computation gives around 5-6 speedup with the PyTorch LSTM in a non-iterative rescoring setup.

Compared with $N$-best rescoring (with different $N$s) in a parallel computation mode within each lattice as well, the proposed non-iterative lattice rescoring method accelerates the process by 1-3 times while obtains the same WERs. Besides, lattice rescoring with the Transformer LM is faster than with the PyTorch LSTM LM. This is expected considering the non-recurrent structure and fewer total parameters of the Transformer LM.

### 4.7. WERs on SWBD

We present WERs on SWBD with both $N$-best and lattice rescoring in Table 4. The Transformer LM was used in both non-iterative and iterative rescoring methods. For $N$-best rescoring with the PyTorch trained LSTM, the state-carry trick [24] was used. $N$-best rescoring with the Transformer LM obtains slightly better performance than the PyTorch LSTM, consistent with the results from lattice rescoring in Table 1. As expected, both non-iterative and iterative lattice rescoring methods obtain better recognition performance than $N$-best rescoring, and smaller expansion threshold generally leads to better WER. However, since the computation cost for the iterative rescoring method is roughly doubled, non-iterative rescoring is more practical considering latency.

**Table 4**: WERs (%) from proposed lattice rescoring strategies with a Transformer LM. $N$ is set to 20 for $N$-best rescoring.

| Method | Hub5'00 | Swb | Callhm |
|---|---|---|---|
| 4-gram KN | 12.8 | 8.6 | 17.0 |
| $N$-best (LSTM) | 10.9 | 7.1 | 14.6 |
| $N$-best (Transformer) | 10.8 | 7.2 | 14.4 |
| Non-iterative ($\epsilon$ = 0.5) | 10.6 | 6.8 | 14.3 |
| Non-iterative ($\epsilon$ = 0.005) | 10.4 | 6.8 | 14.0 |
| Iterative ($\epsilon$ = 0.1) | 10.3 | 6.6 | 14.0 |
| Iterative ($\epsilon$ = 0.001) | **10.2** | **6.5** | **13.9** |

## 5. CONCLUSION AND FUTURE WORK

In this work, we propose an efficient lattice rescoring strategy that computing neural LM scores within a lattice in parallel. The proposed method mainly consists of a posterior-based lattice expansion algorithm and a constrained path cover method for converting a lattice into a list representation. We also propose a refined rescoring strategy for further accuracy improvement. Experiments on SWBD show that the posterior-based lattice expansion outperforms $n$-gram expansion. The proposed rescoring strategy obtains comparable WERs with marginally faster speed compared with the pruned lattice rescoring. To achieve the same recognition performance, the proposed rescoring method generally is faster than $N$-best rescoring in batch computation mode as well.

The proposed parallel rescoring strategy makes it easier and more flexible to perform lattice rescoring with PyTorch-trained neural LMs in Kaldi. In the future, we plan to explore more effective ways of lattice expansion for further speedup.

## 6. REFERENCES


[1] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur, "Recurrent neural network based language model," in *Proc. of Interspeech*, 2010.

[2] Xie Chen, Xunying Liu, Mark JF Gales, and Philip C Woodland, "Recurrent neural network language model training with noise contrastive estimation for speech recognition," in *Proc. of ICASSP*, 2015.

[3] Hainan Xu, Ke Li, Yiming Wang, Jian Wang, Shiyin Kang, Xie Chen, Daniel Povey, and Sanjeev Khudanpur, "Neural network language modeling with letter-based features and importance sampling," in *Proc. of ICASSP*, 2018.

[4] Neil Zeghidour, Qiantong Xu, Vitaliy Liptchinsky, Nicolas Usunier, Gabriel Synnaeve, and Ronan Collobert, "Fully convolutional speech recognition," *arXiv preprint arXiv:1812.06864*, 2018.

[5] Gabriel Synnaeve, Qiantong Xu, Jacob Kahn, Tatiana Likhomanenko, Edouard Grave, Vineel Pratap, Anuroop Sriram, Vitaliy Liptchinsky, and Ronan Collobert, "End-to-end asr: from supervised to semi-supervised learning with modern architectures," *arXiv preprint arXiv:1911.08460*, 2019.

[6] Kazuki Irie, Albert Zeyer, Ralf Schlüter, and Hermann Ney, "Language modeling with deep transformers," in *Proc. of Interspeech*, 2019.

[7] Ke Li, Zhe Liu, Tianxing He, Hongzhao Huang, Fuchun Peng, Daniel Povey, and Sanjeev Khudanpur, "An empirical study of transformer-based neural language model adaptation," in *Proc. of ICASSP*, 2020.

[8] Takaaki Hori, Yotaro Kubo, and Atsushi Nakamura, "Real-time one-pass decoding with recurrent neural network language model for speech recognition," in *Proc. of ICASSP*, 2014.

[9] Yongzhe Shi, Wei-Qiang Zhang, Meng Cai, and Jia Liu, "Efficient one-pass decoding with nnlm for speech recognition," *IEEE Signal Processing Letters*, 2014.

[10] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter, "From feedforward to recurrent lstm neural networks for language modeling," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 517–529, 2015.

[11] Eugen Beck, Wei Zhou, Ralf Schlüter, and Hermann Ney, "Lstm language models for lvcsr in first-pass decoding and lattice-rescoring," *arXiv preprint arXiv:1907.01030*, 2019.

[12] Javier Jorge, Adria Giménez, Javier Iranzo-Sánchez, Joan Albert Silvestre-Cerda, Jorge Civera, Albert Sanchis, and Alfons Juan, "Lstm-based one-pass decoder for low-latency streaming," in *Proc. of ICASSP*, 2020.

[13] Anoop Deoras, Tomáš Mikolov, and Kenneth Church, "A fast re-scoring strategy to capture long-distance dependencies," in *Proc. of EMNLP*, 2011.

[14] Martin Sundermeyer, Zoltán Tüske, Ralf Schlüter, and Hermann Ney, "Lattice decoding and rescoring with long-span neural network language models," in *Proc. of Interspeech*, 2014.

[15] Xunying Liu, Yongqiang Wang, Xie Chen, Mark JF Gales, and Philip C Woodland, "Efficient lattice rescoring using recurrent neural network language models," in *Proc. of ICASSP*, 2014.

[16] Xunying Liu, Xie Chen, Yongqiang Wang, Mark JF Gales, and Philip C Woodland, "Two efficient lattice rescoring methods using recurrent neural network language models," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 8, pp. 1438–1449, 2016.

[17] Xie Chen, Xunying Liu, Anton Ragni, Yu Wang, and Mark JF Gales, "Future word contexts in neural network language models," in *Proc. of ASRU*, 2017.

[18] Shankar Kumar, Michael Nirschl, Daniel Holtmann-Rice, Hank Liao, Ananda Theertha Suresh, and Felix Yu, "Lattice rescoring strategies for long short term memory language models in speech recognition," in *Proc. of ASRU*, 2017.

[19] Hainan Xu, Tongfei Chen, Dongji Gao, Yiming Wang, Ke Li, Nagendra Goel, Yishay Carmiel, Daniel Povey, and Sanjeev Khudanpur, "A pruned rnnlm lattice-rescoring algorithm for automatic speech recognition," in *Proc. of ICASSP*, 2018.

[20] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al., "The kaldi speech recognition toolkit," in *Proc. of ASRU*, 2011.

[21] Daniel Povey, Gaofeng Cheng, Yiming Wang, Ke Li, Hainan Xu, Mahsa Yarmohammadi, and Sanjeev Khudanpur, "Semi-orthogonal low-rank matrix factorization for deep neural networks.," in *Proc. of Interspeech*, 2018.

[22] 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 *Proc. of Interspeech*, 2016.

[23] Ke Li, Daniel Povey, and Sanjeev Khudanpur, "Neural language modeling with implicit cache pointers," in *Proc. of Interspeech*, 2020.

[24] Kazuki Irie, Albert Zeyer, Ralf Schlüter, and Hermann Ney, "Training language models for long-span cross-sentence evaluation," in *Proc. of ASRU*, 2019.