CUED RNNLM Toolkit

Xie(Jeff) Chen
xc257@eng.cam.ac.uk

October 31, 2015

Abstract

This document includes a more detailed description about the usage of CUED-RNNLM Toolkit introduced in [1].

1 Language Model

Language model is a crucial component in many fields, such as speech recognition, machine translation, spoken language understanding. The aim of language model is to estimate the probability of a given sentence $\mathbf{W}$ as shown in Eqn 1.

$$P(\mathbf{W}) = \prod_i P(w_i|w_{i-1}..w_1) \approx \prod_i P(w_i|h_i) \tag{1}$$

Due to their good generalization and easy implementation, n gram LMs have been dominating the field of language model for several decades. However, there are two long existing drawbacks for n gram LMs. One is data sparsity. Smoothing techniques are required for robust parameter estimation[2]. The other problem lies in the n order Markov assumption (where $h_i =< w_{i-1}..w_{i-N+1} >$). The predicted probability is only related to the preceding $n-1$ words, while longer history information is ignored. Recurrent neural network provides a good solution for these two issues. Each word is projected into a small, continues space and the whole sentence history is able to be modelled using the recurrent connection. Promising results have been reported in many areas and tasks by combining RNNLM and n gram LM, which raised research upsurge and led to the development of a range of potential applications during recent years.

However, the heavy computation and slow training speed obstruct the application of RNNLMs in processing large amount of data. The efficient training of RNNLMs on large amount of data as well as model size is necessary in order to fully explore the power of RNNLMs. We developed an open-source toolkit which is more suitable for training large amount of data with large model size, which is named CUED-RNNLM Toolkit.

The remaining of this document is organized as follows. Section 2 gives a general overview of recurrent neural network language models (RNNLMs). The main functions of the CUED-RNNLM toolkit are introduced in Section 3. The usage of the toolkit is depicted in Section 4. Section 5 gives the experimental results in AMI corpus.

2 Overview of RNNLMs

RNNLMs [3] represent the full, non-truncated history $h_i =< w_{i-1}, \ldots, w_1 >$ for word w_i using a 1-of-k encoding of the previous word w_{i-1} and a continues vector v_{i-2} for the remaining context. For an empty history, this is initialised. An out-of-vocabulary (OOV) input node can also be used to represent any input word not in the chosen recognition vocabulary. The topology of the recurrent neural network used to compute LM probabilities $P_{\text{RNN}}(w_i|w_{i-1}, v_{i-2})$ consists of three layers. The full history vector, obtained by concatenating w_{i-1} and v_{i-2}, is fed into the input layer. The hidden layer compresses the information from these two inputs and computes a new representation v_{i-1} using a sigmoid activation to achieve non-linearity. This is then passed to the output layer to produce normalized RNNLM probabilities using a softmax activation, as well as recursively fed back into the input layer as the "future" remaining history to compute the LM probability for the following word $P_{\text{RNN}}(w_{i+1}|w_i, v_{i-1})$.

For the sake of simplicity, RNNLMs discussed in this section contains single hidden layer. RNNLMs with multiple hidden layers could be easily extended and supported in the toolkit.

2.1 RNNLM with full output layer

An example RNNLM architecture with an unclustered, full output layer is shown in Figure 1. RNNLMs can be trained using an extended form of the standard back propagation algorithm, back propagation through time (BPTT) [4], where the error is propagated through the recurrent connections back for a specific number of time steps, for example, 4 or 5 [5]. This allows RNNLMs to keep information for several time steps in the hidden layer. To reduce the computational cost, a shortlist [6, 7] on output layer limited to the most frequent words can be used. To reduce the bias to in-shortlist words during RNNLM training and improve robustness, an additional node is added at the output layer to model the probability mass of out-of-shortlist (OOS) words [8, 9, 10].

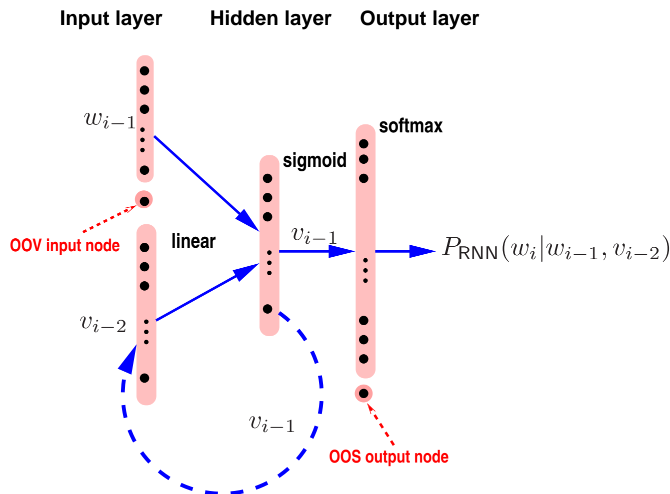

Figure 1: *A full output layer RNNLM with OOS nodes.*

RNNLMs with full output layer are computationally heavy during both training and evaluation, especially when large output vocabulary is applied. Assuming that the size of input word layer is S, the hidden layer is H and size of output layer is V. For each word in the training data, the forward operation requires $H + H * H + H * V$ multiplication totally. And for back propagation, the computation is comparable to the forward pass. Normally the output layer size V is significantly larger than hidden layer size H, given a smaller hidden layer. The computation complexity for each train sample is $O(H * V)$.

2.2 RNNLM with class output layer

An alternative way to improve computation efficiency is using factorizated output layer, e.g. classed based output layer, as illustrated in Figure 2.

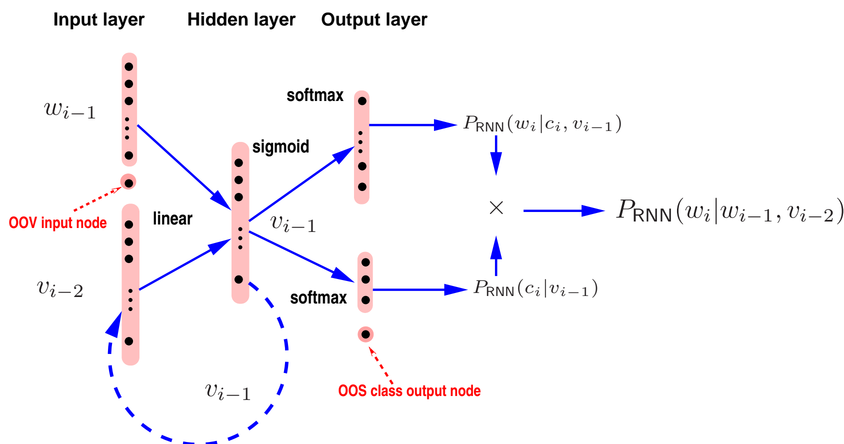

Figure 2: *A class based output layer RNNLM with OOS nodes.*

In class based RNNLM (CRNNLM), the output layer is factorizated into two separate layers, one is the word layer as in FRNNLM, and the other is the class layer. Each target word is assigned to a distinct class. The word probability in CRNNLM consists of two components, class probability and word probability given the assigned class. Firstly the class probability $p(c_i|v_i)$ is calculated, where c_i is the class assignment of word w_i. Then all words belonging to class c_i are calculated and normalized, which is the probability of $p(w_i|c_i, v_i)$. The word probability could be obtained from the multiplication of these two probabilities. Normally the number of class and number of word within one class are significantly smaller than the output word vocabulary.

Averagely, there are about $\sqrt{V}$ classes and each class contains $\sqrt{V}$ words in the output layer. The computation complexity becomes $O(H * \sqrt{V})$ for each train sample, assumming H is still significantly smaller than $\sqrt{V}$. Hence, by using this structure, the computation could be reduced significantly. An extension of this simple class based output could be obtained by using hierarchical output layer [11] for more efficient computation.

However, there are two potential drawbacks for CRNNLMs. The first is that the performance is sensitive to the class assignment, the other is that it will be complicated to apply bunch (i.e. minibatch) mode for training. The words

within the same bunch maybe from different classes, which requires different submatrice in output word layer to be called.

2.3 Spliced Sentence Bunch

One choice for efficient training is bunch (i.e. minibatch) based training, which was applied widely in deep neural networks (DNNs). However, for recurrent neural networks, this becomes difficult as the prediction of current target requires the whole sequence history. Previous work tried to align multiple sentences for RNNLM training using bunch mode. However, the highly variable sentence length introduces computational inefficiency. To solve this issue, spliced sentence bunch [12] was proposed to minimize the synchronization overhead between bunch streams introduced by sentence length variation. The idea of spliced sentence bunch is illustrated in Figure 3. Multiple sentences are spliced into one stream. N is the bunch size. During training, an input word vector of N dimension is formed by taking one word from each stream. The target word vector is a group of following words in each stream. In this case, RNNLMs could be trained using bunch mode. The implementation details could be found in [12].

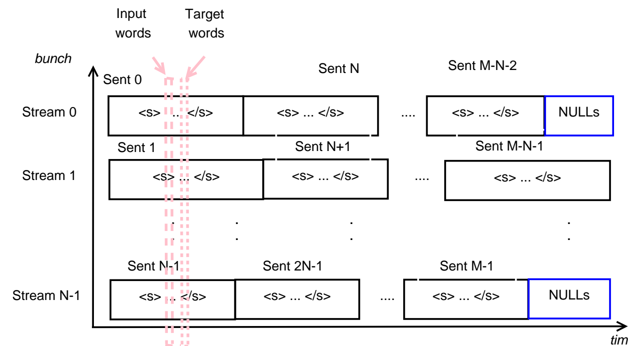

Figure 3: *RNNLM training with spliced sentence bunch*

2.4 Model Structure in CUED-RNNLM

In CUED RNNLM, we aim to utilize the parallel power of GPU for efficient computation. Hence, we mainly use full output RNNLM for training and evaluation[1]. Sentence bunch mode was used for efficient training. To sum up, there are mainly three distinct features for the RNNLM trained in the CUED-RNNLM toolkit.

- **full output layer**
  The use of class layer in the output layer could be very efficient when RNNLM is trained sample by sample on CPU. However, it is not suitable for bunch (i.e. minibatch) based training. Samples within the same bunch are probably from different class, which requires different submatrix in

[1] class based RNNLMs are also supported

output word layer. Hence, we choose to use full output layer RNNLM and utilize of the computational power of GPU for efficient training.

- **specified input, output list, with OOS node**
  The vocabulary used for RNNLM training is important as it defines the coverage of words to be estimated. In many situation, the RNNLM vocabulary is task dependent. Taking speech recognition for example, the vocabulary suitable for RNNLM training maybe not consistent as the vocabulary used for decoding. In our default configuration, we used the intersection between RNNLM train vocabulary and decode vocabulary as input layer, and use the most frequent words in the input layer as output layer. An OOV node is added in the input layer and OOS node added in output layer for out of shortlist words, which is shown in Figure 1.

- **GPU supported**
  In CUED-RNNLM toolkit, we only provide GPU based training, as the CPU based training is expected to be very slow due to the use of full output layer and large output list.

3 CUED-RNNLM Toolkit

3.1 Train

The most common objective function used for RNNLM training is cross entropy (CE). However, in test time, the explicit normalization in output layer for full output layer RNNLM is quite slow on CPU. To improve the evaluation efficiency, two improved training criteria are also implemented. They are variance regularization (VR) and noise contrastive estimation (NCE) respectively.

- Cross Entropy (CE)
  The objective function of CE is shown in Eqn. 2.

$$J^{\mathsf{CE}}(\theta) \;=\; -\frac{1}{N_w}\sum_{i=1}^{N_w} \ln P_{\mathsf{RNN}}(w_i|h_i) \tag{2}$$

  where N_w is the total number of training words. The CE based RNNLM could be trained on GPU efficiently using bunch (i.e. minibatch) mode, and full output RNNLM without factorization in the output layer is generated. Cross entropy is also the default training criterion for training. However, the softmax layer in output layer requires the computation of normalization term, as shown in Eqn (3),

$$P_{\mathsf{RNN}}(w_i|h_i) = \frac{e^{v_{i-1}^T a_i}}{\sum_j e^{v_{i-1}^T a_j}} = \frac{e^{v_{i-1}^T a_i}}{Z_i} \tag{3}$$

  where a_i is the weight vector associated with word w_i. The computation of normalization term Z_i is expensive during both training and test time.

- Variance Regularization
  Variance regularization (VR) adds the variance of normalization term into

the objective function, which could be written as below,

$$J^{VR} = J^{CE} + \frac{\gamma}{2}\frac{1}{N_w}\sum_{i=1}^{N_w}((\ln(Z_i) - (\overline{LnZ}))^2) \tag{4}$$

where Z_i is the normalization term for ith training sample before softmax function in output layer. $\overline{LnZ}$ is the mean of log normalization term over training data, which is estimated dynamically on each minibatch and used as constant. γ is the penalty term on variance to tune the effect of variance regularization. More details could be find at [13].

The normalization term is constraint explicitly during training and could be viewed as a constant C. In test time, the RNNLM probability could be approximated using the unnomalized probability, as shown in Eqn 5.

$$P_{\mathsf{RNN}}(w_i|h_i) \approx \frac{e^{v_{i-1}^T a_i}}{C} \tag{5}$$

- Noise Contrastive Estimation (NCE)
  In NCE training, each word in the training corpus is assumed to be generated by two distributions independently, one is the data distribution, which is RNNLM, and the other is noise distribution, where unigram is normally chosen as noise distribution. The objective function of NCE is to discriminate these two distributions given the training data and a group of randomly generated noise samples. The objective function could be written as below,

$$\begin{aligned} J^{\mathsf{NCE}}(\theta) &= -\frac{1}{N_w}\sum_{i=1}^{N_w}\Big(\ln P(C_{w_i}^{\mathsf{RNN}} = 1|w_i, h_i) \\ &\quad + \sum_{j=1}^{k}\ln P(C_{\check{w}_{i,j}}^{n} = 1|\check{w}_{i,j}, h_i)\Big) \end{aligned} \tag{6}$$

  where w_i is the ith target word, $\check{w}_{i,j}$ is the jth noise word generated for ith word, and k is the number of noise sample. $P(C_{w_i}^{\mathsf{RNN}} = 1|w_i, h_i)$ is the posterior probability of word w_i is generated by RNNLM, and $P(C_{\check{w}_{i,j}}^{n} = 1|\check{w}_{i,j}, h_i)$ is the posterior probability of word $\check{w}_{i,j}$ is generated by noise distribution. The details could be find at [14]. In training stage, only weights associated with the sampled noise words and target words need to be calculated and updated in each bunch. Hence, the computation in output layer could be reduced dramatically. Besides, the variance of normalization term is constraint implicitly during training. In test time, the unnomalized probability shown in Eqn 5 could be applied as variance regularisation for fast evaluation.

3.2 Evalution

In order to evaluate RNNLMs trained by CUED-RNNLM, the calculation of perplexity (PPL) and N-best rescoring are provided. Lattice rescoring is also

supported by using an extended version of HTK tool HLRescore [2]. This part of code is implemented and run on CPU.

- Perplexity (PPL)

  The PPL could be evaluated alone, or linearly interpolated with other LM. If it is used for interpolation with other LM, the stream file of the other LM is required. In current version, the perplexity could only be calculated using CPU, and it will be slow due to the normalization on output layer (GPU based PPL calculation will be added later).

- N-best rescoring

  The N-best rescoring is quite similar to the use of perplexity calculation. One difference lies here is that the unnomalized probability could be applied for RNNLMs trained by VR or NCE for fast evaluation. When unnomalized probability is applied, the option "-lognorm <float>" need to be specified in the command line. The log of normalization term will be substituted by the constant value of lognorm.

- Lattice rescoring

  The RNNLM could be used for lattice rescoring as well. HTK lattice is supported by using an extended version of HLRescore in HTK Toolkit. Besides, tools to convert Kaldi lattice to HTK lattice are also provided. Hence, both Kaldi and HTK format lattice are supported.

3.3 Sample

Another application of RNNLM is to generate a large quantities of words. Then a n gram LM is trained on these sampled words, interpolated with the baseline n gram LM[15]. This interpolated n gram LM could be used directly for decoding or lattice rescoring. Previous research has shown that it can retain part of the improvements from the original RNNLM over the baseline n-gram LM.

3.4 Other options

- additional input feature

  The additional input feature is supported to incorporate other informative feature in the toolkit. E.g. the topic representation vector[16]. The feature is appended in input layer and each sentence are assumed to use the same appended feature.

- class based output layer

  The class based output layer is also supported. In this way, an additional class layer is added in the output layer as shown in Figure 2 and trained on GPU.

- multiple hidden layers

  RNNLMs with more than one hidden layer are supported. Currently, only the first hidden layer is allowed to be recurrent connections.

[2] the source code of this extended HLRescore could be downloaded in CUED-RNNLM website

- data shuffle

  In many applications, the training data is from different sources. The order of training data for RNNLM could impact the performance. In order to obtain good performance on the in-domain test data, a general practice is to present the out of domain data to the network first during RNNLM training, before the more important in-domain training data is processed. Taking this into consideration, the training data is not shuffled during training. It is therefore recommended to shuffle sentences in each source of data as a separate preprocessing step, while keeping the order of data sources.

- prerequisite

  The toolkit doesn't require the use of other third-party library except standard CUDA library for GPU computation. Various CUDA versions from 5.0 to 7.5 were supported in our experiments. The debug infomation and output are similar as RNNLM toolkit[17]

4 Usage

4.1 Description

This program **rnnlm** could be used for training, evaluation (perplexity or word error rate) or sampling words.

4.2 Use

rnnlm is invoked by typing the command line
rnnlm {-train | -ppl | -nbest | -sample} [options]

-train	RNNLM training (GPU supported only)
-ppl	RNNLM evaluation for calculation of perplexity (CPU supported only)
-nbest	RNNLM evaluation for N best rescoring (CPU supported only)
-sample	Sample a specified number of words from a well-trained RNNLM (GPU supported only)

4.3 Train

• Description
RNNLMs are trained on GPU for CUED-RNNLM toolkit. Three training criteria could be chosen, they are cross entropy (CE), variance regularisation (VR) and noise contrastive estimation (NCE) repsectively. The input and output list used to construct the input and output layer need to specified. Various options are supported for the training of RNNLMs, which could be found from the following command options.
• Use
The RNNLM training is invoked by typing the command line
rnnlm -train [options]
The detailed operation for RNNLM training is controlled by the following command line options.

-trainfile	*string*	Specify the train text file
-validfile	*string*	Specify the valid text file
-inputwlist	*string*	Specify the input word list
-outputwlist	*string*	Specify the output word list
-feafile	*string*	Specify the feature matrix file if additional feature is appended in the input layer
-device	*int*	Specify the GPU id for RNNLM training (default: 0)
-minibatch	*int*	Specify the minibatch (i.e. bunch) size for training (default: 32)
-layers	*int* : *int*... : *int*	Specify the model structure and size of RNNLM. The first integer is the size of input word layer, the last integer is the size of output word layer. The middle integers set the number of hidden nodes in each hidden layer
-bptt	*int*	Specify the step of back propogation through time (default: 5)
-bptt-delay	*int*	Specify the delayed step of update for BPTT (default: 8)
-traincrit	*string*	Specify the train criterion [ce (default) | nce | vr]
-learnrate	$float$	Specify the initial learning rate (default: 0.8)
-vrpenalty	$float$	Specify the penalty of variance regularization γ (default: 0.0)
-ncesample	*int*	Specify the number of noise sample in NCE training (default: 10)
-nclass	*int*	Specify the number of class in output layer. If nclass is larger than 0, class based RNNLM will be trained (default: 0)
-lognormconst	$float$	Specify the log norm const in NCE training, or specify the log of normalization term in test time (default: 0.0)
-cachesize	*int*	Specify the cache size. When cachesize equals to 0, all train samples will be loadod into memory without cache (default: 0)
-randseed	*int*	Specify the rand seed for random value generation (default: 1)
-readmodel	*string*	Specify the existed RNNLM model to continue training
-writemodel	*string*	Specify the RNNLM model to be written
-independent	*int*	Specify sentence independent or dependent mode. Sentence independent model to be trained by default (default: 1)
-binformat		Specify the model is stored with binary format
-nthread	*int*	Specify the number of thread for computation (default: 1)
-min_improvement	$float$	Specify the minimum improvement to stop RNNLM training (default: 1.003)
-debug	*int*	Specify the debug level (default: 1)

- Example

The following example command will train RNNLM with cross entropy. There is a single hidden layer with 200 hidden nodes. The input and output layer sizes are 31858 and 20002 respectively. Minibatch size is 64, initial learning rate is 1.0 for each minibatch.

```
rnnlm   -train
        -trainfile data/train.dat
        -validfile data/dev.dat
        -device 0
        -minibatch 64
        -layers 31858:200:20002
        -bptt 5
        -traincrit ce
        -inputwlist ./wlists/input.wlist
        -outputwlist ./wlists/output.wlist
        -writemodel h200.mb64/rnnlm.txt
        -debug 2
        -independent 1
        -learnrate 1.0
```

4.4 Evalution

• Description
The evaluation of RNNLMs are run on CPUs. RNNLMs could be evaluated in terms of perplexity or word error rate. Besides, unnomalized probability as shown in Eqn 5 could be applied for calculating WER for fast evaluation.
• Use
The RNNLM evaluation is invoked by typing the command line
rnnlm {-ppl | -best} [options]

-readmodel	*string*	Specify the RNNLM model to be read
-binformat		Specify the model is read with binary format
-testfile	*string*	Specify the test file
-feafile	*string*	Specify the feature file
-inputwlist	*string*	Specify the input word list
-outputwlist	*string*	Specify the output word list
-lambda	*float*	Specify the interpolation weight for RNNLM (default: 0.5)
-fullvocsize	*int*	Specify the full vocabulary size, all OOS words will share the probability
-nglmstfile	*string*	Specify the ngram lm stream file for interpolation
-nthread	*int*	Specify the number of thread for computation (default: 1)
-debug	*int*	Specify the debug level (default: 1)

• Example

```
rnnlm  -ppl
       -readmodel h200.mb64/rnnlm.txt
       -testfile data/test.dat
       -inputwlist ./wlists/input.wlist
       -outputwlist ./wlists/output.wlist
       -nglmstfile ng.st
       -lambda 0.5
       -debug 2

rnnlm  -nbest
       -readmodel h200.mb64/rnnlm.txt.nbest
       -testfile data/test.dat
       -inputwlist ./wlists/input.wlist
       -outputwlist ./wlists/output.wlist
       -nglmstfile ng.st
       -lambda 0.5
       -debug 2
```

4.5 Sample

• Description
RNNLM could be used to randomly sample sentences from a well-trained RNNLM. If this option is chosen, a specified amount of words will be sampled. This option is run on GPU for efficient computation.
• Use
The RNNLM sample is invoked by typing the command line
rnnlm -sample [options]

-readmodel	*string*	Specify the RNNLM model to be read
-sampletextfile	*string*	Specify text file for sampling words from RNNLM
-unigramfile	*string*	Specify unigram lm file
-nsample	*int*	Specify number of sample word from RNNLM (default: 1000)
-debug	*int*	Specify the debug level (default: 1)

• Example

```
rnnlm   -sample
        -readmodel h200.mb64/rnnlm.txt
        -inputwlist ./wlists/input.wlist.index
        -outputwlist ./wlists/output.wlist.index
        -unigramfile ./uglm.txt
        -sampletextfile text/10k.txt
        -nsample 10000
        -minibatch 50
        -fullvocsize 49413
        -device 0
        -debug 2
```

4.6 File Format

- **trainfile, validfile, testfile**

each line contains one sentence. The sentence boundaries ($< s >$ and $< /s >$) are optional. An example could be found below.

```
OKAY
DO YOU WANT TO INTRODUCE YOURSELF AGAIN
SO WHO WOULD LIKE TO GO FIRST
UM WELL THIS IS THE KICK OFF MEETING FOR OUR OUR PROJECT
```

- **inputwlist, outputwlist**

each line contains a pair of wordid($< int >$) and word($< string >$). For the training of CRNNLM, if it is trained from existed model, each line in output word list includes three elements, they are <wordid word classid > respectively. In both input and output layers, the first node is sentence boundary. ($< s >$ for input layer, $< /s >$ for output layer) and the last node is out-of-short list node ($< OOS >$). The two nodes are optional for the inputwlist and outputwlist files.

```
0 I
1 YOU
2 AND
3 THE
4 TO
...
32633 ROWE
32634 MERCED
```

- **feafile**

feafile is used for RNNLM training with additional input feature. The first line includes the number and dimension of feature. From the second line, each line contains a feature vector of float value, of specified dimension. When feafile is applied, in the trainfile and validfile, each line is started by an integer, which indicates the feature index. Each line will share the same feature.

```
4 3
0.1 0.8 0.1
0.5 0.3 0.2
0.3 0.3 0.4
0.1 0.2 0.7
```

- **unigramfile**

an ARPA formatted unigram file will be used to generate OOS words when RNNLMs randomly sample words.

4.7 Recipe

The example of training, evaluation, sampling could be found from the recipe file (in example.AMI/Readme).

5 Experiments

Experiments are conducted on the AMI meeting corpus [18]. 78 hours of speech was used in acoustic model training. 8 meetings were kept from the training set and used as the development and test sets. A Kaldi acoustic model training recipe featuring sequence training [19] was applied for DNN training. FMLLR transformed MFCC feature was used as input and 4000 senones clustered as target. DNN was trained with 6 hidden layers, each layers with 2048 hidden nodes. The first part of the Fisher corpus of 13M words was also used to further improve language modelling performance. A 49k word decoding vocabulary was used. A 33k RNNLM input vocabulary was constructed from the intersection between the decoding vocabulary and all words present in the LM training data. The 22k most frequent words were then selected as output vocabular. BPTT was applied in RNNLM training with a step of 5. All RNNLMs in this paper use one hidden layer.

The first experiment is based on 1M AMI transcription for language model construction. One hidden layer including 200 hidden nodes is used for RNNLMs. Class based RNNLM (CRNNLM) was trained with RNNLM toolkit[3][17] with 200 classes. FRNNLMs were trained by cross entropy(CE), variance regularization(VR) and noise contrastive estimation (NCE) respectively. In ASR experiment, RNNLMs are used for N best and lattice rescoring. The experiment results are shown in Table 2. It could be seen that RNNLMs obtain significant gains over baseline 3 gram LM in terms of PPL and WER. FRNNLMs are slightly better than CRNNLMs with CE. Lattice rescoring and 50 best rescoring give comparable results. Unnomalized probability as shown in Eqn 5 is applied for VR and NCE models. According to the results, VR don't affect performance, while much faster evaluation speedup is achieved by using unnomalized probability. NCE has a slight performance degradation, but largely improves both train and test speed.

The next experiment is to evaluate the performance of RNNLMs using additional Fisher data. A pruned 3-gram LM was used in the first-pass decoding and followed by lattice rescoring using an un-pruned 4-gram LM. For RNNLM training, AMI corpus (1M) was processed after the Fisher data (13M) during training. RNNLMs with 512 hidden nodes were trained using the cross entropy criterion. Table 2 shows the performance of RNNLMs trained by **RNNLM** and **CUED-RNNLM**. RNNLMs give significant perplexity and word error rate (WER) improvements over the baseline 4-gram LM. The full output layer RNNLM trained by **CUED-RNNLM** toolkit slightly outperformed the class based RNNLM. Rescoring lattices and 500-best lists gave comparable performance. Additional WER reduction of 0.2% absolute was obtained by confusion network (CN) decoding performed on the RNNLM rescored lattices, while CN decoding using the rescored 50-best lists gave no improvement.

[3]the latest version v0.4 used in this paper

Table 1: Performance of CRNNLMs and FRNNLMs trained using **RNNLM** and **CUED-RNNLM** toolkits on 1M (AMI) data

LM Type	Train Crit	Re score	PPL		WER	
			dev	eval	dev	eval
3g	-	-	93.6	82.8	25.2	25.4
+CRNN	CE	lattice	83.3	75.2	24.0	24.1
		50 best			23.9	24.1
+FRNN	CE	lattice	81.0	71.7	24.0	23.9
		50 best			23.9	24.0
	VR	lattice	80.4	71.6	23.9	24.0
		50 best			23.9	23.9
	NCE	lattice	81.1	72.8	24.1	24.1
		50 best			24.0	24.1

Table 2: Performance of CRNNLMs and FRNNLMs trained using **RNNLM** and **CUED-RNNLM** toolkits on 14M (AMI+Fisher) data

LM Type	Re score	PPL		WER	
		dev	eval	dev	eval
3g	-	84.5	79.6	24.2	24.7
4g	lattice	80.3	76.3	23.7	24.1
+CRNN	lattice	70.5	67.5	22.4	22.5
	50 best			22.4	22.6
+FRNN	lattice	69.8	67.0	22.0	22.3
	50 best			22.2	22.5

The next experiment investigates the performance of RNNLMs trained using various criteria. 50-best rescoring was used. The Fisher and AMI corpora were shuffled separately before being concatenated into single training data file. Shuffling gave a small reduction of WER [4]. The performance of variance regularisation (VR) and NCE trained RNNLMs are shown in Table 3. RNNLMs trained using cross entropy (CE), variance regularisation and NCE respectively were found to give comparable performance. In order to obtain stable convergence, NCE based training required two more epochs than the CE baseline .

Table 3: Performance of RNNLMs trained using various criteria

Train Crit	PPL		WER	
	dev	eval	dev	eval
CE	67.5	63.9	22.1	22.4
VR	68.0	64.4	22.1	22.4
NCE	68.5	65.1	22.1	22.4

Table 4 presents the training and evaluation speed of RNNLMs. Dual Intel Xeon E5-2680 2.5GHz processors with 24 physical cores were used for CPU-based CRNNLM training and evaluation. The Nvidia GeForce GTX TITAN GPU was used for training FRNNLMs. As expected, FRNNLM training on GPU is much faster than CRNNLM training on CPU. NCE training provided further speedup. FRNNLMs trained the VR and NCE criteria were also found to be 2.55 times faster than CRNNLMs.

Table 4: Training and testing speed of RNNLMs

RNN Type	Train Crit	Train(GPU) Speed(kw/s)	Test (CPU) Speed(kw/s)
CRNN	CE	0.45	6.0
FRNN	CE	11.5	0.32
	VR	11.5	15.3
	NCE	20.5	15.3

The training speed heavily depends on the hidden layer size. Table 5 compares the training speed using a varying number of hidden nodes with **RNNLM** and **CUED-RNNLM**. It can be seen that CRNNLMs are efficient when a small sized hidden layer is used. However, the training speed decreases dramatically as the hidden layer becomes larger. When the hidden layer size is increased from 128 to 2048 nodes, the number of words processed per second is decreased by a factor of 340 to 12 words for CRNNLM. In contrast, the training speed of FRNNLMs were found less sensitive to such increase in hidden layer size. This shows the **CUED-RNNLM** toolkit's superior scalability when used to train larger RNNLMs.

6 Acknowledgment

Xie Chen is supported by Toshiba Research Europe Ltd, Cambridge Research Lab and Cambridge Overseas Trust. The author also would like to thank the

[4]No improvements obtained on CRNNLMs using data shuffling.

Table 5: Train Speed (kw/s) against number of hidden nodes

Toolkit	# Hidden node				
	128	256	512	1024	2048
RNNLM	4.1	1.7	0.45	0.095	0.012
CUED-RNNLM	19.8	14.2	11.5	6.6	3.7

work from Xunying Liu on lattice rescoring and Yanmin Qian on building acoustic model and Mark Gales and Phil Woodland for the suggestions about the document.

References

[1] Xie Chen, Xunying Liu, Yanmin Qian, Mark Gales, and Phil Woodland, "CUED-RNNLM – an open-source toolkit for efficient training and evaluation of recurrent neural network language models," in *Submitted to Proc. ICASSP*. IEEE, 2016.

[2] Reinhard Kneser and Hermann Ney, "Improved backing-off for n-gram language modeling," in *Proc. ICASSP*. IEEE, 1995.

[3] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur, "Recurrent neural network based language model.," in *Proc. ISCA Interspeech*, 2010.

[4] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, *Learning representations by back-propagating errors*, MIT Press, Cambridge, MA, USA, 1988.

[5] Tomas Mikolov, Stefan Kombrink, Lukas Burget, J.H. Cernocky, and Sanjeev Khudanpur, "Extensions of recurrent neural network language model," in *Proc. ICASSP*. IEEE, 2011.

[6] Holger Schwenk, "Continuous space language models," *Computer Speech & Language*, vol. 21, no. 3, pp. 492–518, 2007.

[7] Ahmad Emami and Lidia Mangu, "Empirical study of neural network language models for Arabic speech recognition," in *ASRU, IEEE Workshop on*. IEEE, 2007.

[8] Junho Park, Xunying Liu, Mark Gales, and Phil Woodland, "Improved neural network based language modelling and adaptation," in *Proc. ISCA Interspeech*, 2010.

[9] Hai-Son Le, Ilya Oparin, Alexandre Allauzen, J Gauvain, and François Yvon, "Structured output layer neural network language models for speech recognition," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 21, no. 1, pp. 197–206, 2013.

[10] Xunying Liu, Yongqiang Wang, Xie Chen, Mark Gales, and Phil Woodland, "Efficient lattice rescoring using recurrent neural network language models," in *Proc. ICASSP*. IEEE, 2014.

[11] Frederic Morin and Yoshua Bengio, "Hierarchical probabilistic neural network language model," in *Proceedings of the international workshop on artificial intelligence and statistics*, 2005, pp. 246–252.

[12] Xie Chen, Yongqiang Wang, Xunying Liu, Mark Gales, and P. C. Woodland, "Efficient training of recurrent neural network language models using spliced sentence bunch," in *Proc. ISCA Interspeech*, 2014.

[13] Xie Chen, Xunying Liu, Mark Gales, and Phil Woodland, "Improving the training and evaluation efficiency of recurrent neural network language models," in *Proc. ICASSP*, 2015.

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

[15] Anoop Deoras, Tomas Mikolov, Stefan Kombrink, and Kenneth Church, "Approximate inference: A sampling based modeling technique to capture complex dependencies in a language model," *Speech Communication*, vol. 55, no. 1, pp. 162–177, 2013.

[16] Xie Chen, Tian Tan, Xunying Liu, Pierre Lanchantin, Moquan Wan, Gales Mark, and Phil Woodland, "Recurrent neural network language model adaptation for multigenre broadcast speech recognition.," in *Proc. ISCA Interspeech*, 2015.

[17] Tomas Mikolov, Stefan Kombrink, Anoop Deoras, Lukas Burget, and Jan Cernocky, "Recurrent neural network language modeling toolkit," in *ASRU, IEEE Workshop*, 2011.

[18] Jean Carletta et al., "The AMI meeting corpus: A pre-announcement," in *Machine learning for multimodal interaction*, pp. 28–39. Springer, 2006.

[19] Karel Veselỳ, Arnab Ghoshal, Lukás Burget, and Daniel Povey, "Sequence-discriminative training of deep neural networks.," in *Proc. INTERSPEECH*, 2013.