ACCELERATING RECURRENT NEURAL NETWORK TRAINING VIA TWO STAGE CLASSES AND PARALLELIZATION

Zhiheng Huang, Geoffrey Zweig, Michael Levit, Benoit Dumoulin, Barlas Oguz and Shawn Chang*

Speech at Microsoft, Mountain View, CA
*Microsoft Research, Redmond, WA
{zhihuang,gzweig,Michael.Levit,bedumoul,barlaso,Shawn.Chang}@microsoft.com

ABSTRACT

Recurrent neural network (RNN) language models have proven to be successful to lower the perplexity and word error rate in automatic speech recognition (ASR). However, one challenge to adopt RNN language models is due to their heavy computational cost in training. In this paper, we propose two techniques to accelerate RNN training: 1) two stage class RNN and 2) parallel RNN training. In experiments on Microsoft internal short message dictation (SMD) data set, two stage class RNNs and parallel RNNs not only result in equal or lower WERs compared to original RNNs but also accelerate training by 2 and 10 times respectively. It is worth noting that two stage class RNN speedup can also be applied to test stage, which is essential to reduce the latency in real time ASR applications.

***Index Terms*—** language modeling, recurrent neural network (RNN), speed up, parallelization, hierarchical classes

1. INTRODUCTION

Statistical language modeling assigns a probability to a word conditioned on previous context. It plays a vital role in several research fields such as automatic speech recognition, machine translation, text classification and optical character recognition. For a long time smoothed n-gram models [1] have been dominating in language modeling due to their simplicity and accuracy.

There is considerable research on different language modeling techniques. For example, neural network (NN) based language models [2, 3] have proven to be successful among other language modeling techniques. Maximum entropy modeling [4, 5, 6, 7, 8] have gained significant attention due to its flexibility in handling heterogeneous features. Recently recurrent neural network has been reported to obtain significant improvement [9, 10, 11, 12].

Despite the good performance of RNN, it is a challenge to adopt RNN language models due to heavy computational complexity of training RNN models. It can take up to weeks to train an accurate RNN model for a large data set (say, 200M tokens). Unfortunately, it is not easy to speed up RNN training using GPUs as RNN models are typically updated for each training example. In addition, training is slowed down by backpropagation through time (BPTT) [21], which proves to be essential to train accurate RNN models [10]. Thus research effort has been devoted to make these models more computationally efficient. For example, classes have been introduced to RNN infrastructure [10] and joint training of RNN and Maximum Entropy model has been proposed [13] to reduce training time. In this paper, we propose two techniques to accelerate RNN training: 1) two stage class RNN and 2) parallel RNN training. Two stage class RNNs can lead to 2 times speedup and such speedup applies to both training and test stages. Related work includes [14, 15, 16] which applies hierarchical classes scheme to neural network. In the RNN context, [17] recently introduced a speed-based regularization term in the likelihood objective function to balance between the computational efficiency and perplexity performance. However, it used just one layer of classes. In addition to two stage class RNNs, we propose parallel RNN training via data parallelization, which is inspired by the work of [18]. Parallel RNN training can lead to 10 times speedup for RNN model training. We show the efficiency and effectiveness of the proposed speedup techniques on Microsoft internal short message dictation (SMD) data set.

2. RNN MODEL STRUCTURE

Fig. 1 shows the infrastructure of RNN, which includes the Maximum Entropy features as proposed in [13]. The network has an input layer x, hidden layer h and output layer y. We use t to refer to different time stamp. The input vector x at time t, $x(t)$, is formed as follows by concatenating input vocabulary vector $x'(t)$ (index of current word is 1 and all rests are 0) and hidden layer at time $t-1$.

$$\mathbf{x}(t) = [\mathbf{x}'(t)^T \mathbf{h}(t-1)^T]^T. \quad (1)$$

Output vector $y(t)$ contains predicted probabilities of all words in vocabulary. To compute these probabilities, a class component $c(t)$ is introduced as follows to output layer to reduce computational complexity [14, 19, 10].

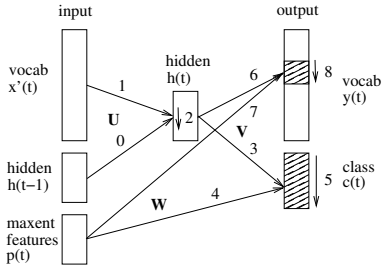

Fig. 1. Recurrent neural network infrastructure

$$P(y(t)|history) = P(c(t)|h(t))P(y(t)|c(t),h(t)). \quad (2)$$

The hidden and output layers are thus computed as follows:

$$h_j(t) = f(\sum_i x_i(t)u_{ji}) \quad (3)$$

$$c_k(t) = g(\sum_m h_m(t)v_{km} + \sum_n p_n(t)w_{kn}) \quad (4)$$

$$y_l(t) = g(\sum_m h_m(t)v_{lm} + \sum_n p_n(t)w_{ln}) \quad (5)$$

where u_{ji} is the network weight between unit i in input layer and unit j in hidden layer, v_{km} is the network weight between unit m in hidden layer and unit k in class component of output layer, $p_n(t)$ is the Maximum Entropy features, w_{kn} is the network weight between maximum entropy feature $p_n(t)$ and unit k in class component of output layer. Similarly, v_{lm} and w_{ln} are counterparts of v_{km} and w_{kn} except that they are for connections to vocabulary component (as opposed to class component) in output layer. We hereafter use $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ to refer to the weight matrices in RNN network. Index j iterates over hidden layer units, index k iterates over classes, and index l iterates over words which belong to the predicted word's class (see hashed output vocab in Fig. 1, which is the reason for speedup using classes as we shall see at the end of this section), and $f(z)$ and $g(z)$ are sigmoid and softmax activation functions. Softmax function in the output layer is used to ensure that the outputs form a valid probability distribution. That is, all output words have probability greater than 0 and their sum is 1.

$$f(z) = \frac{1}{1+e^{-z}} \quad (6)$$

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}} \quad (7)$$

The RNN training algorithm [20] is shown in Algorithm 1. We first initialize epoch number to be 0, the last word to be 0 (corresponding to end of sentence tag </s>), previous log probability to be $-\infty$, learning rate to be 0.1, and learning rate division to be false. We initialize a RNN model by assigning random values between -0.1 and 0.1 to weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$. We then go to an infinite loop until the training finds a local optimal model and breaks (line 21) out of the loop. In each iteration within the loop, we adjust the learning rate on line 9, depending on whether variable *divide* is true of false. Large learning rates result in quick convergence and they are used in the beginning of RNN model training. Small learning rates are used to fine tune RNN models at the end of training. We enumerate all words in training data set and this enumeration is denoted as an epoch. Method *computeNet* computes probability of each word conditioned on previous word (and other history words). Method *learnNet* updates RNN weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$. After each epoch, method *evaluateNet* is called on line 16 to evaluate the current RNN model using a validation data set. If the current model is not significantly better than previous one for the first time, we activate the switch to halve learning rate in next epoch. Otherwise, the algorithm terminates and the latest RNN model is returned.

The key components in Algorithm 1 are methods *computeNet* and *learnNet*. Method *computeNet* computes the probability of a word conditioned on previous words. It consists of 9 steps as shown in Fig. 1. Step 0 updates current hidden layer activation vector by multiplying hidden layer activation vector in previous time and weight matrix (part of $\mathbf{U}$) between them. Step 1 updates hidden layer vector by multiplying input vocabulary vector (with index of *lastWord* being 1 and all others being 0) and the weight matrix (part of $\mathbf{U}$) between vocabulary component in input layer and hidden layer. Step 2 applies sigmoid function to hidden layer. Therefore the combination of step 0, 1, and 2 corresponds to equation (3). Similarly, the combination of step 3, 4, and 5 corresponds to equation (4), and the combination of step 6, 7, and 8 corresponds to equation (5).

Algorithm 1 RNN training algorithm

```
1:  epoch = 0
2:  lastWord = 0
3:  preLogp = −∞
4:  learnRate = 0.1
5:  divide = false
6:  initializeModel()
7:  while true do
8:      if divide == true then
9:          learnRate = learnRate/2
10:     end if
11:     for each word in training data do
12:         computeNet(lastWord, word) //forward pass
13:         learnNet(lastWord, word, learnRate) //backward pass
14:         lastWord = word
15:     end for
16:     logp = evaluateNet()
17:     if logp × minImprovement < preLogp then
18:         if divide == false then
19:             divide = true
20:         else
21:             break
22:         end if
23:     end if
24:     preLogp = logp
25:     epoch++
26: end while
```

The method *computeNet* is a *forward pass* which propagates from left to right. On the other hand, method *learnNet* is a *backward process* in which the weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ are updated to better fit the data. Similar 9 steps are carried out to update the weight matrices with a reverse order (from step 8 to 0) and reverse direction (from right to left). For example, the following is used to update the weights $\mathbf{V}(t)$ between hidden layer and output layer at time t

$$\mathbf{V}(t) = (1-\beta)\mathbf{V}(t-1) + \alpha \mathbf{e}_0(t-1)^T \mathbf{H}(t-1) \quad (8)$$

where β is the regularization term, α is the learning rate, $\mathbf{e}_0$ is the prediction error vector in output layer, and $\mathbf{H}$ is the hidden activation vector. It is worth noting that in our experiments, backpropagation through time (BPTT) [21] is used to update $\mathbf{U}$ as it leads to better accuracy [10]. The error is propagated through recurrent connections back in time for a specific number of times steps.

In order to derive RNN training complexity, we assume *frequency binning* [10] in which word probability mass is equally distributed on classes. That is, if we choose 20 classes, words that correspond to the first 5% of the probability mass would be mapped to class 1, the words that correspond to the next 5% of the probability mass would be mapped to class 2, etc. Based on this assumption, the average computational complexity of training a RNN model as shown in Fig 1 is of

$$O(I \times W \times (H \times H + H \times (C + \frac{V}{C}))), \quad (9)$$

where I is the number of training epochs (usually 10 to 20, around 13 in our experiments in Section 5) before convergence is achieved, W is the number of tokens in training

set, H is the size of hidden layer, V is the size of vocabulary, and C is the size of classes. Note that for simplicity we ignore the small constant complexity $O(1 \times H)$ introduced by the connection between input word and hidden layer, and $O((N-1) \times (C + \frac{V}{C}))$ by N-gram based Maximum Entropy features in factor $O(H \times H + H \times (C + \frac{V}{C}))$ at Equation 9. If we choose

$$C = \sqrt{V}, \tag{10}$$

we obtain the following average computational complexity

$$O(I \times W \times (H \times H + H \times \sqrt{V})), \tag{11}$$

The original RNN model [9] without the use of classes has computational complexity of

$$O(I \times W \times (H \times H + H \times V)). \tag{12}$$

Usually V is in the order of $100K$ and thus the class technique significantly reduces the complexity factor of $O(V)$ to $O(\sqrt{V})$.

3. TWO STAGE CLASS RNN MODEL

It has been shown in [10] that the adoption of classes can lead to 15 times speedup for both RNN training and test phases. It is natural to extend the classes to hierarchical classes [14, 15, 16, 19]. In this paper, we extend the classes to two stage classes and we denote them as *super class* and *class* respectively. Fig. 2 shows the modified RNN mode infrastructure, in which super class component is introduced to output layer. The algorithm to train a two stage class RNN remains the same as Algorithm 1, except that we now have three extra steps (3', 4', and 5') besides the 9 steps in Fig. 1. These three

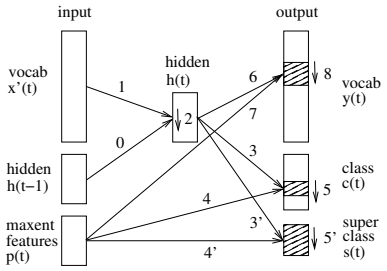

Fig. 2. Recurrent neural network structure

extra steps compute the probabilities of super classes and they correspond to the following equation in *computeNet*

$$s_k(t) = g(\sum_m h_m(t) v_{km} + \sum_n p_n(t) w_{kn}) \tag{13}$$

where s_k is the super class and index k iterates over all super classes. Equations (3) to (5) remain the same except that Equation (4) does not need to iterate over all classes. Instead, it only iterates over classes which belong to the current word's super class. Therefore, the computational complexity for two stage class RNN becomes

$$O(I \times W \times (H \times H + H \times (S + \frac{C}{S} + \frac{V}{C}))), \tag{14}$$

where S is the super class size. If we choose

$$S = \sqrt[3]{V}, \tag{15}$$

$$C = \sqrt[3]{V^2}, \tag{16}$$

the training complexity becomes

$$O(I \times W \times (H \times H + H \times \sqrt[3]{V})). \tag{17}$$

We can in general derive n stage classes ($n > 2$), with the complexity reduced to

$$O(I \times W \times (H \times H + H \times \sqrt[n+1]{V})), \tag{18}$$

where the complexity factor of $O(\sqrt{V})$ is reduced to $O(\sqrt[n+1]{V})$. It is worth noting that the overall complexity also has a factor $O(H^2)$, thus the speedup may not be strictly from $O(\sqrt{V})$ to $O(\sqrt[n+1]{V})$.

In order to facilitate two stage classes, we need to specify the number of super classes and classes. We can use the frequency binning [10] as assumed in complexity analysis to derive classes. However, as suggested by Povey [20], taking the square root of the frequencies prior to frequency binning can result in faster training. We thus adopt this *square root frequency binning* as baseline in our experiments in Section 5. Note that square root frequency binning differs from the speed-optimal classing of [17] in ignoring exact word frequencies, but is a reasonable approximation in terms of training speed in practice.

We can apply the same strategy to derive super classes by treating classes as words. We can alternatively divide the classes to a group of super classes, each of which consists of the same number of classes. We tried both approaches in experiments in Section 5 and they resulted in similar perplexities. We thus only report the results with classes being evenly divided to a group of super classes.

4. PARALLEL RNN TRAINING

One intuitive way to speed up RNN training is via data parallelization, which is inspired by the work in [18] to speed up deep neural network (DNN) training. Algorithms 2 and 3 show the proposed master and slave RNN training algorithms respectively, which may run on different machines.

Master algorithm is responsible for slave jobs dispatching and master model update. In particular, an initial RNN model is first generated on line 5 in Algorithm 2. It could be initialized by assigning to random weights as in Algorithm 1. However, we find that *warm start* strategy can result in faster convergence in training. That is, we train a RNN model using entire training data for one epoch and use such trained RNN model as an initial model. The initial RNN model is evaluated using a validation data set on line 6. We then go to an infinite loop until the training finds a local optimal model and break out the loop (line 27). For each iteration within the loop, we partition the entire training data to batches[1], with possible overlapping between batches. We found that the overlapping is helpful for training convergence in our experiments. For example, we can have 50 batches and each batch takes 5% of entire training data.

Each data batch is dispatched to a slave node (line 13) to train a new slave RNN model based on previous master RNN

[1]Partition may differ for different iterations.

model (preM) (see Algorithm 3). Once all slave RNN models are available (line 16), we evaluate each updated slave model by comparing its log likelihood ($\log s$) on a validation data set with preM's log likelihood ($\log p$). If $\log s$ is significantly worse than $\log p$, that is, $\log s < t \times \log p$, where $t = 1.1$ is the tolerance parameter, such a slave model is rejected to compute average model (averagedM) on line 17, in which all RNN network weights $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ of slave models are averaged component wise. We found in experiments that the rejection rate is low. For example, we may have 0 or 1 slave models rejected out of 50 batches. However, the rejection is essential to ensure the training convergence; The master model update (*modelUpdate*) using a rejected bad model may result in a new master model with much higher perplexity than the old master model. We then use the preM and averagedM to generate a new master model (m). Method *evaluateNet* is called on line 22 to evaluate the new master RNN model. If it is not significantly better than previous one for the first time, we activate the switch to halve learning rate in next epoch. Otherwise, the algorithm terminates and the latest RNN model is returned.

Algorithm 2 Master RNN training algorithm

```
1:  epoch = 0
2:  preLogp = −∞
3:  learnRate = 0.1
4:  divide = false
5:  RNN preM = initializeModel()
6:  preLogp = evaluateNet(preM)
7:  while true do
8:      if divide == true then
9:          learnRate = learnRate/2
10:     end if
11:     partition training data into batches
12:     for each batch b_i do
13:         call slave(preM, b_i, learnRate) to train a slave RNN model
14:     end for
15:     while true do
16:         if all slave training finished then
17:             averagedM = averaged slave models
18:             m = modelUpdate(preM, averagedM)
19:             break
20:         end if
21:     end while
22:     logp = evaluateNet(m)
23:     if logp × minImprovement < preLogp then
24:         if divide == false then
25:             divide = true
26:         else
27:             break
28:         end if
29:     end if
30:     preM = m
31:     preLogp = logp
32:     epoch++
33: end while
```

Algorithm 3 Slave RNN training algorithm

1: Train one epoch on model $preM$ using batch data b_i with learning rate *learnRate* and return the updated model

Method *modelUpdate* creates a new master model based on previous master model *preM* and averaged slave model *averagedM*. The model weights update for $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are similar. We take model weights $\mathbf{V}$ between hidden layer and output layer as example and the updating equation is

$$\mathbf{V}(t) = (1-\beta)\mathbf{V}(t-1) + \alpha(\mathbf{V}_a - \mathbf{V}(t-1)), \qquad (19)$$

Where $\mathbf{V}(t)$ and $\mathbf{V}(t-1)$ are weight matrices in epoch t and $t-1$ respectively. $\mathbf{V}_a$ is weight matrix of averaged slave model, α is master learning rate (0.1 in our experiments in Section 5), and β is master regularization parameter. We found that the β in range of [1e-7,1e-5] leads to similar results. Fig. 3 shows the master model update pictorially.

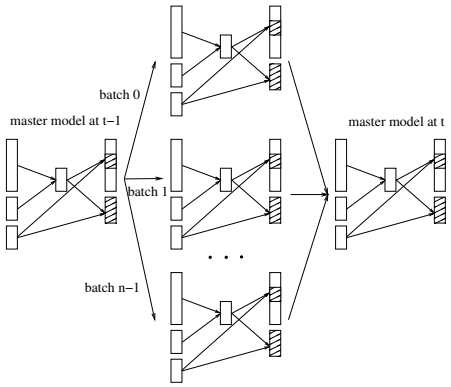

Fig. 3. Master RNN model update

Parallel RNN training reduces complexity from Equation (9) to

$$O(I \times W' \times (H \times H + H \times (C + \frac{V}{C}))) \qquad (20)$$

where W' is the batch training data size. The overhead is the data partition time and master model updating time, which is negligible compared to the overall training time. If each batch contains 5% of entire training data and the training epochs are the same (which is usually the case in experiments in Section 5), the speedup factor is 20. In practice, we may get less speedup due to the warm start strategy.

5. EXPERIMENTS

We train RNN models using open source RNN package [20] which includes speedup tricks of square root frequency binning [10, 17] and Maximum Entropy features [13]. This package has been reasonably optimized for training speed. We implemented two stage RNN and parallel RNN training on top of the open package for the experiments as described below.

We use Microsoft internal short message dictation data set to test two stage class RNN and parallel RNN training. Table 1 shows number of sentences and tokens for training, validation, and test data sets. Training data consist of a collection of transcribed utterances for daily short message dictation. Example utterances are *I am on my way to San Francisco* or *Can I come to your house later*. Validation and test data are transcribed utterances which are randomly sampled in two different months. Validation set is used to early stop RNN training.

Table 1. Stats of training, validation and test data sets.

data	sentences	tokens
train	7M	56M
valid	6K	48K
test	2K	15K

The baseline language model is a KN4 n-gram model trained on entire training data set. This baseline language

model is used with a Microsoft production quality acoustic model to produce n-best list in the first pass decoding for both validation and test utterances. In doing so, we obtain the baseline perplexities (PPLX) and word error rates (WER).

We conduct the second pass n-best re-scoring using various trained RNN models, including original RNN, two-stage class RNNs, and parallel RNNs. For all RNN models, we set maximum entropy feature size to be 100M and maximum entropy based n-gram feature order to be 3. We vary hidden layer size to be 25, 50, 100 and 200 to see how they affect the performance. For each hidden layer size setting, we have two settings for two stage class RNNs: 40x1200 and 40x1600, with the first number being the number of super classes and the second being the number of classes. For example, 40x1200 represents the number of super classes being 40 and the number of classes being 1200. We also have two settings for parallel RNN training: 25x0.1 and 50x0.05, with the first number being the number of batches, and the second being the batch size (percentage of entire training data size).

Fig. 4 shows the perplexities of validation data with respect to elapsed training time for various RNNs with hidden layer size of 100. Each data point in the plot represents the perplexity evaluated on completion of one epoch. All five RNNs training asymptotically converges to similar perplexity on validation data set, with original RNN resulting in slightly lower perplexity. Two stage class RNNs and original RNN have a similar convergence pattern in the sense that they all have a sudden drop of perplexities during epoch 6. This is due to the fact that learning rate is halved from 0.1 to 0.05. However, two stage RNNs take less time (40 hours) than original RNN (75 hours) to converge. Parallel RNNs have identical perplexity as RNN after first epoch due to the warm start strategy. However, they have a different converge pattern to RNN; They results in quick convergence during the second epoch. The learning rate is halved from 0.1 to 0.05 during the 9-th and 10-th epochs for parallel RNN (25x0.1) and (50x0.05) respectively. More importantly, RNN (25x0.1) and (50x0.05) only take 13 and 9 hours respectively to converge.

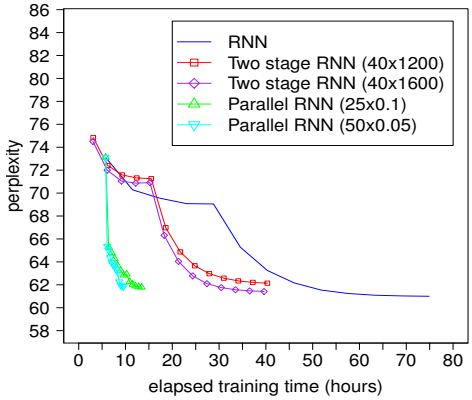

Fig. 4. Perplexity of validation data set in function of elapsed training time for various RNNs training with 100 hidden layer size

In n-best list re-scoring, the RNN model re-estimates a log-probability score for each n-best hypothesis s:

$$\log L(s) = n \times p + \sum_{i=1}^{n} a_i + w \sum_{i=1}^{n} \log P(w_i|h_i), \quad (21)$$

where n is the number of words, p is the word insertion penalty, a_i is the acoustic model score for word w_i, h_i is the history $w_1 \dots w_{i-1}$ and w is he language model scale. P is the combined probability estimate of KN 4-gram and RNN models, which is obtained by linear interpolation:

$$P(w_i|h_i) = \lambda P_{rnn}(w_i|h_i) + (1-\lambda) P_{kn}(w_i|h_i), \quad (22)$$

where λ is the weight for RNN model (we fix $\lambda = 0.7$ in our experiments), P_{rnn} is the RNN model probability and P_{kn} is KN4 n-gram language model probability.

Table 2 shows the training speed (tokens per second), training time of various RNNs, perplexities and word error rates of interpolated modes (KN4 + RNN) for validation and test data sets respectively. It is a general trend that larger hidden layer size RNNs result in lower perplexities and WERs (see Fig. 5). For example, the 50 hidden layer size RNN results in perplexity of 75.32 and WER of 24.37 for test data set, while the 100 hidden layer size RNN results in 73.21 and 23.76 respectively. On the other hand as the hidden layer size increases, the training of RNN requires longer training time due to the quadratic complexity term of $O(H^2)$. For example, the training of hidden layer size 100 RNN takes 3.1 days, while the training of hidden size 200 RNN takes 9.4 days.

The lowest PPLXs or WERs of different RNNs for various hidden layer sizes are highlighted in bold in Table 2. As can been seen, the original RNNs always result in lowest PPLXs for both validation and test data sets, with two stage class RNNs and parallel RNNs leading to slightly higher PPLXs. In terms of WER metric, it is interesting to see that two stage RNNs can lead to lower WERs than RNNs on test data set for hidden layer sizes of 50, 100, and 200. For example, two stage class RNN (40x1600) results in a WER of 23.64 while RNN results in a WER of 23.70 for test data with hidden layer size of 200. Parallel RNNs can lead to lower WERs than RNNs on validation data set for hidden layer sizes of 25 and 50. It is surprising that two stage RNN (40x1200) results in the lowest WER (23.59) on test data set with hidden layer size 100, possibly due to noise. RNN, two stage RNN (40x1600) and parallel RNN (50x0.05) lead to lower WERs from hidden layer size 100 to 200.

In terms of training speed, both two stage class RNN and parallel RNN can speed up training compared to RNN training. For example with hidden layer size 100, RNN has training speed of 2.7K tokens per second, while two stage class RNN (40x1600) has training speed of 5.1K tokens per second. It is worth noting the speedup also applies to test stage. Parallel RNN training can result in more significant speedup than two stage class RNN training.[2] For example with hidden layer size 200, the parallel RNN (50x0.05) can obtain 20 times speedup, due to the fact that each data batch is 20 times smaller than the entire training data. In practice, we may get less speedup due to the warm start strategy. Assuming a RNN model training needs 13 epochs[3] and the total training time is denoted as T. RNN (50x0.05) takes training time of $0.13T = T/13 + T/20$ only, with the first term accounting for the warm start (one epoch trained on the entire training data) and the second term accounting for parallelization speedup. Similarly, RNN (25x0.1) takes training time of $0.18T = T/13 + T/10$ only. Finally, we note that parallel RNN training can result in even greater speedup if batch size is smaller. In our experiments, we obtained similar good performance for 25 and 50 batches. For a given training data, it

[2] Although such speedup does not apply to test stage.

[3] It is usually the case in our experiments.

Table 2. Training speed, training time, perplexity and word error rate of single RNN, parallel RNN and two stage class RNN training

hidden size	model	setting	speed (w/s)	train time (days)	valid		test	
					PPLX	WER	PPLX	WER
N/A	baseline	N/A	N/A	N/A	79.99	26.02	83.37	24.51
25	RNN	N/A	12K	0.7	**75.63**	25.87	**78.64**	24.35
	two stage class RNN	40x1200	29K	0.3	76.54	26.02	79.49	24.38
	two stage class RNN	40x1600	28K	0.3	75.76	25.97	78.94	24.42
	parallel RNN	25x0.1	12K	0.1	76.24	**25.83**	79.25	**24.23**
	parallel RNN	50x0.05	12K	0.1	76.52	25.84	79.51	24.40
50	RNN	N/A	6.1K	1.4	**72.39**	25.50	**75.32**	24.37
	two stage class RNN	40x1200	15K	0.6	73.08	25.69	75.86	**23.95**
	two stage class RNN	40x1600	12K	0.7	72.83	25.79	75.62	24.05
	parallel RNN	25x0.1	6.1K	0.2	73.06	**25.49**	75.99	24.30
	parallel RNN	50x0.05	6.1K	0.2	72.80	25.58	75.73	24.25
100	RNN	N/A	2.7K	3.1	**70.47**	**25.34**	**73.21**	23.76
	two stage class RNN	40x1200	5.0K	1.7	71.31	25.50	73.98	**23.59**
	two stage class RNN	40x1600	5.1K	1.7	70.82	25.65	73.55	23.81
	parallel RNN	25x0.1	2.7K	0.6	71.16	25.45	73.85	23.78
	parallel RNN	50x0.05	2.7K	0.4	71.28	25.38	73.95	23.83
200	RNN	N/A	0.9K	9.4	**68.74**	**25.16**	**71.26**	23.70
	two stage class RNN	40x1200	1.5K	5.6	69.58	25.32	72.37	23.78
	two stage class RNN	40x1600	1.6K	5.3	69.24	25.42	71.89	**23.64**
	parallel RNN	25x0.1	0.9K	1.7	70.50	25.29	73.18	23.88
	parallel RNN	50x0.05	0.9K	1.2	69.95	25.25	72.68	23.76

would be interesting to find optimal number of batches which balance model performance and training time.

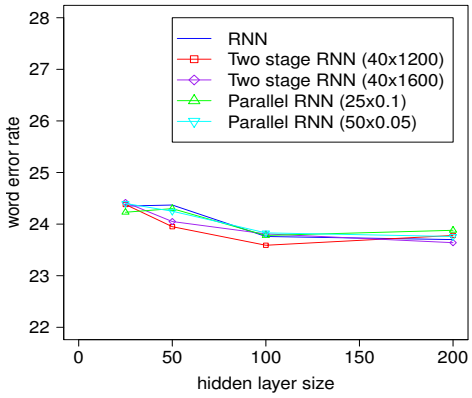

Fig. 5. Word error rate of test data set in function of hidden layer size for various RNN training

6. CONCLUSION

In this paper, we proposed two techniques to accelerate RNN training: 1) two stage class RNN training and 2) parallel RNN training. We demonstrated these two techniques on Microsoft internal short message dictation (SMD) data set. Two stage class RNNs and parallel RNNs not only result in equal or lower WERs compared to original RNNs but also accelerate training by 2 and 10 times respectively. Finally, it is worth noting that two stage class RNNs can also be used to accelerate test stage, which is essential to reduce the latency in real time ASR applications.

7. REFERENCES

[1] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *ACL*, 1996, pp. 310–318.

[2] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, T. Hoffman, T. Poggio, and J. Shawe-taylor, "A neural probabilistic language model," *Journal of Machine Learning Research*, , no. 3, 2003.

[3] H. K. J. Kuo, E. Arisoy, A. Emami, and P. Vozila, "Large scale hierarchical neural network language models," in *INTERSPEECH*, 2012.

[4] R. Rosenfeld, *Adaptive Statistical Language Modeling: A Maximum Entropy Approach*, Ph.D. thesis, Carnegie Mellon University, 1994.

[5] S. F. Chen, "Performance prediction for exponential lanuage models," in *HLT-NAACL*, 2009, pp. 450–458.

[6] T. Alumae and M. Kurimo, "Efficient estimation of maximum entropy language models with n-gram features: an srilm extension," in *INTERSPEECH*, 2010.

[7] P. Xu, S. Khudanpur, and A. Gunawardana, "Randomized maximum entropy language models," in *IEEE ASRU*, 2011.

[8] G. Zweig and S. Chang, "Personalizing model m for voice-search," in *INTERSPEECH*, 2011.

[9] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010.

[10] T. Mikolov, S. Kombrink, L. Burget, J. H. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *ICASSP*, 2011.

[11] S. Kombrink, T. Mikolov, M. Karafiat, and L. Burget, "Recurrent neural network based language modeling in meeting recognition," in *INTERSPEECH*, 2011.

[12] Y. Shi, P. Wiggers, and C. M. Jonker, "Towards recurrent neural networks language models with linguistic and contextual features," in *INTERSPEECH*, 2012.

[13] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. H. Cernocky, "Strategies for training large scale neural network languge models," in *ASRU*, 2011.

[14] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model," in *AISTATS*, 2005.

[15] A. Mnih and G. E. Hinton, "A scalable hierarchical distributed languaged model," in *NIPS*, 2009.

[16] H. S. Le and I. Oparin and A. Allauzen and J. L. Gauvain and F. Yvon, "Structured output layer neural network language model," *ICCASP*, 2011.

[17] G. Zweig and K. Makarychev, "Speed regularization and optimality in word classing," in *ICASSP*, 2013.

[18] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[19] J. Goodman, "Classes for fast maximum entropy training," in *ICASSP*, 2001.

[20] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky, "Rnnlm - recurrent neural network language modeling toolkit," in *ASRU Demo Session*, 2011.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams., "Learning internal representations by back-propagating errors," in *Nature*, 1986, pp. 533–536.