Distributed Training of Neural Network Language Models

CS244B Project Report, Autumn 2014

Kai Sheng Tai, Shawn Xu
kst@cs.stanford.edu, jinhuaxu@google.com

December 5, 2014

1 Introduction

The purpose of *statistical language models* is to estimate the joint probability distribution over sequences of words in a language. Typically, the modelled distribution is the conditional probability $P(w_i|w_{i-k},\ldots,w_{i-1})$ of a word w_i given its preceding context $\{w_{i-k},\ldots,w_{i-1}\}$ for some window of size k. Language models are used in a wide range of applications involving natural language such as statistical machine translation, automatic speech recognition, spelling correction, parsing, and optical character recognition. *Neural network language models* (NNLMs) [1] are a subclass of statistical language models that model $P(w_i|w_{i-k},\ldots,w_{i-1})$ using a neural network. NNLMs have recently become an increasingly popular class of models in the machine learning and natural language processing communities.

In this project, we implemented a distributed system for NNLM training. The model is trained by maximizing the likelihood of the training data with optimization performed using stochastic gradient descent (SGD). For each input (a length-k sequence of words), we compute the loss with respect to the target word (the word immediately following the context). This loss is *backpropagated* through the network to obtain the gradient of the objective function with respect to the parameters of the model. As with other statistical language models, NNLMs are trained on large text corpora of the order of hundreds of millions or billions of words [2].

The architecture of our system is based on that of existing systems for distributed training of large-scale neural networks [4, 3]. In [4], computation is distributed over clusters of up to ~ 5000 machines, while in [3], extremely large networks upwards of 2 billion parameters in size are trained using a cluster of 120 nodes. For this project, we train much smaller networks of about 2 million parameters.

2 Model Description

We train the NNLM illustrated in Fig. 1. The model takes as input a context window of size 3. Each word in the context is represented by a 50-dimensional vector of real numbers. The concatenated 150-dimensional input vector is mapped to a 100-dimensional hidden layer. An elementwise tanh nonlinearity is applied to the hidden layer, which is then mapped to the output layer of dimension equal to the size of the vocabulary. In our experiments, we use a vocabulary of size 40000. The output layer is normalized using the softmax function such that it represents a probability distribution over the vocabulary. The parameters of the model are the 40000 word vectors $v_k \in \mathbb{R}^{50}$,

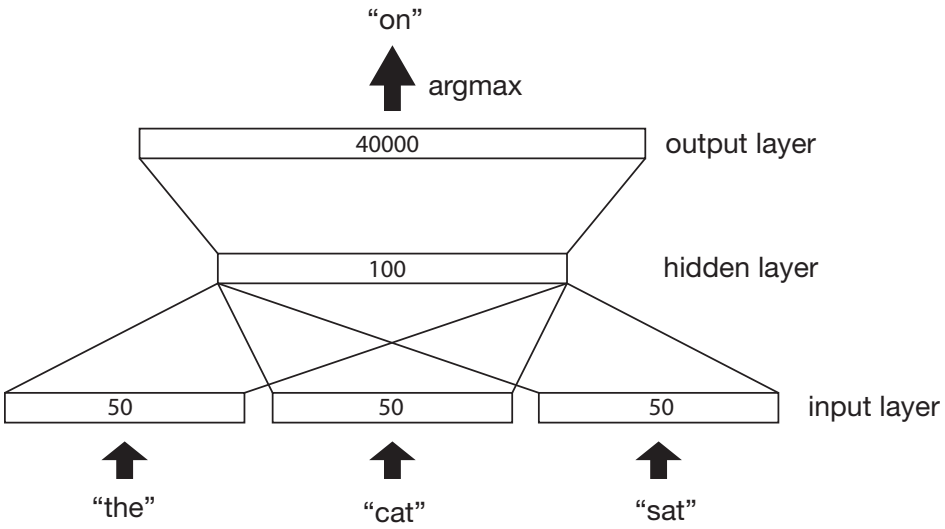

Figure 1: Our neural network language model (NNLM) architecture. Numbers denote the dimensionality of each vector.

the input-hidden matrix $W_{ih} \in \mathbb{R}^{100\times150}$, the hidden-output matrix $W_{ho} \in \mathbb{R}^{40000\times100}$, and the bias vectors $b_h \in \mathbb{R}^{100}$ and $b_o \in \mathbb{R}^{40000}$.

3 System Design

3.1 Architecture

Parameter update computations are distributed over several *worker nodes*. The training data is sharded and distributed over the worker nodes, each of which operates independently on its assigned shard(s). The workers are coordinated by a single *parameter server*. The parameter server manages the assignment of shards to workers and stores a master copy of the model parameters. This architecture is illustrated in Fig. 2.

Our implementation is available at `https://github.com/kaishengtai/distrust`.

3.2 Parameter Server

The parameter server handles initialization of the language model, replication of the parameters, dynamic sharding of training data to a variable number of worker nodes, processing of parameter queries and updates from worker nodes, and heartbeat monitoring of worker nodes. The parameter server exposes the following API:

```
<ModelInfo m, Params p, list<string> shard_paths> announce(<ip, port> worker_addr)
void push_update(ParamUpdate update)
Params pull_params()
```

The parameter server stores a mapping from each worker to its set of shards. A worker announces its presence to the parameter server via the `announce()` RPC. Upon receipt, the server dynamically reshards the training data by assigning shards not yet started to the new worker and updates the internal worker-to-shard mapping. The parameter server also returns the current language model along with the shard assignment.

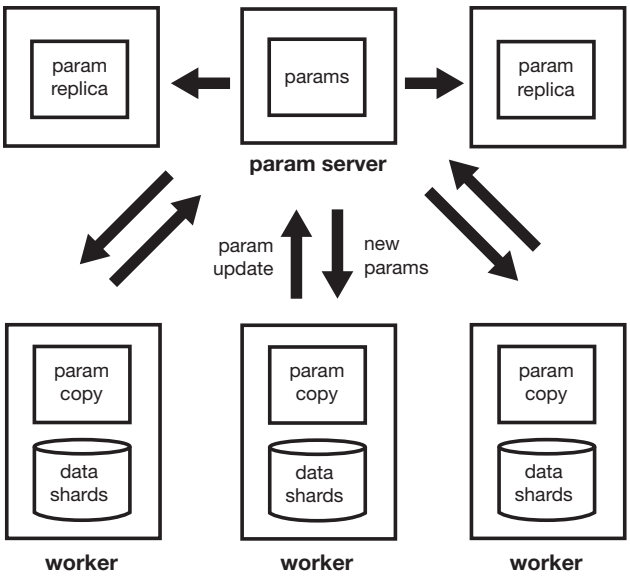

Figure 2: The architecture of the distributed NNLM training system.

To monitor and handle worker failure, the parameter server spawns a heartbeat monitoring thread for each worker added. The thread periodically pings the worker via an open TCP connection and watches for network or worker exceptions. On the event of such failures, the parameter server kicks off a reshard event, where the remaining shards are redistributed to alive workers. Worker failure handling is described in more detail in Section 4.2.

The workers periodically pull the most up-to-date parameters via the `pull_params()` RPC, and send parameter updates via the `push_update(ParamUpdate update)` RPC. The parameter server asynchronously handles each RPC, and protects against concurrent access of the parameters using read/write locks.

The size of each parameter update from the workers is large (on the order of 100MB), therefore we chose to keep an in-memory copy of the language model datastructure to maintain high throughput. Upon receiving the `ParamUpdate` RPC request, the server acquires a write lock on the language model and calculates the updated model based on the deltas, and modifies the in-memory datastructure.

3.3 Worker

Each worker handles the computation of gradient updates on a subset of the training data. This gradient computation is performed using the worker's local replica of the language model parameters. This replica is periodically refreshed with up-to-date parameters via a request to the parameter server, as described in the previous section.

The worker exposes the following RPC interface:

```
<list<string> completed_shards> heartbeat()
void start(<list<string> shard_paths, double learn_rate, int batch_size>)
void stop()
void reassign(list<string> shard_paths)
```

The `batch_size` parameter controls the number of training samples the worker processes before pushing an update to the parameter server and pulling new parameters.

We compute an adaptive per-parameter learning rate using AdaGrad [5]. When using stochastic gradient descent with a fixed learning rate and multiple workers, we find that our loss on the validation set quickly diverges during training. We find that AdaGrad's adaptive learning rates provide a stabilizing effect during training.

4 Fault Tolerance and Recovery

4.1 Parameter Server Failure

The in-memory language model parameters are replicated on two replica nodes using Raft [6]. Parameter writing to the replicated state machine is handled by an asynchronous backup thread. In the event of a server failure, this backup will typically only be stale by a few seconds of training time; training can then be resumed from this state without needing to start over from scratch.

Our implementation can be improved with automatic failover to a parameter server replica. This can be achieved by letting the parameter server replica corresponding to the newly-elected Raft leader take over as the primary parameter server. The language model and most recent parameters are populated from Raft into the in-memory datastructures of the new primary. Existing workers failover to this parameter server, and `announce()` themselves to the server. The parameter server does not replicate the worker configurations or worker to shard mapping, because this can be easily reconstructed upon the `announce()` RPC.

4.2 Worker Failure and Resharding

As mentioned previously, the parameter server monitors the health of all workers via `heartbeat()` RPCs. If the heartbeat times out or a network exception occurs, the parameter server removes the worker from the worker pool and triggers shard reassignment.

The parameter server maintains a set of completed shards to eliminate duplicate work upon shard reassignment, in the event of a worker join or failure. The set of completed shards per worker is piggybacked on the worker `heartbeat()` response RPC. In the event of a reshard, the parameter server uses a different algorithm depending on whether a new worker is added or an existing worker failed.

If a new worker is added, the server minimizes duplication of shard computation by popping from the end of existing workers' shard queues. The parameter server then adds the non-started shards to the new worker's shard queue until an uniform distribution is reached.

If an existing worker failed, the server takes the unfinished shards of the failing worker and uniformly distributes it to the remaining workers. This redistribution may cause some of the data to be duplicated during training; our models are resilient to such potentially-repeated samples.

After updating its internal worker to shard mapping, the parameter server notifies all workers of their new shard assignments via the `reassign(list<string> shard_paths)` RPC. In the case of a new worker join, the shard assignments are returned via the `announce()` response RPC.

In addition to providing robustness in the event of worker failure, our resharding capability also allows the operator to assign more computational resources to the training job as required, or as more machines become available during training.

5 Evaluation

We evaluate our system by training the language model on a subset of the 1-Billion Word Language Modeling benchmark [2]. During training, we compute the log-perplexity loss of the model on a

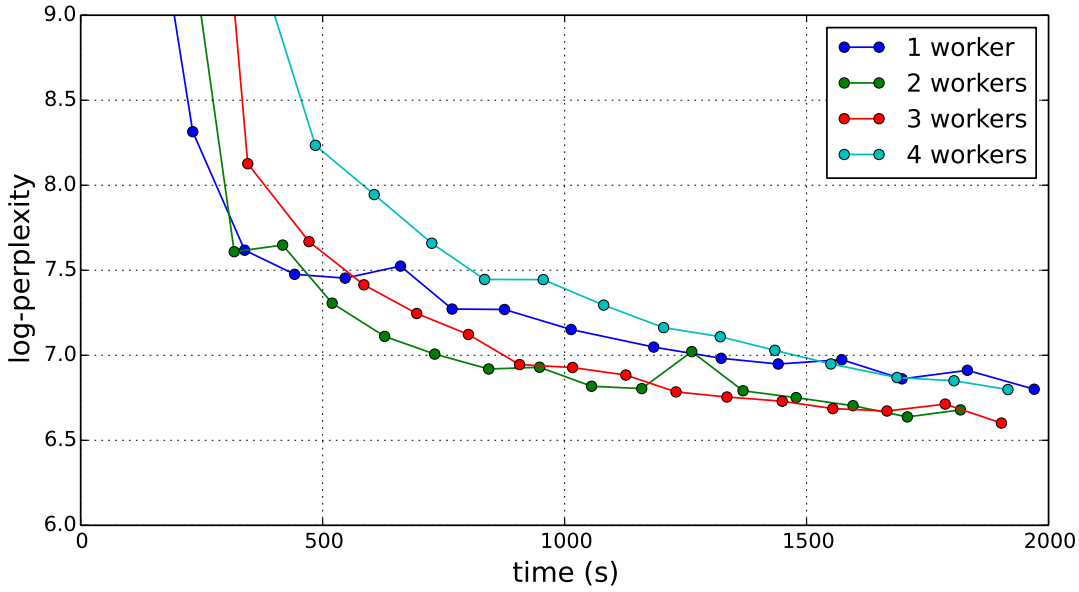

Figure 3: Loss on validation set over time with different numbers of worker nodes. Informally, the log-perplexity metric is a measure of how well the NNLM predicts correct words given the preceding context (lower is better).

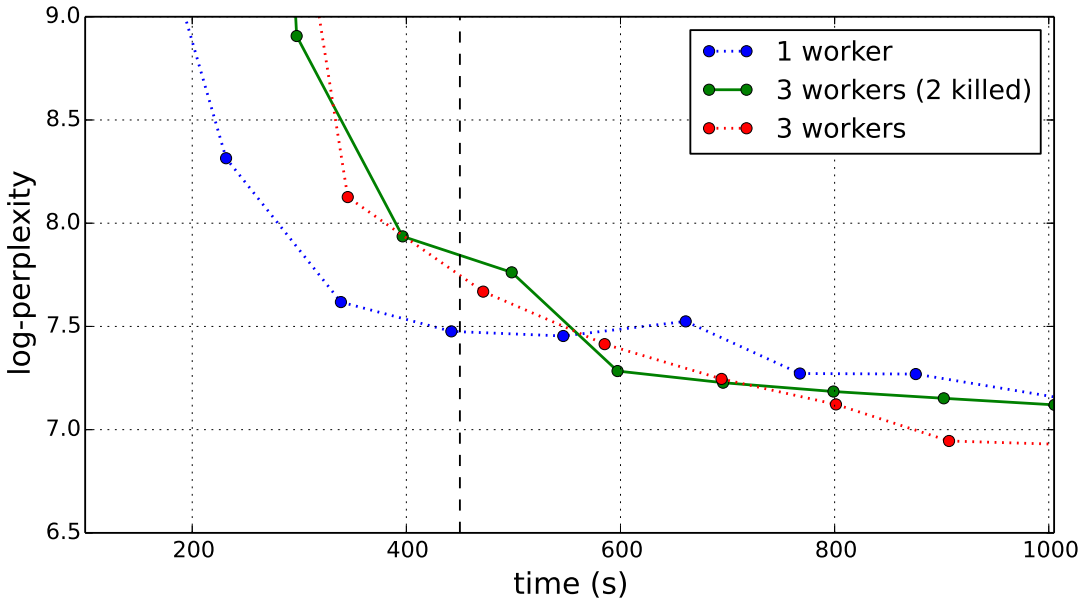

Figure 4: Loss on validation set when workers are killed. Two workers out of a three-worker cluster are killed at $t = 450$ (dotted line in the figure). For comparison, loss curves for clusters with no workers killed are plotted.

held-out validation set of 500 sentences. The log-perplexity L with respect to the parameters θ is defined as:

$$L(\theta) = \frac{1}{N} \sum_{i=k}^{N} \log P(w_i | w_{i-k}, \ldots, w_{i-1}; \theta), \tag{1}$$

where $k = 3$ is the window size and N is the number of words in the dataset. We evaluated our system on the `corn` cluster.

Though each worker computes updates based on a potentially stale copy of the language model parameters, we find that the system still manages to make progress as measured by the perplexity on the validation set. In Fig. 3, we see that using 2 or 3 worker nodes give similarly improved performance over a single worker when training for up to 2000 seconds. No improvement is seen with 4 workers over this training duration; this indicates that the benefit of additional parallelization over the training data is outweighed by the increased stochasticity of the gradient updates.

The effect of worker failure on training is shown in Fig. 4. Here, two out of three workers in a cluster are killed. The parameter server detects the worker failure and reassigns all shards to the remaining worker. The loss curve initially tracks that of the 3-worker cluster, and after the two workers are killed, the slope flattens out to match that of the 1-worker cluster.

6 Conclusions

In this project, we have implemented a system for the distributed training of a neural network language model. It is a simplified version of several existing systems for distributed neural network training. Due to high network I/O demands of this architecture, it is best suited to high-bandwidth, low-latency datacenter environments. In our experiments, we show that the system successfully parallelizes training among several worker nodes.

References

[1] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer, 2006.

[2] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[3] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX Association.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[5] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[6] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm.