Parameter and Inference Server

Hugh Chen and Ayse Dincer

December 2018

1 Introduction

Deep learning models continue to produce state-of-the-art results for a broad range of applications from natural language processing to computer vision to bioinformatics. However, deep learning often requires at least thousands of parameters (e.g., ResNet which has millions of parameters (He et al., 2016)) trained on at least thousands of samples (e.g., ImageNet which has millions of images (Deng et al., 2009)) which might make training and inference difficult and time-consuming. Therefore, the typical approach is to utilize batch gradient descent which subsamples the training data to approximate the gradient for each update step. One key observation is that the update to the network weights is exactly a summation over the individual sample gradients. Which suggests that it is possible to split gradient calculations across different machines. To this end, we have designed a *Parameter and Inference Server* which trains neural networks parallelized across many nodes. Our server offers a system to train models and perform inference. Users can connect to train their models efficiently and reliably. Our system computes batch gradient descent on random samples with replacement in parallel across different nodes. This approach gives us 2 main advantages compared to training the neural network model in each node sequentially: (1) training can be done in parallel (along with inference) (2) since the data is provided in batches to different nodes, it is possible to train from a random set of samples with replacement (rather than without replacement), which might make the gradient updates more robust.

2 Related Work

Efficient training of machine learning models via parameter/inference servers is not unexplored. One of the first studies in this direction designed an inference server for latent topic models which allowed for simultaneous sampling on a cluster of nodes (Smola and Narayanamurthy, 2010). Smola and Narayanamurthy (2010) introduce a novel communication model which uses (key,value) pairs to ensure all machines are in a compatible state. Similarly, Ahmed et al. designed and implemented an inference server for latent variable models (2012). The study offers novel solutions to fundamental problems of an inference server: they use a delta-based aggregation system for efficient communication between nodes, use schedule-aware out-of-core storage for dealing with big data and adapt approximate forward sampling in order to be able make the system work for streaming data like news. In addition to inference servers, other research focuses on scalable and efficient systems for deep learning. DistBelief is a system designed to efficiently train very large neural network models on thousands of machines (Dean et al., 2012). They used an asynchronous stochastic gradient descent procedure and distributed batch optimizations to facilitate efficient training. Ho et al. (2013) also implemented a parameter server which incorporates a Stale Synchronous Parallel model so that many machines can make updates to the model simultaneously. Li et al. (2014) implemented a parameter server that distributes data and training among nodes. They designed a system that is highly scalable and available. Some other studies focused on more scalable solutions to introduce frameworks for distributed training of machine learning models. One example is Apache Mahout: an API for distributed algorithm development (ApacheFoundation, 2012). Another is MLI: an API to make it easy for clients to design and implement distributed machine learning algorithms (Sparks et al., 2013). Our work further explores the space of distributed deep learning training servers and we discuss further optimizations to make the system more efficient yet reliable.

In terms of our optimizations, we perform three experiments. The first experiment (Section 8.1) analyzes greedy progress with what amounts to a reduced batch size. Although larger batch sizes mean better estimates of the overall training gradient, some papers have looked into small batch sizes as in Masters and Luschi (2018). Clearly, small batch sizes can work well, particularly since stochastic gradient descent (single sample) itself has been shown to have nice convergence properties (Shamir and Zhang, 2013). The second experiment (Section 8.2) takes a look at reducing precision in networks which is also certainly not an unexplored territory for deep networks (examples include Hubara et al. (2016); Wu et al. (2018). Lastly, the third section takes a look at network pruning for transmitting networks mainly in the setting of multi-task neural networks as discussed in (Ruder, 2017). A slightly different example of network pruning that may be applicable to our case is presented in (Lu et al., 1996).

3 System Architecture

Our server consists of a Central Controller and N worker nodes. The Central Controller takes care of reading the input data, splitting the training data into batches and distributing among nodes, collecting the gradients from all the nodes to update the network weights, and take care of client communication and node failures. Nodes in our system receive a portion of the training data and a snapshot of the model and compute gradients they send to the Central Controller.

Our system is a TCP server that maintains connections via sockets. Central Controller consists of several independent components. Communication Manager is responsible for maintaining communications with clients. Communications with clients are handled in parallel; therefore, training service can be offered to multiple clients at the same time. Communication manager reads the neural networks model and dataset file from the user, and when the training/inference is completed, it returns the final model and results. Messenger module is responsible for communication with the nodes. The messenger module can concurrently send messages to nodes and receive the responses from the nodes. The passing of model weights and gradients are also taken care of by the messenger. File System is another component which takes care of reading input and model files. Our Training and Inference Server reads model and data files, and the updated models are again saved to files (although the models would likely need to be transmitted in a setting where there are no shared memory/disks). Failure Manager is responsible for detecting the failed worker nodes and ensuring availability when all the nodes are not functioning properly. The Central Controller controls all sub-managers like Communication Manager or Messenger. The Worker nodes are responsible for calculating gradients. Worker nodes communicate to Central Controller through Messenger to receive their data and models as well as send the gradients. Figure 1 shows the architecture of our system.

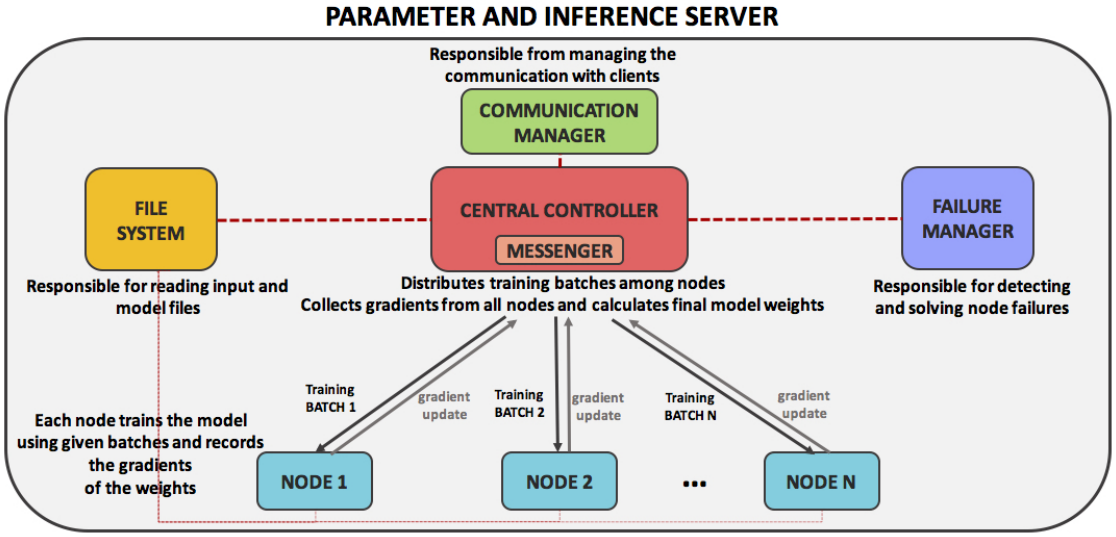

Figure 1: The system architecture of Parameter and Inference Server

4 Training and Inference Protocol

Assuming that all nodes are functional during the training process, our Training and Inference Server implements the following protocol:

Algorithm 1 Training and Inference Protocol for Central Controller

procedure CENTRALCONTROLLERPROTOCOL
    Receive connection request from client
    model_file ← *read(neuralnetwork_model)*
    data_file ← *read(training_data)*
    S ← *size(training_data)*
    for *epoch* ≤ *number_of_epochs* **do**
        for $n \leq$ *number_of_nodes* **do**
            $batch_n$ ← *random_batch*(*training_data*)
            Send model and $batch_n$ *to* $node_n$
            $gradient_n$ ← *receive from* $node_n$
            $gradients[n]$ ← $gradient_n$
        gradient_update ← *mean(gradients)*
        model_weights ← *old_weights - learning_rate * gradient_update*
    Send final model to client
    Close connection

Algorithm 2 Training and Inference Protocol for Worker Nodes

procedure WORKERNODEPROTOCOL
    Receive weights and input file from Central Controller
    model_weights ← *read(weights_file)*
    batch ← $read(input\_file)_{batch_n}$
    gradient ← *Stochastic Gradient Descent* ($batch_n$)
    Send gradient to Central Controller

Detailed description of Training and Inference protocol is provided in Figure 2.

5 Performance

Our Training and Inference Server aims to offer efficient training of neural network models. Our server distributes batches of training data across different machines or computing clusters to facilitate efficient training. Therefore allowing each independent node to train from a much smaller portion of the data simultaneously (this would likely be beneficial for memory locality). In order to analyze the costs associated with out system, Formula 1 describes the global computational costs we expect to see for a model with N nodes under the assumption of zero node failures.

$$I(|data|) + \sum_{batch \in data} \Big(U(|batch|) + \sum_{n \in nodes} 2L(|model|) + G(|model|, \frac{|batch|}{|nodes|}) \Big) \tag{1}$$

I is the initialization cost of splitting the data sets. U is the cost to update the weights using the gradient from all N nodes, $2L$ is the latency of communication, once from the central controller to the node and once from the node to the central controller. Then, G is the gradient computation cost. Although our server computes the same batch gradient descent as in a sequential case, the global cost of parallel computation is larger because we have to transmit the models (meaning an increase in cost of roughly $\sum_{batch \in data} \sum_{n \in nodes} 2L(|model|)$). Of course, we would hope that this cost would be reduced thanks to the parallelization that occurs over the node summation: $\sum_{n \in nodes}$. In words, the Training and Inference Server requires a large enough network, a large enough number of worker nodes, and enough data to make

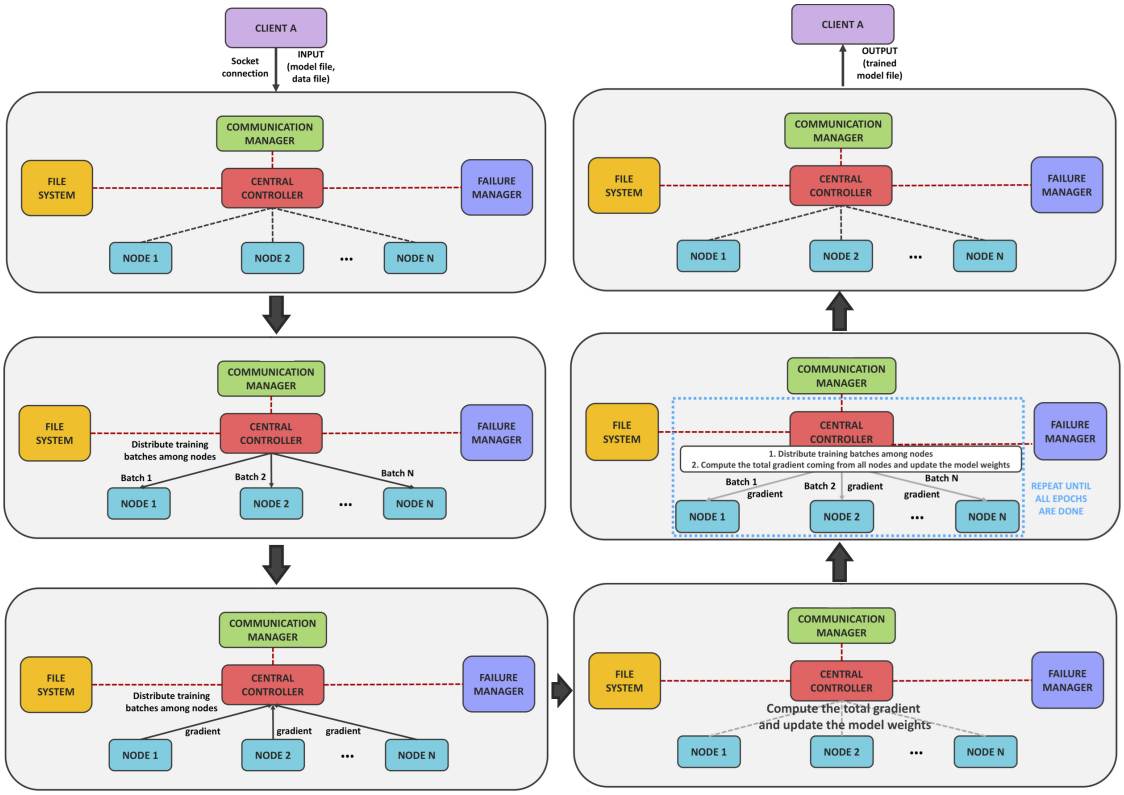

Figure 2: The graphical description of the training procedure for Parameter and Inference Server.

transferring models a negligible cost in comparison to parallelization. As a final observation, the latency is the primary cost that is introduced by our server. As such, we describe methods for reducing the latency in Sections 8.2 and 8.3.

Another significant contributing factor to performance (that we do not highlight in our formula) is the frequency of node failures and the overhead of managing such failures. The time spent in case of failure depends on the requirements of the client. If the client does not require all epochs to be trained from all training samples, as long as the majority of the nodes are available, the training can continue without any interruptions. However, if we want to detect any node failures and redistribute the batches among the rest of the nodes, the training would slow down. In this case, the frequency of node failures would be an essential factor for performance.

6 Availability

Our Training and Inference Server aims to maximize the availability of the system. In the event of a node failure, the samples are redistributed to the remaining nodes. When the central controller sends the batches of data to worker threads, it waits for the gradient from all nodes and then continues with the calculation of model weights. The Central Controller times out when it cannot receive the gradient descent update from all nodes. In this case, we distribute the data among remaining nodes and resume batch gradient descent, assuming we have fewer nodes now. This result is tied to the results shown in Section 8.1.

7 Scalability

Our system aims to offer a scalable server which can work for a high number of nodes and clients. Our system allows multiple client communications to make answering concurrent requests possible. The protocol

we implemented makes it possible to increase the number of Central Controllers to make the system larger-scale. Furthermore, the system can work with any number of nodes, and additional nodes can be added to the server as long as the Central Controller recognizes the new machine/node. Thus, the fundamental design of our system aims to find scalable design solutions to ensure reliable and efficient service even in a much bigger system.

8 Optimizations

In this section, we discuss several optimizations. The first is answering whether we can make progress without the gradients from all nodes (related to availability). The second and third are aiming to reduce latency costs associated with transmitting potentially large networks. We do not actually implement these in our system, but merely analyze them to determine their merits.

8.1 Experiment - Greedily progress without all gradients

We can evaluate the network on a validation set (inference) and make progress as long as our metric (AP, ROC-AUC, log-loss, etc.) has improved. This methodology assumes evaluating our metric on a validation set is cheap. We perform an experiment on the popular MNIST data set which contains 60,000 training images, 10,000 test images (28 by 28 pixels).

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

We train a simple MLP with two layers, each with 512 nodes:

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
        optimizer=RMSprop(),metrics=['accuracy])
```

In our setup, we will have 10 nodes for training, and a batch size of 100. Additionally, we will assume we have nodes dedicated to inference to evaluate our model on a validation set. In the **default case**, we simply wait until all nodes have returned which results in gradients that are for all 10 nodes (each with 10 samples) resulting in a full batch size of 100. For the **greedy case**, we evaluate the validation performance every time we obtain the results from a mini batch. If the validation performance improves, then we don't wait for the remaining nodes and we update this can result in full batch sizes of $10, 20, 30, \cdots, 100$.

Figure 3 suggests that we can perform updates with varying batch sizes, and that smaller batch sizes may even be preferable for the initial epochs of training. One important caveat is that this requires many evaluations on the validation set, which would likely surpass the benefits afforded by this greedy mini-batching. This framework suggests that, particularly for early epochs, smaller batch sizes (which often result in greater variability) are sufficient for improving the performance of the network. Later on, more mini-batches are required to continually improve the network. This optimization could certainly be incorporated within our parameter/inference server.

8.2 Experiment - Reduce precision of network parameters

Reduce precision (to reduce communication costs) similar to this paper and this paper.

In this experiment we primarily look at reducing precision for inference, although future work on reducing precision for training could be meaningful as well. We quantize weights by:

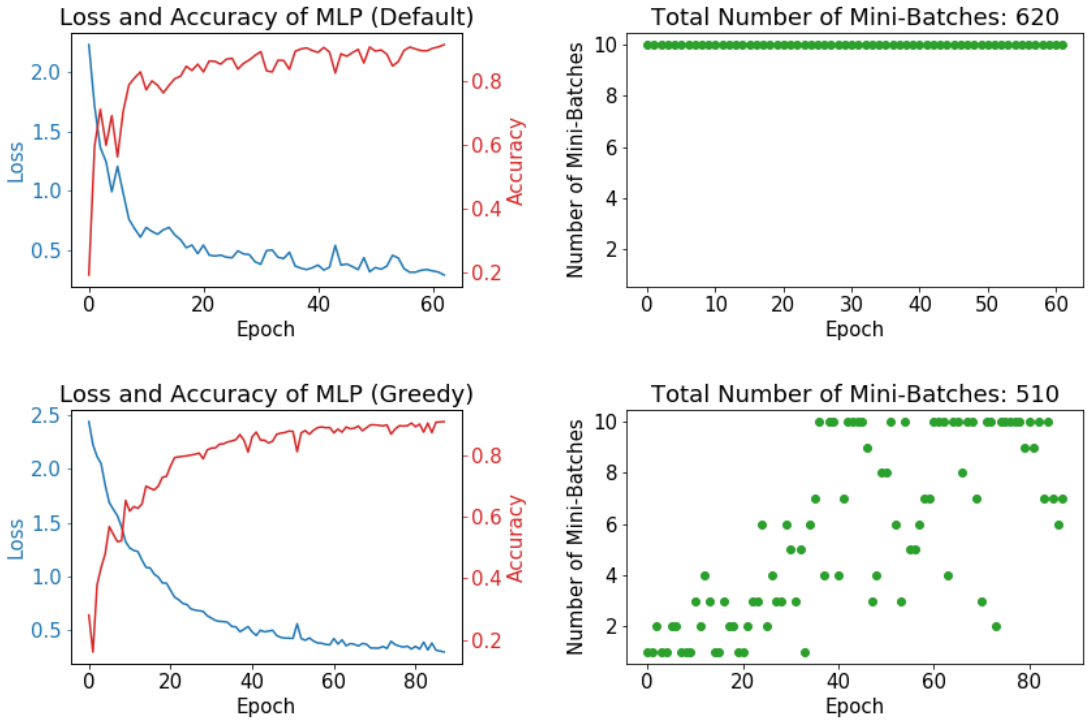

Figure 3: Both models are trained to the same validation loss (of 0.3) which results in an accuracy of around .88. We can see that the number of mini-batches, which can be considered a surrogate for the overall run-time. In the greedy approach it is roughly 80% of the number of mini-batches evaluated in the default case.

```
def quantize(weight,base=1):
    return(np.round(weight/base)*base)
```

Quantizing weights in this manner results in fewer unique weights in the model. We reduce the number of unique weights in the models until we reach the single digits, implying that the communication costs L and potentially the training costs G or inference costs could be reduced as well.

8.2.1 Data sets and models

Reuters newswire topics classification (MLP) – Dataset of 11,228 newswires from Reuters, labeled over 46 topics. As with the IMDB dataset, each wire is encoded as a sequence of word indexes (same conventions). Trained an MLP:

```
model = Sequential()
model.add(Dense(512, input_shape=(max_words,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
    optimizer='adam',metrics=['accuracy'])
```

MNIST database of handwritten digits (CNN) – Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

```
model = Sequential()
```

```
model.add(Dense(512, input_shape=(max_words,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
    optimizer='adam',metrics=['accuracy'])
```

IMDB Movie reviews sentiment classification (LSTM) – Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers). LSTM model:

```
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
    optimizer='adam',metrics=['accuracy'])
```

8.2.2 Results

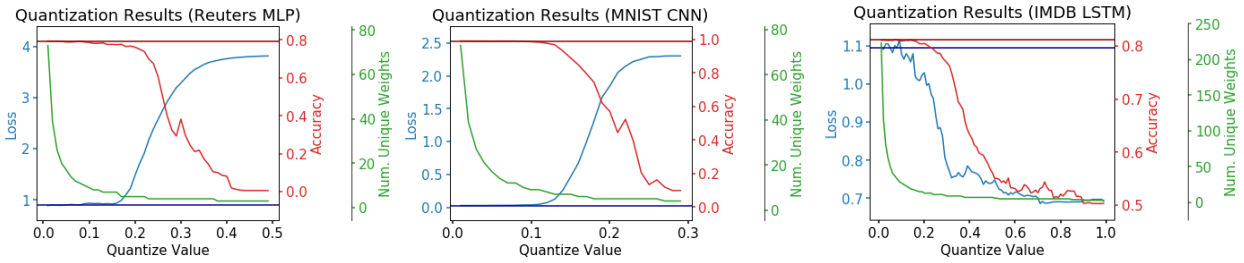

Figure 4: Reduce the quantization for all three data sets and model types.

Across all data sets and model types, we can see in Figure 4, that we are able to quantize extremely effectively in terms of inference. For the MLP, CNN, and the LSTM, we see that we are able to get to very low amounts of unique weights, before the performance on the test set degrades. Interestingly, for the Reuters MLP we are able to have weights composed of only 32 different numbers without any degradation in performance. This suggests that we only need 5 bits to represent each weight in order to get good inference on the validation set. For the MNIST CNN, we see even stronger results, where there is no degradation even at roughly 16 different numbers, which means each weight only requires 4 bits. For the IMDB LSTM we see similar results that suggest that the models we trained with 32 bits (representing floats), can be significantly quantized before transferring the models to perform inference.

8.3 Analysis - Network pruning

Network pruning is another interesting areas of optimization. We may reduce the communication cost by pruning the network whenever it is prudent to do so. One prominent example where pruning would come in handy is in multi-task learning. If we prescribe to the framework in Figure 5, where the shared layers are all the same size. Here, we can define the number of tasks to be $\mathcal{T}$, the number of parameters in the shared layers to be $\mathcal{W}_s$ and the number of parameters in a single task specific layer to be $\mathcal{W}_t$.

Assuming the communication cost L is on the order of the size of the network being transferred, we have:

$$L \propto \mathcal{W}_s + \mathcal{T} \times \mathcal{W}_t \tag{2}$$

However, if we operate under the framework of network pruning learning, L^{prune} would be:

$$L^{prune} \propto \mathcal{W}_s + \mathcal{W}_t \tag{3}$$

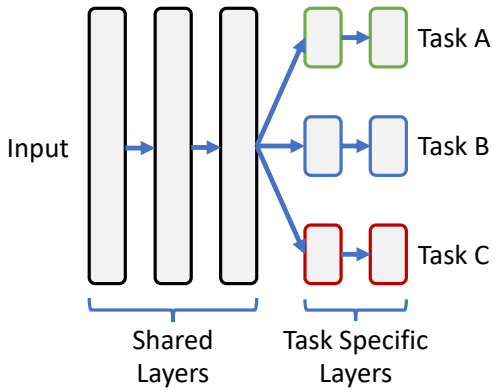

Figure 5: Multi-task learning - an example where network pruning may be applied.

Since these networks are transferred over the duration of an entire training sequence, we have that these costs would be multiplied by the product of the number of nodes and number of epochs, which suggests that pruning would give us an overall saving of $|epochs| \times |nodes| \times (\mathcal{T} - 1) \times \mathcal{W}_t$ over the course of training, which could be quite meaningful under large network settings.

9 Discussion

In this project, we designed and implemented a parameter and inference server. Our server distributes the batches of training data different nodes to calculate the updated weights of the neural network. Our parameter and inference server consists of a Central Controller and multiple worker nodes. Our server takes the input file, and the model file from the client and equally distributes the training batches among the nodes of our system and then collects gradients from all nodes to calculate the final gradient and update the weights. These steps are repeated for all epochs, and the final trained model is returned to the client.

Our system aims to offer a high-performance yet reliable way to train neural networks. The main performance gain is the parallel training of the model which is enabled by the formulation of batch gradient descent (which averages over many gradients). Assuming that the time of training of all nodes is equal, the training time can be reduced from $\propto |samples|$ to $\propto \frac{|samples|}{|nodes|}$. One of the major drawbacks of this distributed system is the communication overhead of transmitting network weights. We worked on some optimizations to investigate how much the communication cost can be reduced. We showed that reducing the precision of network weights can significantly reduce the communication cost while still obtaining the same accuracy/loss in terms of inference. We also showed that for multi-task learning, the network could be pruned to reduce the size of the model that is passed around by nodes. Currently, our server passes the model weights from the Central Controller to nodes. One potential improvement is to integrate a reliable file system or database management system to our server. This way, it might be enough to pass the model file name only, and the content can be read through a file management system. Data locality may be problematic in such a case if the original training data/model file is located on one machine and each node is working on a different machine. The efficiency might depend on how the file system manages remote data reading. More specifically, if the model files are small, it would be much more efficient to pass the actual model along with messages. However, for huge networks, using a file management system might be preferable. Perhaps the system can be adapted based on the specific client request which would make the system scalable.

Our server aims to maximize availability and ensure reliability in case of a node failure in a performant way. When a node fails (determined by a time out), we redistribute the training data among the remaining nodes and continue with the training. With this approach, we ensure that the results are entirely reliable and we only need to repeat one epoch to make sure that we did not lose the gradient value from a batch. We assume fail-stop behavior for nodes; however, it is a straightforward extension to handle nodes being revived. We also showed in Section 8.1 that there is nothing sacred about the batch size. Particularly for the first epochs, the model can make progress even without the full batch gradient. Even though our system is robust to node failures, the system is entirely dependent on the central controller, and if the central machine

fails, all the server operations will be blocked. Our server can be made much more available if a Paxos-like system can be adapted to eliminate the central manager. This way, the model state will be consistently updated.

Another interesting aspect is fairness. Currently, our system assumes that each worker node is equal and distributes the batches equally among nodes. The system performance can be improved by taking the capacities of worker nodes into account. Since each worker node can be running on a different machine or CPU/GPU, the server can send messages to all worker nodes to learn about the memory and CPU/GPU capacities. This way, more samples can be sent to higher-capacity nodes, and busy or low-capacity machines can take less or no new samples. The gradient update will be entirely reliable if the weighted average of gradients from nodes is taken. This optimization can also be extended to multiple clients by assigning priorities in a flexible way.

Our system also offers a scalable server. The current model can work with any number of nodes, but we assume that the central controller knows the list of worker nodes beforehand. The system can easily be extended to recognize any node or machine. Then a client can introduce a new set of nodes to the system and train using those nodes as well. This way the server can become highly scalable, and it can train on the nodes the client owns.

In this project, we implemented a *Parameter and Inference Server* to enable efficient training of neural network. We focused on optimizations which can improve performance and availability and we also discussed future improvements to the system.

References

Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132. ACM, 2012.

ApacheFoundation. Mahout project, 2012. URL `http://mahout.apache.org`.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations, 2016.

Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

Hongjun Lu, Rudy Setiono, and Huan Liu. Effective data mining using neural networks. *IEEE transactions on knowledge and data engineering*, 8(6):957–961, 1996.

Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

Ohad Shamir and Tong Zhang. Stochastic gradient descent for non-smooth optimization: Convergence results and optimal averaging schemes. In *International Conference on Machine Learning*, pages 71–79, 2013.

Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1187–1192. IEEE, 2013.

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.