Layer-wise Asynchronous Training of Neural Network with Synthetic Gradient

Final Report

Xupeng Tong
Carnegie Mellon University
Computational Biology Department
xtong@andrew.cmu.edu

Hao Wang
Carnegie Mellon University
Language Technologies Institute
haow2@andrew.cmu.edu

Ning Dong
Carnegie Mellon University
Language Technologies Institute
ndong1@andrew.cmu.edu

Griffin Thomas Adams
Carnegie Mellon University
Language Technologies Institute
gta@andrew.cmu.edu

ABSTRACT

Parallelization of Neural Networks has been a hot topic in recent years. In the summer of 2016, researchers at Google Deepmind proposed a new approach called Decoupled Neural Interface (DNI) to address this issue. DNI fully decouples the network by approximating the true input in the forward pass, and the true gradient in the backward pass. In this project, we implement a system inspired by DNI to fully asynchronously train Neural Networks. By deploying a synthetic gradient, we unlock the sequential dependency of network layers during back propagation and achieve higher training speed without sacrificing accuracy. While we also test the Synthetic Input, we find that under our experimental conditions it weakens accuracy. We apply DNI to three different architectures, which differ in terms of the use of input and gradient approximations. We thoroughly discuss the result for each architecture. We test our results on the CIFAR-10 and MNIST image datasets.

Keywords

Asynchronous training; Neural Network; Synthetic Gradient; CIFAR-10; MNIST

1. INTRODUCTION

When dealing with large datasets, parallel computation can provide enormous increases in speed. In recent years, the dual emergence of affordable cloud services and multi-core machines has made parallelization of computation more efficient and cost-effective than ever.

Applying parallelization to the training of neural networks, however, is a nontrivial task. In a typical feed forward neural network, computation is carried out sequentially, layer by layer. During feed forward, downstream layers rely on the output of previous layers as input. During back propagation of the loss function, each layer's gradient update depends on the error signal generated from the downstream layer. The error function is a composite function; application of the chain rule easily reveals that many partial derivative calculations can be re-used for updating each weight in the network. Nonetheless, the sequential nature of the operations is still a significant bottleneck to increasing training and testing speed. This layer locking problem is even more pronounced for multiple neural networks interacting with each other on different, asynchronous time scales [1].

Relaxing some of the stricter requirements of gradient descent can help. Namely, relaxation of the constraint that the gradient be computed with the most up-to-date parameters enables parallelization of both training and testing. Asynchronous Stochastic Gradient Descent (ASGD) represents an intuitive approach to parallelization [2].

ASGD employs a parameter server to manage independent workers. Each worker requests the most up-to-date parameters from the parameter server and computes the gradient on a minibatch of the data. The workers then send the gradients computed on their respective minibatches back to the parameter server which updates the weights accordingly.

Yet while ASGD parallelizes the data, it does not parallelize the model. A fundamental question remains: can

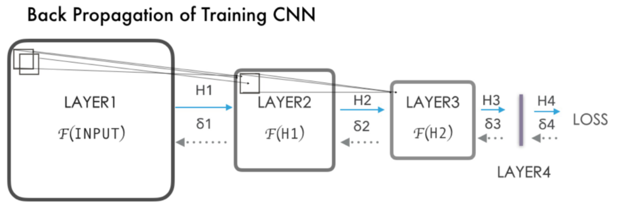


Figure 1: Back Propagation training of Neural Network

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

the layers themselves be trained in parallel? A recent paper published by Google Deepmind demonstrates that model parallelization is indeed possible [1]. The proposed method by the researchers at DeepMind [1] employs two additional small neural networks, that simulate the input and gradient update whenever a layer makes a request.

This lock-free approach to training neural networks could be applied in tasks involving many small neural networks which cross−communicate. Our project, however, does not focus on this discovery. In this project, we build upon preexisting work to develop an elegant and robust approach to layer-wise parallel training of a neural network. Other work proves the effectiveness of the complete decoupled model [1] on simple datasets, such as MNIST. Our goal is to validate this idea and deploy models on a distributed system.

To date, little work has been undertaken on a distributed version of this model. Extensive experimentation demonstrates reasonable effectiveness on naÃŕve to complex models. Future work might focus on decreasing the variance of the input and the gradient update simulated by auxiliary neural networks. Additionally, the data storage and cache system represent other areas for improvement.

2. DATASETS

2.1 CIFAR-10

We test our framework using the CIFAR-10 dataset developed by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset consists of 60,000 labeled images, a small subset of the 80 million tiny images dataset [3]. The images are evenly divided among 10 mutually exclusive classes ranging from airplanes and automobiles to deer and horses. Each data point represents a 32x32 pixel color image. 10,000 of the 60,000 image collection is held-out, to be used for validation to prevent overfitting. The held-out set maintains even representation among the 10 labels - i.e., each label submits 1,000 images to the test set. The 50,000 remaining images are used for training, partitioned into 5 mini-batches of 1,000 images, with no guarantees fors a uniform distribution of labels within these mini-batches.

2.2 MNIST

We also test our framework using the MNIST dataset (Mixed National Institute of Standards and Technology) created by LeCun, Cortes, and Burges. The MNIST database is a subset of the larger NIST database, which is comprised of black and white labeled images of handwritten binary digits. The full MNIST database consists of 70,000 labeled images, 60,000 of which are for training purposes, and the remaining 10,000 for testing. The original NIST images were transformed via bounding-box normalization (20x20 pixels) and anti-aliasing (to minimize distortion at lower resolution). The final result is a 28x28 pixel grayscale image. [4]

3. RELATED WORKS

In this section, we focus on introducing several foundational works which either directly informed our research, or simply advanced the state of the art algorithm with enhanced speed in the training of neural networks.

3.1 Asynchronous Stochastic Gradient Descent and Parameter Server

Researchers have made substantial progress on parallelizing gradient descent by exploiting the fact that the gradient need not be computed on the entire dataset to achieve convergence on a target loss function. Such approaches to parallelizing gradient descent full under the umbrella of data parallelization. Asynchronous Stochastic Gradient Descent (ASGD) represents an intuitive and highly approach to data parallelization [2]. ASGD employs a parameter server to manage independent workers. Each worker requests the most up-to-date parameters from the parameter server and computes the gradient on a mini batch of the data. The workers then send the gradients computed on their respective mini batches back to the parameters which updates the weights accordingly. With N workers, the worst-case scenario is that a worker machine is computing its gradient with stale parameters, on the order of N-1 missing updates. Despite compromising on strict mathematically effectiveness via the asynchronous request-update paradigm, this approach has seen wide adoption. Researchers at Google have made tremendous improvement on both the parameter server [5, 6], as well as the algorithm itself [1].

3.2 Synthetic Gradient

Researchers at Google DeepMind recently pioneered a new framework called Decoupled Neural Interfaces (DNI) [1]. DNI unlocks neural layers under back propagation by constructing synthetic gradients. These synthetic gradients approximate the true gradient using only local information. Hence, neural layers can act independently and asynchronously, requesting the approximate gradient immediately after computing the activation. The synthetic gradient approximates the true gradient by minimizing the L2 loss vis-Ãă-vis the true gradient, which it eventually receives during back propagation. The synthetic gradient converges to the true gradient, while the overall training time is substantially reduced. They even extend the predictive modeling to the input space, which allows computational decoupling across layers in the forward pass. They apply DNI and synthetic gradients to Recurrent Neural Networks (RNNs). The use of synthetic gradients enables prediction far into the future, enabling the modelling of long temporal dependencies across steps. They test the model on the Penn Treebank language modeling dataset and prove that capturing long-term dependencies can be co-achieved with substantial speedups in training time.

3.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of feed-forward neural networks commonly used for image classification. Studies of visual processing by animals in the visual cortex inspired the CNN architecture. The building blocks of CNNs are called convolutional layers, which are created by applying a kernel, or filter, function across the input space in an overlapping, contiguous window (often very small relative to the size of the input). Convolutional layers break from the traditional neural network structure since each neuron is not fully connected and shares its parameters/filter with the other neurons in the same feature map. The convolutional layers are then usually pooled - a form of nonlinear subsampling - to reduce the layer size before being sent to a fully-connected output layer [7]. The sequential process of convoluting and pooling can be repeated several times to process more complex inputs. In addition to image classifi-

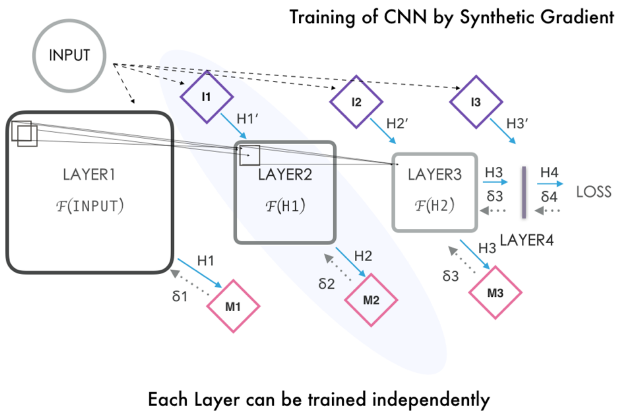


Figure 2: Fully Decoupled Training of Convolutional Neural Network

cation, CNNs have seen widespread adoption in areas where the input can be mapped onto an "image-like" space, such as video recognition, natural language processing, among others.

4. PROBLEM FORMULATION

Let

$$L(w_1, b_1, \cdots, w_i, b_i, \cdots, w_n, b_n) = L(W, B)$$

be the loss function over the neural network, where

$$i \in [1, n]$$

n is the number of layers in the neural network and

$$X_i \triangleq \sigma(X_{i-1} w_i + b_i)$$

be the output of the ith layer, where $\sigma(\cdot)$ is the activation function applied element-wisely to the output. The activation could be the ReLu, sigmoid, or tanh function.

In the back-propagation setting, when calculating the gradient of layer i, by applying the chain rule we can easily achieve

$$\frac{\partial L}{\partial X_i} = \frac{\partial L}{\partial X_n} \frac{\partial X_n}{\partial X_{n-1}} \cdots \frac{\partial X_{i+1}}{\partial X_i}$$

Now consider the synthetic gradient case, where the true gradient is used to tune the parameter of the small neural network M,

$$\delta_i \triangleq \frac{\partial L}{\partial X_i}$$

$$\hat{\delta}_i \triangleq \mathcal{M}(X_i)$$

where $\mathcal{M}(\cdot)$ is defined as a small neural network of layer two or three.

$$L_{\mathcal{M}} = \sum_i \|\delta_i - \hat{\delta}_i\|_2^2$$

Let's change the notation by defining the output of the i th layer to,

$$h_i \triangleq X_i$$

$$\hat{h}_i = \mathcal{I}(h_{i-1})$$

where $\mathcal{I}(\cdot)$ is defined as a small neural network of layer two or three, with similar setting of $\mathcal{M}(\cdot)$.

And its loss function is defined as,

$$L_{\mathcal{I}} = \sum_i \|h_i - \hat{h}_i\|_2^2$$

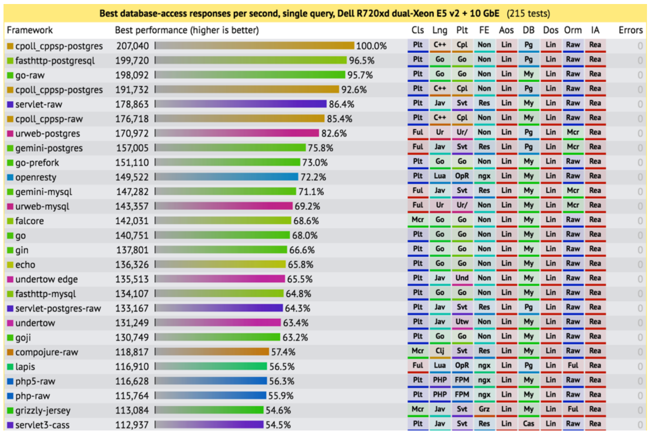
From this setting, each layer can be completely decoupled from its previous and following layer.

5. METHOD

5.1 Direct minimization over the gradient loss

To prove the effectiveness of the synthetic gradient that is proposed, we first construct a convolutional neural network based on back propagation. The convolutional neural network based on back propagation is built as follows,

Input Layer
↓
Convolutional Layer 1, kernel size = [3,3], nodes=32, ReLu
↓
Convolutional Layer 2, kernel size = [3,3], nodes=32, ReLu
↓
Max Pool Layer, pool size = [2,2]
↓
Dropout Layer, keep probability=0.25
↓
Convolutional Layer 3, kernel size = [3,3], nodes=64, ReLu
↓

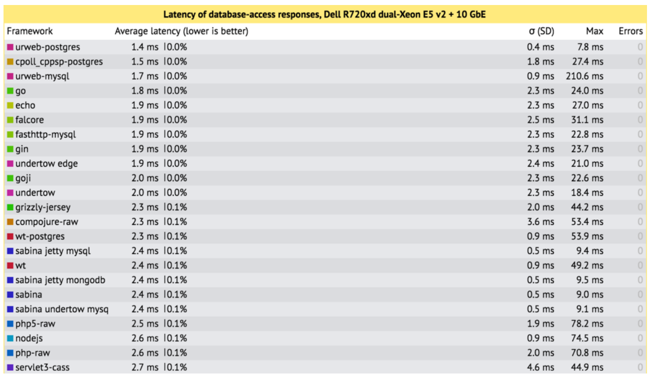


Best database-access responses per second, single query, Dell R720xd dual-Xeon E5 v2 + 10 GbE (215 tests)

Framework	Best performance (higher is better)		Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	Errors
cpoll_cppsp-postgres	207,040	100.0%	Plt	C++	Cpl	Non	Lin	Pg	Lin	Raw	Rea	0
fasthttp-postgresql	199,720	96.5%	Plt	Go	Go	Non	Lin	Pg	Lin	Raw	Rea	0
go-raw	198,092	95.7%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
cpoll_cppsp-postgres	191,732	92.6%	Plt	C++	Cpl	Non	Lin	Pg	Lin	Raw	Rea	0
servlet-raw	178,863	86.4%	Plt	Jav	Svt	Res	Lin	My	Lin	Raw	Rea	0
cpoll_cppsp-raw	176,718	85.4%	Plt	C++	Cpl	Non	Lin	My	Lin	Raw	Rea	0
urweb-postgres	170,972	82.6%	Ful	Ur	Ur/	Non	Lin	Pg	Lin	Mcr	Rea	0
gemini-postgres	157,005	75.8%	Ful	Jav	Svt	Res	Lin	Pg	Lin	Mcr	Rea	0
go-prefork	151,110	73.0%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
openresty	149,522	72.2%	Plt	Lua	OpR	ngx	Lin	My	Lin	Raw	Rea	0
gemini-mysql	147,282	71.1%	Ful	Jav	Svt	Res	Lin	My	Lin	Mcr	Rea	0
urweb-mysql	143,357	69.2%	Ful	Ur	Ur/	Non	Lin	My	Lin	Mcr	Rea	0
falcore	142,031	68.6%	Mcr	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
go	140,751	68.0%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
gin	137,801	66.6%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
echo	136,326	65.8%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
undertow edge	135,513	65.5%	Plt	Jav	Und	Non	Lin	My	Lin	Raw	Rea	0
fasthttp-mysql	134,107	64.8%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
servlet-postgres-raw	133,167	64.3%	Plt	Jav	Svt	Res	Lin	Pg	Lin	Raw	Rea	0
undertow	131,249	63.4%	Plt	Jav	Utw	Non	Lin	My	Lin	Raw	Rea	0
goji	130,749	63.2%	Plt	Go	Go	Non	Lin	My	Lin	Raw	Rea	0
compojure-raw	118,817	57.4%	Mcr	Clj	Svt	Res	Lin	My	Lin	Raw	Rea	0
lapis	116,910	56.5%	Ful	Lua	OpR	ngx	Lin	Pg	Lin	Ful	Rea	0
php5-raw	116,628	56.3%	Plt	PHP	FPM	ngx	Lin	My	Lin	Raw	Rea	0
php-raw	115,764	55.9%	Plt	PHP	FPM	ngx	Lin	My	Lin	Raw	Rea	0
grizzly-jersey	113,084	54.6%	Mcr	Jav	Svt	Grz	Lin	My	Lin	Ful	Rea	0
servlet3-cass	112,937	54.5%	Plt	Jav	Svt	Res	Lin	Cas	Lin	Raw	Rea	0

Figure 3: Comparison on Database Access Responses Per Second for Single Query of Main Web Servers

[8]



Latency of database-access responses, Dell R720xd dual-Xeon E5 v2 + 10 GbE

Framework	Average latency (lower is better)		σ (SD)	Max	Errors
urweb-postgres	1.4 ms	0.0%	0.4 ms	7.8 ms	0
cpoll_cppsp-postgres	1.5 ms	0.0%	1.8 ms	27.4 ms	0
urweb-mysql	1.7 ms	0.0%	0.9 ms	210.6 ms	0
go	1.8 ms	0.0%	2.3 ms	24.0 ms	0
echo	1.9 ms	0.0%	2.3 ms	27.0 ms	0
falcore	1.9 ms	0.0%	2.5 ms	31.1 ms	0
fasthttp-mysql	1.9 ms	0.0%	2.3 ms	22.8 ms	0
gin	1.9 ms	0.0%	2.3 ms	23.7 ms	0
undertow edge	1.9 ms	0.0%	2.4 ms	21.0 ms	0
goji	2.0 ms	0.0%	2.3 ms	22.6 ms	0
undertow	2.0 ms	0.0%	2.3 ms	18.4 ms	0
grizzly-jersey	2.3 ms	0.1%	2.0 ms	44.2 ms	0
compojure-raw	2.3 ms	0.1%	3.6 ms	53.4 ms	0
wt-postgres	2.3 ms	0.1%	0.9 ms	53.9 ms	0
sabina jetty mysql	2.4 ms	0.1%	0.5 ms	9.4 ms	0
wt	2.4 ms	0.1%	0.9 ms	49.2 ms	0
sabina jetty mongodb	2.4 ms	0.1%	0.5 ms	9.5 ms	0
sabina	2.4 ms	0.1%	0.5 ms	9.0 ms	0
sabina undertow mysq	2.4 ms	0.1%	0.5 ms	9.1 ms	0
php5-raw	2.5 ms	0.1%	1.9 ms	78.2 ms	0
nodejs	2.6 ms	0.1%	0.9 ms	74.5 ms	0
php-raw	2.6 ms	0.1%	2.0 ms	70.8 ms	0
servlet3-cass	2.7 ms	0.1%	4.6 ms	44.9 ms	0

Figure 4: Comparison on Database Access Responses' Latency for Single Query of Main Web Servers

[8]

$$\begin{array}{r}
\text{Convolutional Layer 4, kernel size} = [3,3]\text{, nodes}=64\text{, ReLu}\\
\downarrow\\
\text{Max Pool Layer, pool size} = [2,2]\\
\downarrow\\
\text{Dropout Layer, keep probability}=0.5\\
\downarrow\\
\text{Dense Layer, nodes}=512\\
\downarrow\\
\text{Dropout Layer, keep probability}=0.25\\
\downarrow\\
\text{Dense Layer, nodes}=10\text{, softmax for classification}
\end{array}$$

Then by adding additional nodes in the computation graph of the convolutional neural network, we are capable of optimizing over the gradient loss $L_{\mathcal{I}} = \sum_i \|h_i - \hat{h}_i\|_2^2$ directly while keeping the gradient of other graph element updated.

5.2 Update Decoupled Architecture

We implement the update-decoupled architecture using thriftpy [9], a Remote Procedure Call (RPC) library in Python. We place all linear models (Ms and Is) in one node N. When each layer in the neural network updates its weights, it requests the synthetic gradient by directly invoking the corresponding function in N. When the true gradient propagates back, the layer directly invokes the update method in N. RPC guarantees that all functions in N invoked by a layer are executed exactly the same as if they were invoked within that layer. We use the update decoupled method to compare the test accuracy between backpropagation and DNI, ceteris paribus number of iterations. Although not a primary concern for this model, faster implementations of RPC could improve wall clock time without compromising accuracy. For example, thriftpy, which is implemented in Pypy, achieves a 10x greater speedup. Building other architectures represents another method for reducing RPC overhead.

5.3 Fully Decoupled Architecture

We finally move toward the fully decouple architecture by synthesizing both the input and the gradient. For the ease of our experiment, we use a two-layer (three layers if input layer is included) ANN structure on the MNIST dataset.

$$\begin{array}{r}
\text{Input Layer}\\
\downarrow\\
\text{Dense Layer, nodes}=200\text{, Sigmoid}\\
\downarrow\\
\text{Dense Layer, nodes}=10\text{, Softmax for classification}
\end{array}$$

We conduct extensive analysis on the decoupled architecture with regard to the parameters tunings, removing the first layer, etc.

6. INFRASTRUCTURE

6.1 Web Server

We chose Undertow as the web server, which is a highly performant, flexible web server written in java. To arrive at this choice, we compared main web servers' access responses with respect to responses per second (Figure 13) and latency (Figure 14). We found Undertow outperforms many other alternatives in addition to being lightweight in terms of code, runtime memory usage, and minimal unused functionality [10]. Although we used Python for machine learning computation on TensorFlow, it is not as performant as java on the web server side.

6.2 Database

In this project, we use a MySQL database. A MySQL database best satisfies our dual requirements of reliability and support for mature transactions. Since operations need to be thread safe and speed is not of paramount importance, we avoided an in-memory database. Since the master node completes all work for the database with each single thread, the benefit from increased speed from an in-memory database is not impactful enough. We considered a NoSQL database since all insert operations are append-only and NoSQL offers the easiest mechanism for appending data. Yet we ultimately chose SQL over NoSQL because we placed more priority on support for mature transactions.

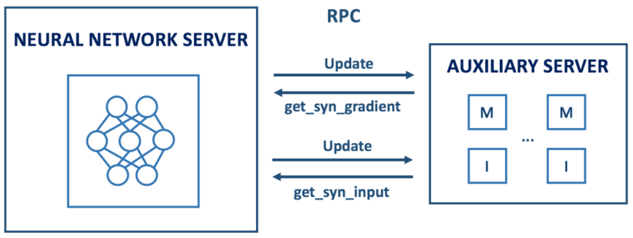


Figure 5: Architecture for Combining True Inputs and True Gradients

6.3 Synchronous Training

Synchronous training represents the ordinary method for training Artificial Neural Networks (ANNs) or Convolutional Neural Networks (CNNs). We employ a single machine for synchronous training, working on each layer sequentially.

6.4 Asynchronous Training

For Asynchronous training, there are two possible architectures. The only difference relates to computation of the true input and true gradient. In the first architecture, an auxiliary server handles computation of both the true input and true gradients. In the second architecture, these computations are decoupled on a layer-by-layer basis, i.e. true gradients and true inputs are computed locally at each layer.

6.5 Seperate M's from Neural Network

For the first architecture, we include a module for computing for computing synthetic gradients in the same auxiliary server. The neural network nodes for computing the network are separate, and communicate with the auxiliary server via a Remote Procedure Call (RPC). When a neural network node needs the synthetic gradient, it makes an RPC to request the value from the auxiliary server. The neural network server uses the synthetic gradient to update its weight, and then when true gradient is propogated back, it is sent to the corresponding gradient module M in the auxiliary server. The auxiliary server, in turn, receives the true gradient and updates the weight accordingly. Also, this architure supports adding input module I.

For this architecture, computations are perfectly divided into two parts, which makes each part of the system highly independent. Layers need not wait for each other for gradients. This architecture, however, involves higher network communication cost because for each update on each layer, the neural network node needs to send two extra HTTP requests, in addition to receiving the result after two more HTTP requests. These network calls make the system insufficiently fast. This bottleneck motivates the second architectureâĂŹs attempt to minimize the number calls in network via local computation.

6.6 Group Each Layer Together

The second architecture reduces network costs by grouping all computations in the same layer. Unlike the first architecture for asynchronous training, synthetic gradients and synthetic inputs are computed locally. For each iteration, this local computation eliminates the need for four additional HTTP requests. In real time, this amounts to a roughly one second speedup for each iteration. As the net-

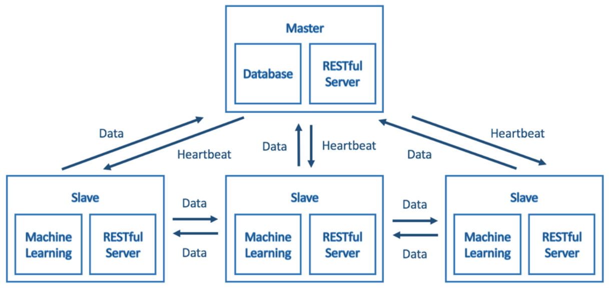


Figure 6: Architecture for Grouping Each Layer Together

work gets deeper, the speedup resulting from fewer network calls improves linearly and can lead to substantial absolute gains in training time.

The master server starts out with two threads. One thread handles RESTful requests, while the other monitors the health of each slave node. Upon each request to the master node, Undertow automatically creates a request handler.

The thread for monitoring health runs indefinitely until the server crashes or is explicitly closed. On fixed intervals, the thread checks, then saves to the database, the health and utilization for each slave node. If the slave node does not provide a response for two consecutive check points, it is marked as failed. After marking the server as failed, the server notifies users to check the server before deciding to deal with the problem or simply restarting it. The server also summarizes the results to give the user insight onto the real-time status of each slave node in RESTful web service. Yet, requesting slave node usage too frequently dramatically diminishes performance. Choosing the right request interval represents a time versus insight tradeoff.

The RESTful web service accepts the data from each slave node. When a slave node finishes processing its mini-batch, it is ready to send the result, the updated gradient or the processed data, to the previous or next layer. Before doing so, the slave node sends an additional request to the master node, an indication to the master node to back up the results. Should the slave node become problematic, then, the user can easily identify the timing of the problem and rollback the slave servers to a particular status.

Each slave node has two components: one representing computation, mainly by GPU on TensorFlow, the other a web service. The web service receives data from the url and responds with a report of its own utilization to master nodes and users.

We extract CPU and disk utilization using java OperatingSystemMXBean and runtime. We uncover GPU status using java runtime and NVIDIA commands.

After finishing each mini-batch task, the slave node simultaneously sends two HTTP requests in TensorFlow. The first is directed at previous and/or subsequent layer servers, while the second is directed at the master layer, an indication to back up the results.

It does not make sense for each slave node to back up the weights itself since each node already computes a great deal. Additionally, the slave nodes do not access the database directly.

6.7 Simplified Architecture

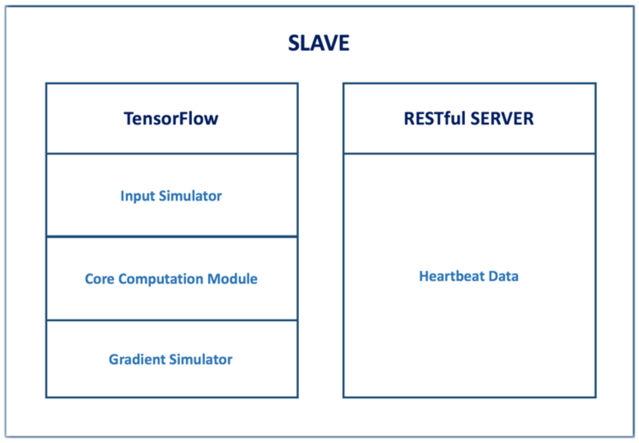


Figure 7: Architecture for Slave servers

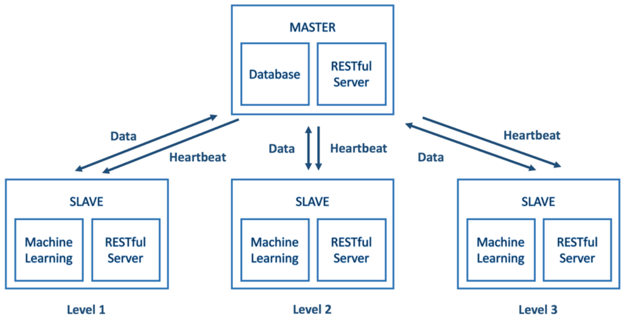


Figure 8: Simplified Architecture for Grouping Each Layer Together

The architecture outlined above is not simplified enough for the configuration. To simplify it, we remove all databases in slave servers, and put all databases in the master server alone. When a layer finishes a mini-batch iteration, it passes the parameters to the master server by the RESTful API. It issues another HTTP request to receive the parameters from the master server. This approach allows the slave server to only have to know the DNS and port of the master server, not the entire network architecture. This simplified approach does not compromise on performance. Figure 8 reveals the new architecture.

7. EXPERIMENTAL RESULTS

7.1 Direct minimization over the gradient loss

To see whether the synthetic gradient works, we implement and test the architecture of our convolutional neural network, then add additional nodes in the computational graph.

In addition to the general loss, we defined the gradient loss, which is

$$L_{\mathcal{M}} = \sum_i \|\delta_i - \hat{\delta}_i\|_2^2$$

In the initial model we built, we directly minimized $L_{\mathcal{M}}$ with respect to the layers, while updating other parameters

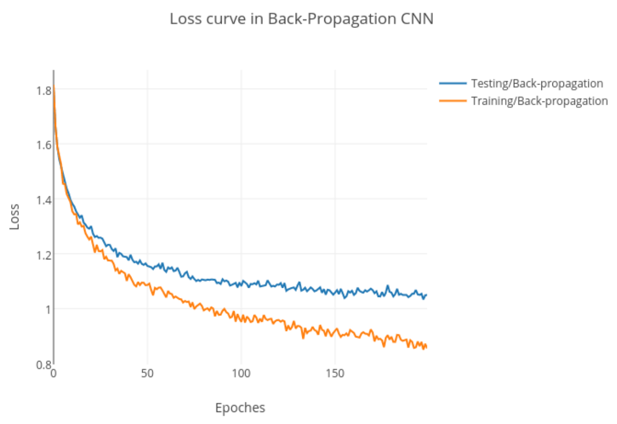


Figure 9: Loss curve with Back-propagation

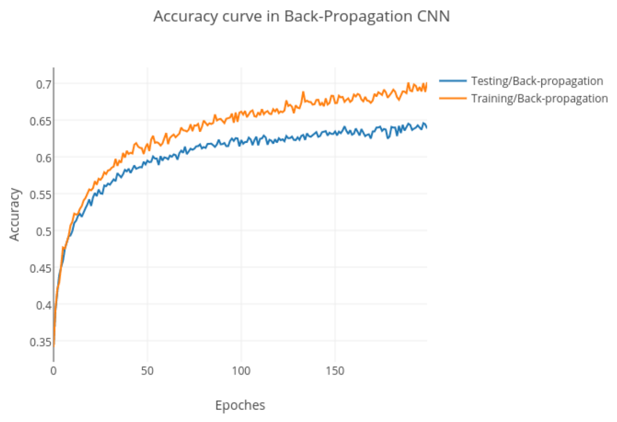


Figure 10: Accuracy curve with Back-propagation

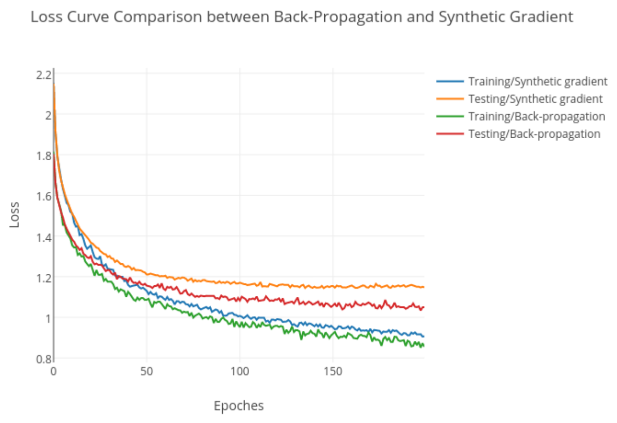


Figure 11: Loss Curve Comparision between Back-Propagation and Synthetic Gradient

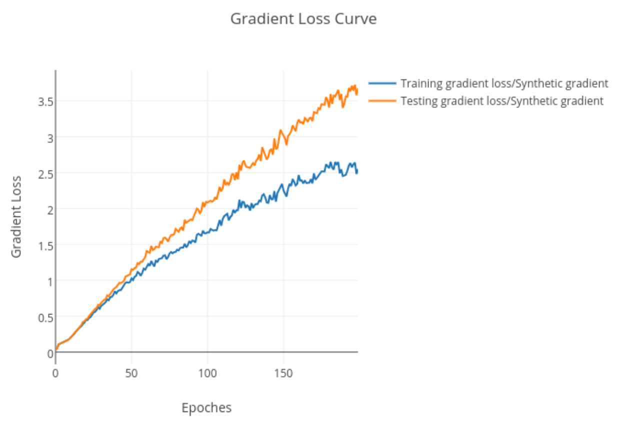


Figure 12: Gradient Loss Curve

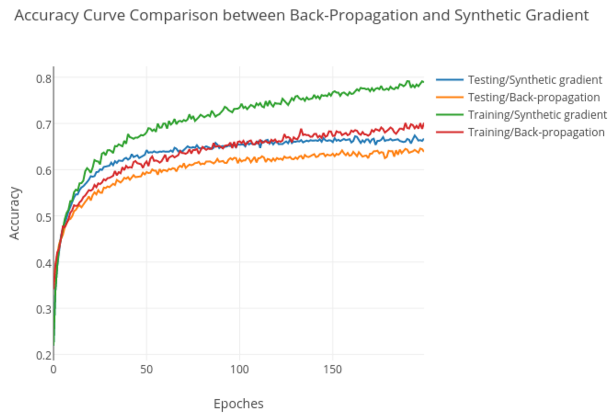


Figure 13: AccuracyCurve Comparison between Back-Propagation and Synthetic Gradient

as well. Our implementation which ran on a single machine trains the neural network in a synchronous way. Our goal is to prove the superiority of this approach on a single machine before moving to a distributed system.

7.2 Update Decoupled Architecture

In Figure 14, we can see that under same learning rate of gradient descent and identical neural network structure, the one using synthetic gradient converges faster. We don't compare the wall clock time because under different network architecture settings it varies a lot. Yet as expected, the baseline without gradient descent reaches higher final accuracy on test set (not shown in the figure).

7.3 Fully Decoupled Architecture

Figure 17 shows the test accuracy increase pattern for the first 200 iterations when introducing synthetic gradient (with learning rate 0.01). Figure 3 shows that of the traditional back propagation neural network without synthetic gradient (with learning rate 0.1). It can be observed that this process with synthetic gradient is more unstable than the traditional neural network where the increase is con-

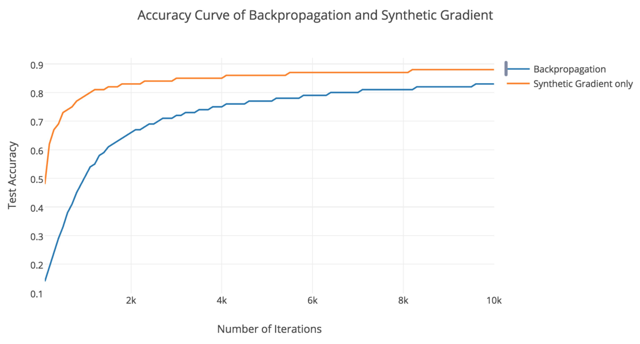


Figure 14: Accuracy Curve of Back-propagation and Synthetic Gradient

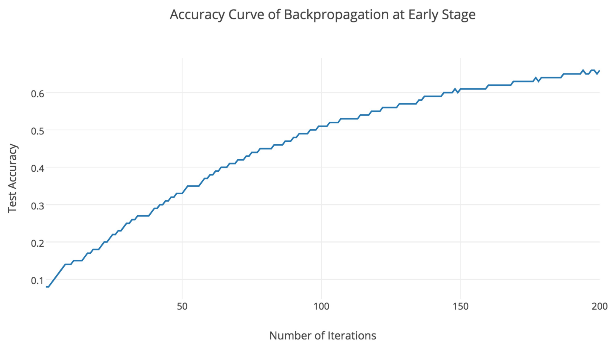


Figure 15: Accuracy Curve of Back-propagation at Early Stage

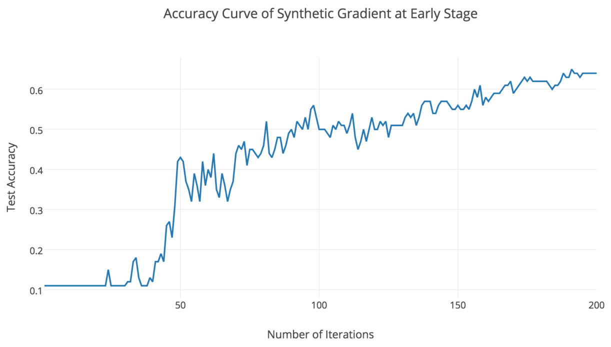


Figure 16: Accuracy Curve of Synthetic Gradient at Early Stage

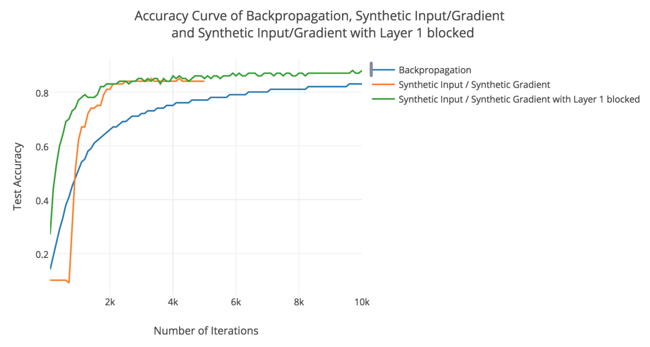


Figure 17: Accuracy Curve of Backpropagation, Synthetic Input/Gradient and Synthetic Input/Gradient with Layer 1 blocked

sistent and smooth. This is intuitive because for the linear model M in the network, the initial weight is randomized and at the beginning of training process, the predicted gradient is not close to true gradient. It needs time for adjustment.

7.4 DNI with Distributed Tensorflow

Distributed Tensorflow offers native support for the distributed execution of a Tensorflow graph. Since a distributed implementation of synthetic gradient training is our main task moving forward, we extensively researched Distributed Tensorflow.

In the process of building DNI on Distributed Tensorflow, we encountered numerous problems. Below, we outline these problems as well as identify solutions.

8. DISCUSSION

8.1 Distributed Tensorflow

1. Distributed Tensorflow is mostly ”transparent” to developers, which mitigates the level of insight into parallelization of the computation. Gradients are calculated in symbolic form prior to running a session; on runtime, we were unable to apply custom gradients retrieved from linear models. The current open-source single-machine implementation defines symbolic gradients by integrating the DNI part. Yet, we cannot ascertain whether or not the model is fully unlocked vis-a-vis the layers, as well as the level of model parallelism attained. To get around this visibility issue, we might need to go beyond simply defining the operation sequence and letting the framework handle the rest. Instead, we might need to manually define different tasks for each worker and add intermediate layers for interaction.

2. Multiple tasks cannot share CPU/GPU. One task would involve using multiple ports to simulate different machines, taking all the memory used in the experiment of running a parameter server and a worker session on a single GPU. Based on the Tensorflow API, a device is the minimum unit for a task. This violates our architecture design, however. This problem is pronounced should we customize each subtask in Distributed Tensorflow, as suggested in the problem stated above. We might tweak the initial design to accommodate the specific mechanisms of Distributed Tensorflow.

3. For future experiments, we need to find a cluster with Tensorflow environment. AWS represents a good choice. Since manually configuring each machine in clusters used for experimentation is a hassle, we might look into using an open-source cluster manager such as Kubernetes.

8.2 Update Decoupled Neural Network

1. In the plot produced, the decoupled CNN achieves a higher accuracy rate than that of the CNN trained with back-propagation. Training both models for 200 epochs reveals a faster convergence rate for the decoupled CNN. Such positive results come with two important caveats. Computational constraints have limited our ability to comprehensively train the CNN structure. As such, we cannot guarantee superiority of our approach over the CNN trained with back propagation. Additionally, the update decoupled CNN is approximately three times slower than the back-propagation CNN per epoch, which might be produced by the training of additional small neural networks used in gradient update.

2. We plot the gradient loss and find that it increases over time. We suspect that the synthetic neural network is struggling to learn sufficient information over an increasingly complex model. We will look to further address this issue.

8.3 Fully Decoupled Neural Network

8.3.1 *Batch trick*

Different layers have different rates of forward evaluation and weight updates. The differing rates produce a divergence of speed in the update of gradient simulator (M) and input simulator (I). Since we store the synthetic gradients and synthetic input we had produced so far in each layer, it will be stored there until a true gradient or true input with the same iteration time to update the model in collaboration. This must be done because it does not make any sense to update the simulator with unmatched true gradient/input and synthetic gradient/input.

In our experiment, we found it unexpectedly, that the last layer is always slower than the first layer, according to the original design, the intermediate result would accumulate in both database and linear models which could not be consumed in time. To better address this problem which is both time and space consuming, we come up with batch trick to restrict the space to $O(N)$ where N is number of batches within a epoch. The idea is as follows.

In different epoches, the model is fed with a sequence of same input, so when we are updating the model and requesting for the true input/gradient from the master server, we can instead querying for its most recent update, by hashing the iteration time to

$$\text{iteration}' = \text{iteration} \ \% \ N$$

By using this trick, the size of intermediate result that stored in both the master server and the slave is bounded. We tested this idea out and found that it indeed works smoothly without affecting the accuracy, and prevent our server from crashing.

8.3.2 Synthetic Input

Removing the first layer of the two layer model - i.e., not updating the first layer parameters - we achieve a better result. This result suggests the synthetic input does not work well under the experimental settings.

We outline several hypotheses for the poor empirical performance of the synthetic input. Model I attempts to learn the input of the current layer given the original input. If the weights for I are randomly initialized and not updated, it represents a random, fixed model which encodes the original input to the size of current layer input. It retains some of the original information and the current layer can learn from it. The random model amounts to the simple application of a transformation matrix to the original input, which is fed directly into the layer.

If the update of I is not working properly, or even worse, the update changes the encoding to effectively unlearn what has been learned so far, the accuracy may decrease. The update of I amounts to an enhancement of the encoding process, to enable the model to retain more information about the original input.

8.3.3 Future Works

For synthetic gradient, our experiments have shown that it could accelerate training by removing locks in neural network meanwhile not affecting the accuracy very much. We plan to discover its usage in RNN where the forward lock in time dimension is significant.

For synthetic input, we have not found a scenario where it's useful. So maybe we should try to come up with new techniques to train smarter I which is better than a random fixed model. To extend our understanding about the synthetic input as encoding matrix, we may consider a scenario that by applying different noise matrices in the same output, this is very much similar to data augmentation where we introduces different noises in the same dataset and try learn things out in a ensemble manner. This is a truly interesting topic and we wish we can have more time exploring it in the future.

We also expect more advanced architecture may be needed to accelerate training and reduce communication cost.

9. CONCLUSION

In this project, we thoroughly explore the Decoupled Neural Interface (DNI) proposed by DeepMind in August 2016. We implement DNI separately for 3 different architectures, and use each method to train and test neural networks in parallel. For the single machine version, we validate the practical feasibility of synthetic gradient. Namely, switching from a true gradient to a synthetic gradient does significantly worsen performance. For the update unlock version (i.e. synthetic gradient only), we find that it outperforms back propagation with respect to speed during the acclimation phase. Upon investigating underlying patterns in this phase, we discover that the test accuracy increase in DNI is more unstable than that that of pure back propagation. For the fully unlocked version, we implement Deepmind's synthetic input, a fundamentally novel concept in the literature on neural networks. But under our experiment settings, synthetic input performs worse than a random model acting as an encoder. Our insights on DNI and future work are proposed above.

10. REFERENCES

[1] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016.

[2] 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.

[3] CIFAR-10. Tinyimages, 2016.

[4] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database. *URL http://yann. lecun. com/exdb/mnist*, 1998.

[5] 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.

[6] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[7] Convolutional neural network. Convolutional neural network — Wikipedia, the free encyclopedia, 2016.

[8] Web Framework Benchmarks. Web framework benchmarks, 2016.

[9] thriftpy, 2016.

[10] undertow, 2016.