Decoupled Parallel Backpropagation with Convergence Guarantee

Zhouyuan Huo [1] **Bin Gu** [1] **Qian Yang** [1] **Heng Huang** [1]

Abstract

Backpropagation algorithm is indispensable for the training of feedforward neural networks. It requires propagating error gradients sequentially from the output layer all the way back to the input layer. The backward locking in backpropagation algorithm constrains us from updating network layers in parallel and fully leveraging the computing resources. Recently, several algorithms have been proposed for breaking the backward locking. However, their performances degrade seriously when networks are deep. In this paper, we propose decoupled parallel backpropagation algorithm for deep learning optimization with convergence guarantee. Firstly, we decouple the backpropagation algorithm using delayed gradients, and show that the backward locking is removed when we split the networks into multiple modules. Then, we utilize decoupled parallel backpropagation in two stochastic methods and prove that our method guarantees convergence to critical points for the non-convex problem. Finally, we perform experiments for training deep convolutional neural networks on benchmark datasets. The experimental results not only confirm our theoretical analysis, but also demonstrate that the proposed method can achieve significant speedup without loss of accuracy.

1. Introduction

We have witnessed a series of breakthroughs in computer vision using deep convolutional neural networks (LeCun et al., 2015). Most neural networks are trained using stochastic gradient descent (SGD) or its variants in which the gradients of the networks are computed by backpropagation algorithm (Rumelhart et al., 1988). As shown in Figure 1, the backpropagation algorithm consists of two processes, the forward pass to compute prediction and the backward pass to compute gradient and update the model. After computing prediction in the forward pass, backpropagation algorithm requires propagating error gradients from the top (output layer) all the way back to the bottom (input layer). Therefore, in the backward pass, all layers, or more generally, modules, of the network are locked until their dependencies have executed.

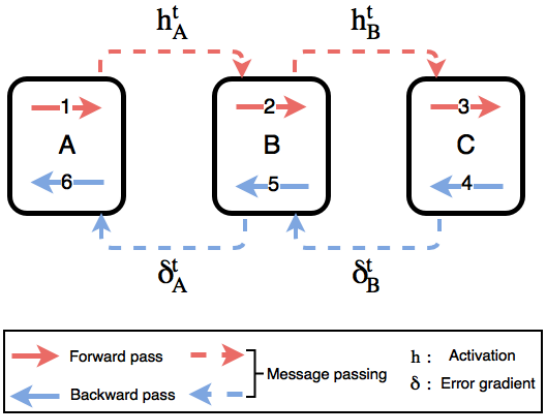


Figure 1. We split a multilayer feedforward neural network into three modules. Each module is a stack of layers. Backpropagation algorithm requires running forward pass (from 1 to 3) and backward pass (from 4 to 6) in sequential order. For example, module A cannot perform step 6 before receiving δ_A^t which is an output of step 5 in module B.

The backward locking constrains us from updating models in parallel and fully leveraging the computing resources. It has been shown in practice (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014; Szegedy et al., 2015; He et al., 2016; Huang et al., 2016) and in theory (Eldan & Shamir, 2016; Telgarsky, 2016; Bengio et al., 2009) that depth is one of the most critical factors contributing to the success of deep learning. From AlexNet with 8 layers (Krizhevsky et al., 2012) to ResNet-101 with more than one hundred layers (He et al., 2016), the forward and backward time grow from (4.31ms and 9.58ms) to (53.38ms and 103.06ms) when we train the networks on Titan X with the input size of $16 \times 3 \times 224 \times 224$ (Johnson, 2017). Therefore, parallelizing the backward pass can greatly reduce the training time when the backward time is about twice of the forward time. We

[1]Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA, United States. Correspondence to: Heng Huang <henghuanghh@gmail.com>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

can easily split a deep neural network into modules like Figure 1 and distribute them across multiple GPUs. However, because of the backward locking, all GPUs are idle before receiving error gradients from dependent modules in the backward pass.

There have been several algorithms proposed for breaking the backward locking. For example, (Jaderberg et al., 2016; Czarnecki et al., 2017) proposed to remove the lockings in backpropagation by employing additional neural networks to approximate error gradients. In the backward pass, all modules use the synthetic gradients to update weights of the model without incurring any delay. (Nøkland, 2016; Balduzzi et al., 2015) broke the local dependencies between successive layers and made all hidden layers receive error information from the output layer directly. In (Carreira-Perpinan & Wang, 2014; Taylor et al., 2016), the authors loosened the exact connections between layers by introducing auxiliary variables. In each layer, they imposed equality constraint between the auxiliary variable and activation, and optimized the new problem using Alternating Direction Method which is easy to parallel. However, for the convolutional neural network, the performances of all above methods are much worse than backpropagation algorithm when the network is deep.

In this paper, we focus on breaking the backward locking in backpropagtion algorithm for training feedforward neural networks, such that we can update models in parallel without loss of accuracy. The main contributions of our work are as follows:

- Firstly, we decouple the backpropagation using delayed gradients in Section 3 such that all modules of the network can be updated in parallel without backward locking.
- Then, we propose two stochastic algorithms using decoupled parallel backpropagation in Section 3 for deep learning optimization.
- We also provide convergence analysis for the proposed method in Section 4 and prove that it guarantees convergence to critical points for the non-convex problem.
- Finally, we perform experiments for training deep convolutional neural networks in Section 5, experimental results verifying that the proposed method can significantly speed up the training without loss of accuracy.

2. Backgrounds

We begin with a brief overview of the backpropagation algorithm for the optimization of neural networks. Suppose that we want to train a feedforward neural network with L layers, each layer taking an input h_{l-1} and producing an activation $h_l = F_l(h_{l-1}; w_l)$ with weight w_l. Letting d be the dimension of weights in the network, we have $w = [w_1, w_2, ..., w_L] \in \mathbb{R}^d$. Thus, the output of the network can be represented as $h_L = F(h_0; w)$, where h_0 denotes the input data x. Taking a loss function f and targets y, the training problem is as follows:

$$\min_{w=[w_1,...,w_L]} f(F(x; w), y). \tag{1}$$

In the following context, we use $f(w)$ for simplicity.

Gradients based methods are widely used for deep learning optimization (Robbins & Monro, 1951; Qian, 1999; Hinton et al., 2012; Kingma & Ba, 2014). In iteration t, we put a data sample $x_{i(t)}$ into the network, where $i(t)$ denotes the index of the sample. According to stochastic gradient descent (SGD), we update the weights of the network through:

$$w_l^{t+1} = w_l^t - \gamma_t \left[\nabla f_{l,x_{i(t)}}(w^t)\right]_l, \ \forall l \in \{1, 2, ..., L\} \tag{2}$$

where γ_t is the stepsize and $\nabla f_{l,x_{i(t)}}(w^t) \in \mathbb{R}^d$ is the gradient of the loss function (1) with respect to the weights at layer l and data sample $x_{i(t)}$, all the coordinates in other than layer l are 0. We always utilize backpropagation algorithm to compute the gradients (Rumelhart et al., 1988). The backpropagation algorithm consists of two passes of the network: in the forward pass, the activations of all layers are calculated from $l = 1$ to L as follows:

$$h_l^t = F_l(h_{l-1}^t; w_l); \tag{3}$$

in the backward pass, we apply chain rule for gradients and repeatedly propagate error gradients through the network from the output layer $l = L$ to the input layer $l = 1$:

$$\frac{\partial f(w^t)}{\partial w_l^t} = \frac{\partial h_l^t}{\partial w_l^t}\frac{\partial f(w^t)}{\partial h_l^t}, \tag{4}$$

$$\frac{\partial f(w^t)}{\partial h_{l-1}^t} = \frac{\partial h_l^t}{\partial h_{l-1}^t}\frac{\partial f(w^t)}{\partial h_l^t}, \tag{5}$$

where we let $\nabla f_{l,x_{i(t)}}(w^t) = \frac{\partial f(w^t)}{\partial w_l^t}$. From equations (4) and (5), it is obvious that the computation in layer l is dependent on the error gradient $\frac{\partial f(w^t)}{\partial h_l^t}$ from layer $l+1$. Therefore, the backward locking constrains all layers from updating before receiving error gradients from the dependent layers. When the network is very deep or distributed across multiple resources, the backward locking is the main bottleneck in the training process.

3. Decoupled Parallel Backpropagation

In this section, we propose to decouple the backpropagation algorithm using delayed gradients (DDG). Suppose we split a L-layer feedforward neural network to K modules, such that the weights of the network are divided into K groups. Therefore, we have $w = [w_{\mathcal{G}(1)}, w_{\mathcal{G}(2)}, ..., w_{\mathcal{G}(K)}]$ where $\mathcal{G}(k)$ denotes layer indices in the group k.

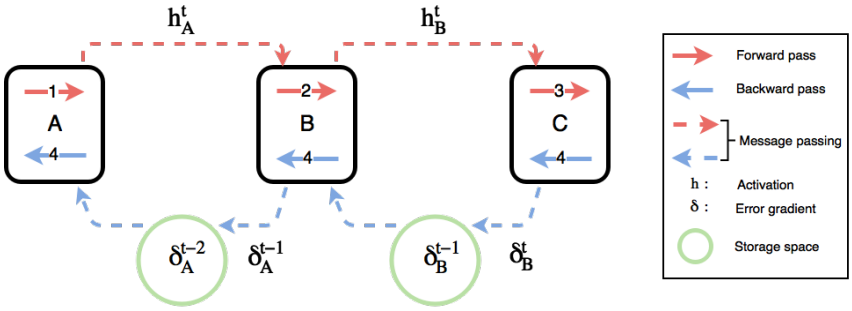


Figure 2. We split a multilayer feedforward neural network into three modules (A, B and C), where each module is a stack of layers. After executing the forward pass (from 1 to 3) to predict, our proposed method allows all modules to run backward pass (4) using delayed gradients without locking. Particularly, module A can perform the backward pass using the stale error gradient δ_A^{t-2}. Meanwhile, It also receives δ_A^{t-1} from module B for the update of the next iteration.

3.1. Backpropagation Using Delayed Gradients

In iteration t, data sample $x_{i(t)}$ is input to the network. We run the forward pass from module $k=1$ to $k=K$. In each module, we compute the activations in sequential order as equation (3). In the backward pass, all modules except the last one have delayed error gradients in store such that they can execute the backward computation without locking. The last module updates with the up-to-date gradients. In particular, module k keeps the stale error gradient $\frac{\partial f(w^{t-K+k})}{\partial h_{L_k}^{t-K+k}}$, where L_k denotes the last layer in module k. Therefore, the backward computation in module k is as follows:

$$\frac{\partial f(w^{t-K+k})}{\partial w_l^{t-K+k}} = \frac{\partial h_l^{t-K+k}}{\partial w_l^{t-K+k}} \frac{\partial f(w^{t-K+k})}{\partial h_l^{t-K+k}}, \quad (6)$$

$$\frac{\partial f(w^{t-K+k})}{\partial h_{l-1}^{t-K+k}} = \frac{\partial h_l^{t-K+k}}{\partial h_{l-1}^{t-K+k}} \frac{\partial f(w^{t-K+k})}{\partial h_l^{t-K+k}}. \quad (7)$$

where $\ell \in \mathcal{G}(k)$. Meanwhile, each module also receives error gradient from the dependent module for further computation. From (6) and (7), we can know that the stale error gradients in all modules are of different time delay. From module $k=1$ to $k=K$, their corresponding time delays are from $K-1$ to 0. Delay 0 indicates that the gradients are up-to-date. In this way, we break the backward locking and achieve parallel update in the backward pass. Figure 2 shows an example of the decoupled backpropagation, where error gradients $\delta := \frac{\partial f(w)}{\partial h}$.

3.2. Speedup of Decoupled Parallel Backpropagation

When $K=1$, there is no time delay and the proposed method is equivalent to the backpropagation algorithm. When $K \neq 1$, we can distribute the network across multiple GPUs and fully leverage the computing resources. Table 1 lists the computation time when we sequentially allocate the network across K GPUs. When $\mathcal{T}_F$ is necessary to compute accurate predictions, we can accelerate the training by reducing the backward time. Because $\mathcal{T}_B$ is much large than $\mathcal{T}_F$, we can achieve huge speedup even K is small.

Table 1. Comparisons of computation time when the network is sequentially distributed across K GPUs. $\mathcal{T}_F$ and $\mathcal{T}_B$ denote the forward and backward time for backpropagation algorithm.

Method	Computation Time
Backpropagation	$\mathcal{T}_F + \mathcal{T}_B$
DDG	$\mathcal{T}_F + \frac{\mathcal{T}_B}{K}$

Relation to model parallelism: Model parallelism usually refers to filter-wise parallelism (Yadan et al., 2013). For example, we split a convolutional layer with N filters into two GPUs, each part containing $\frac{N}{2}$ filters. Although the filter-wise parallelism accelerates the training when we distribute the workloads across multiple GPUs, it still suffers from the backward locking. We can think of DDG algorithm as layer-wise parallelism. It is also easy to combine filter-wise parallelism with layer-wise parallelism for further speedup.

3.3. Stochastic Methods Using Delayed Gradients

After computing the gradients of the loss function with respect to the weights of the model, we update the model using delayed gradients. Letting $\nabla f_{\mathcal{G}(k),x_{i(t-K+k)}}\left(w^{t-K+k}\right) :=$

$$\begin{cases} \sum_{l\in\mathcal{G}(k)} \frac{\partial f(w^{t-K+k})}{\partial w_l^{t-K+k}} & \text{if } t-K+k \geq 0 \\ 0 & \text{otherwise} \end{cases}, \quad (8)$$

for any $k \in \{1, 2, ..., K\}$, we update the weights in module k following SGD:

$$w_{\mathcal{G}(k)}^{t+1} = w_{\mathcal{G}(k)}^t - \gamma_t [\nabla f_{\mathcal{G}(k),x_{i(t-K+k)}}\left(w^{t-K+k}\right)]_{\mathcal{G}(k)}. \quad (9)$$

where γ_t denotes stepsize. Different from SGD, we update the weights with delayed gradients. Besides, the delayed iteration $(t-K+k)$ for group k is also deterministic. We summarize the proposed method in Algorithm 1.

Algorithm 1 SGD-DDG

Require:
Initial weights $w^0 = [w^0_{\mathcal{G}(1)}, ..., w^0_{\mathcal{G}(K)}] \in \mathbb{R}^d$;
Stepsize sequence $\{\gamma_t\}$;
for $t = 0, 1, 2, \ldots, T-1$ **do**
  for $k = 1, \ldots, K$ **in parallel do**
    Compute delayed gradient:
$$g^t_k \leftarrow \left[\nabla f_{\mathcal{G}(k), x_{i(t-K+k)}}\left(w^{t-K+k}\right)\right]_{\mathcal{G}(k)};$$
    Update weights:
$$w^{t+1}_{\mathcal{G}(k)} \leftarrow w^t_{\mathcal{G}(k)} - \gamma_t \cdot g^t_k;$$
  end for
end for

Moreover, we can also apply the delayed gradients to other variants of SGD, for example Adam in Algorithm 2. In each iteration, we update the weights and moment vectors with delayed gradients. We analyze the convergence for Algorithm 1 in Section 4, which is the basis of analysis for other methods.

4. Convergence Analysis

In this section, we establish the convergence guarantees to critical points for Algorithm 1 when the problem is non-convex. Analysis shows that our method admits similar convergence rate to vanilla stochastic gradient descent (Bottou et al., 2016). Throughout this paper, we make the following commonly used assumptions:

Assumption 1 ***(Lipschitz-continuous gradient)*** *The gradient of $f(w)$ is Lipschitz continuous with Lipschitz constant $L > 0$, such that $\forall w, v \in \mathbb{R}^d$:*

$$\|\nabla f(w) - \nabla f(v)\|_2 \leq L\|w - v\|_2 \tag{10}$$

Assumption 2 ***(Bounded variance)*** *To bound the variance of the stochastic gradient, we assume the second moment of the stochastic gradient is upper bounded, such that there exists constant $M \geq 0$, for any sample x_i and $\forall w \in \mathbb{R}^d$:*

$$\|\nabla f_{x_i}(w)\|_2^2 \quad \leq \quad M \tag{11}$$

Because of the unnoised stochastic gradient $\mathbb{E}\left[\nabla f_{x_i}(w)\right] = \nabla f(w)$ and the equation regarding variance $\mathbb{E}\|\nabla f_{x_i}(w) - \nabla f(w)\|_2^2 = \mathbb{E}\|\nabla f_{x_i}(w)\|_2^2 - \|\nabla f(w)\|_2^2$, the variance of the stochastic gradient is guaranteed to be less than M.

Under Assumption 1 and 2, we obtain the following lemma about the sequence of objective functions.

Lemma 1 *Assume Assumption 1 and 2 hold. In addition, we let $\sigma := \max_t \frac{\gamma_{\max\{0, t-K+1\}}}{\gamma_t}$ and $M_K = KM + \sigma K^4 M$.*

Algorithm 2 Adam-DDG

Require:
Initial weights: $w^0 = [w^0_{\mathcal{G}(1)}, ..., w^0_{\mathcal{G}(K)}] \in \mathbb{R}^d$;
Stepsize: γ; Constant $\epsilon = 10^{-8}$;
Exponential decay rates: $\beta_1 = 0.9$ and $\beta_2 = 0.999$;
First moment vector: $m^0_{\mathcal{G}(k)} \leftarrow 0, \forall k \in \{1, 2, ..., K\}$;
Second moment vector: $v^0_{\mathcal{G}(k)} \leftarrow 0, \forall k \in \{1, 2, ..., K\}$;
for $t = 0, 1, 2, \ldots, T-1$ **do**
  for $k = 1, \ldots, K$ **in parallel do**
    Compute delayed gradient:
$$g^t_k \leftarrow \left[\nabla f_{\mathcal{G}(k), x_{i(t-K+k)}}\left(w^{t-K+k}\right)\right]_{\mathcal{G}(k)};$$
    Update biased first moment estimate:
$$m^{t+1}_{\mathcal{G}(k)} \leftarrow \beta_1 \cdot m^t_{\mathcal{G}(k)} + (1-\beta_1) \cdot g^t_k$$
    Update biased second moment estimate:
$$v^{t+1}_{\mathcal{G}(k)} \leftarrow \beta_2 \cdot v^t_{\mathcal{G}(k)} + (1-\beta_2) \cdot (g^t_k)^2$$
    Compute bias-correct first moment estimate:
$$\hat{m}^{t+1}_{\mathcal{G}(k)} \leftarrow m^{t+1}_{\mathcal{G}(k)} / (1-\beta_1^{t+1})$$
    Compute bias-correct second moment estimate:
$$\hat{v}^{t+1}_{\mathcal{G}(k)} \leftarrow v^{t+1}_{\mathcal{G}(k)} / (1-\beta_2^{t+1})$$
    Update weights:
$$w^{t+1}_{\mathcal{G}(k)} \leftarrow w^t_{\mathcal{G}(k)} - \gamma \cdot \hat{m}^{t+1}_{\mathcal{G}(k)} / \left(\sqrt{\hat{v}^{t+1}_{\mathcal{G}(k)}} + \epsilon\right)$$
  end for
end for

The iterations in Algorithm 1 satisfy the following inequality, for all $t \in \mathbb{N}$:

$$\mathbb{E}\left[f(w^{t+1})\right] - f(w^t) \leq -\frac{\gamma_t}{2}\left\|\nabla f(w^t)\right\|_2^2 + \gamma_t^2 L M_K \tag{12}$$

From Lemma 1, we can observe that the expected decrease of the objective function is controlled by the stepsize γ_t and M_K. Therefore, we can guarantee that the values of objective functions are decreasing as long as the stepsizes γ_t are small enough such that the right-hand side of (12) is less than zero. Using the lemma above, we can analyze the convergence property for Algorithm 1.

4.1. Fixed Stepsize γ_t

Firstly, we analyze the convergence for Algorithm 1 when γ_t is fixed and prove that the learned model will converge sub-linearly to the neighborhood of the critical points.

Theorem 1 *Assume Assumption 1 and 2 hold and the fixed stepsize sequence $\{\gamma_t\}$ satisfies $\gamma_t = \gamma$ and $\gamma L \leq 1, \forall t \in \{0, 1, ..., T-1\}$. In addition, we assume w^* to be the optimal solution to $f(w)$ and let $\sigma = 1$ such that $M_K = KM + K^4 M$. Then, the output of Algorithm 1 satisfies that:*

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\|\nabla f(w^t)\right\|_2^2 \leq \frac{2\left(f(w^0) - f(w^*)\right)}{\gamma T} + 2\gamma L M_K \tag{13}$$

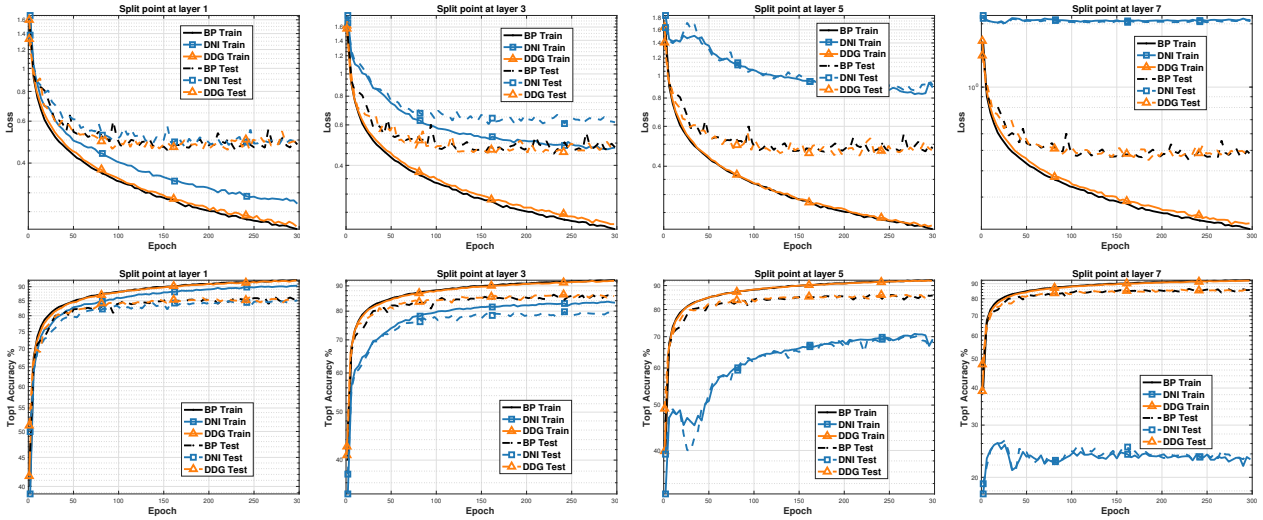


Figure 3. Training and testing curves regarding epochs for ResNet-8 on CIFAR-10. **Upper:** Loss function values regarding epochs; **Bottom:** Top 1 classification accuracies regarding epochs. We split the network into two modules such that there is only one split point in the network for DNI and DDG.

In Theorem 1, we can observe that when $T \to \infty$, the average norm of the gradients is upper bounded by $2\gamma LM_K$. The number of modules K affects the value of the upper bound. Selecting a small stepsize γ allows us to get better neighborhood to the critical points, however it also seriously decreases the speed of convergence.

4.2. Diminishing Stepsize γ_t

In this section, we prove that Algorithm 1 with diminishing stepsizes can guarantee the convergence to critical points for the non-convex problem.

Theorem 2 *Assume Assumption 1 and 2 hold and the diminishing stepsize sequence $\{\gamma_t\}$ satisfies $\gamma_t = \frac{\gamma_0}{1+t}$ and $\gamma_t L \leq 1, \forall t \in \{0, 1, ..., T-1\}$. In addition, we assume w^* to be the optimal solution to $f(w)$ and let $\sigma = K$ such that $M_K = KM + K^5 M$. Setting $\Gamma_T = \sum_{t=0}^{T-1} \gamma_t$, then the output of Algorithm 1 satisfies that:*

$$\frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \gamma_t \mathbb{E} \left\| \nabla f(w^t) \right\|_2^2 \leq \frac{2\left(f(w^0) - f(w^*)\right)}{\Gamma_T} + \frac{2\sum_{t=0}^{T-1} \gamma_t^2 LM_K}{\Gamma_T} \quad (14)$$

Corollary 1 *Since $\gamma_t = \frac{\gamma_0}{t+1}$, the stepsize requirements in (Robbins & Monro, 1951) are satisfied that:*

$$\lim_{T\to\infty} \sum_{t=0}^{T-1} \gamma_t = \infty \quad \textit{and} \quad \lim_{T\to\infty} \sum_{t=0}^{T-1} \gamma_t^2 < \infty. \quad (15)$$

Therefore, according to Theorem 2, when $T \to \infty$, the right-hand side of (14) converges to 0.

Corollary 2 *Suppose w^s is chosen randomly from $\{w^t\}_{t=0}^{T-1}$ with probabilities proportional to $\{\gamma_t\}_{t=0}^{T-1}$. According to Theorem 2, we can prove that Algorithm 1 guarantees convergence to critical points for the non-convex problem:*

$$\lim_{s\to\infty} \mathbb{E} \|\nabla f(w^s)\|_2^2 = 0 \quad (16)$$

5. Experiments

In this section, we experiment with ResNet (He et al., 2016) on image classification benchmark datasets: CIFAR-10 and CIFAR-100 (Krizhevsky & Hinton, 2009). In section 5.1, we evaluate our method by varying the positions and the number of the split points in the network; In section 5.2 we use our method to optimize deeper neural networks and show that its performance is as good as the performance of backpropagation; finally, we split and distribute the ResNet-110 across GPUs in Section 5.3, results showing that the proposed method achieves a speedup of two times without loss of accuracy.

Implementation Details: We implement DDG algorithm using PyTorch library (Paszke et al., 2017). The trained network is split into K modules where each module is running on a subprocess. The subprocesses are spawned using multiprocessing package [1] such that we can fully leverage

[1] https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing

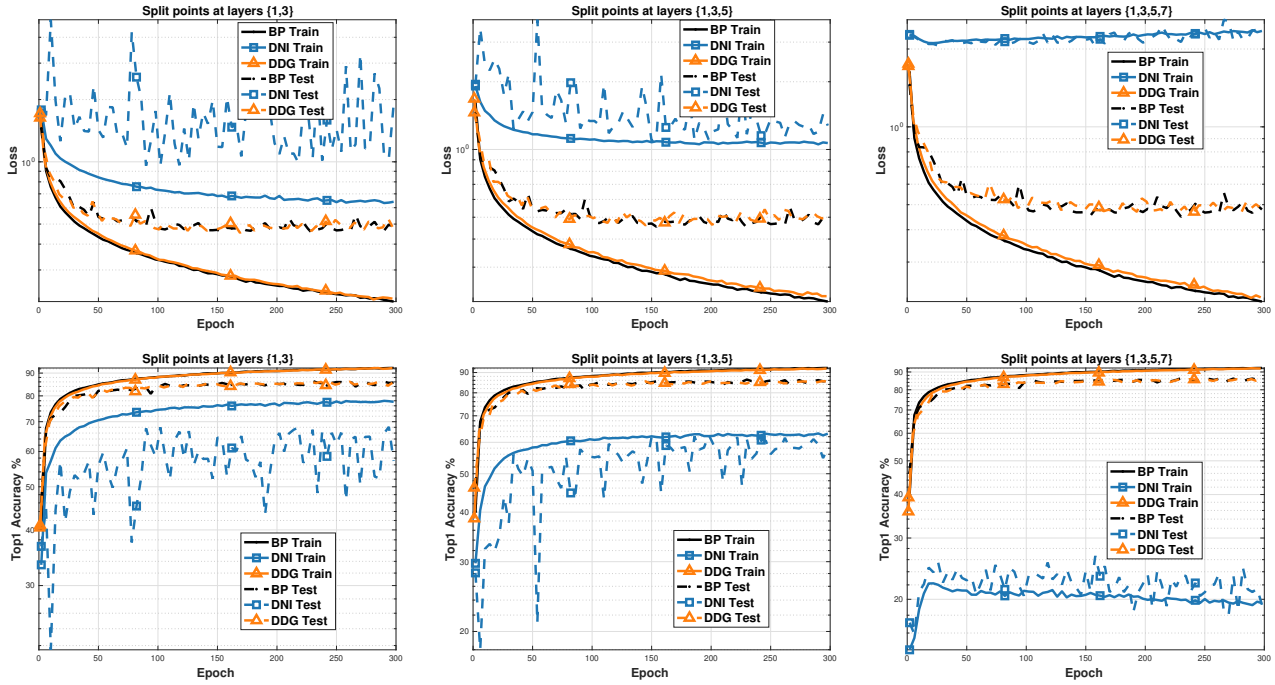


Figure 4. Training and testing curves regarding epochs for ResNet-8 on CIFAR-10. **Upper:** Loss function values regarding epochs; **Bottom:** Top1 classification accuracies regarding epochs. For DNI and DDG, the number of split points in the network ranges from 2 to 4.

multiple processors on a given machine. Running modules on different subprocesses make the communication very difficult. To make the communication fast, we utilize the shared memory objects in the multiprocessing package. As in Figure 2, every two adjacent modules share a pair of activation (h) and error gradient (δ).

5.1. Comparison of BP, DNI and DDG

In this section, we train ResNet-8 on CIFAR-10 on a single Titan X GPU. The architecture of the ResNet-8 is in Table 2. All experiments are run for 300 epochs and optimized using Adam optimizer (Kingma & Ba, 2014) with a batch size of 128. The stepsize is initialized at 1×10^{-3}. We augment the dataset with random cropping, random horizontal flipping and normalize the image using mean and standard deviation. There are three compared methods in this experiment:

- **BP**: Adam optimizer in Pytorch uses backpropagation algorithm with data parallelism (Rumelhart et al., 1988) to compute gradients.
- **DNI**: Decoupled neural interface (DNI) in (Jaderberg et al., 2016). Following (Jaderberg et al., 2016), the synthetic network is a stack of three convolutional layers with L 5×5 filters with resolution preserving padding. The filter depth L is determined by the position of DNI. We also input label information into the synthetic network to increase final accuracy.

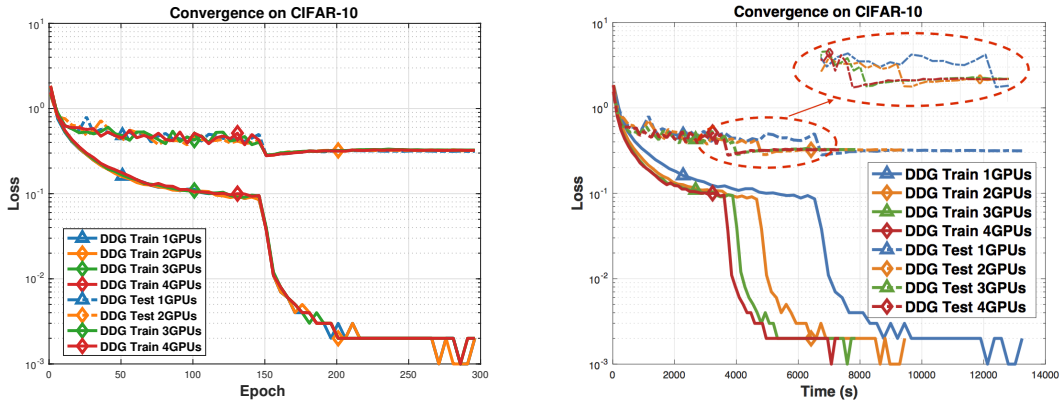
Table 2. Architectural details. **Units** denotes the number of residual units in each group. Each unit is a basic residual block without bottleneck. **Channels** indicates the number of filters used in each unit in each group.

Architecture	Units	Channels
ResNet-8	1-1-1	16-16-32-64
ResNet-56	9-9-9	16-16-32-64
ResNet-110	18-18-18	16-16-32-64

- **DDG**: Adam optimizer using delayed gradients in Algorithm 2.

Impact of split position (depth). The position (depth) of the split points determines the number of layers using delayed gradients. Stale or synthetic gradients will induce noises in the training process, affecting the convergence of the objective. Figure 3 exhibits the experimental results when there is only one split point with varying positions. In the first column, we know that all compared methods have similar performances when the split point is at layer 1. DDG performs consistently well when we place the split point at deeper positions 3, 5 or 7. On the contrary, the performance of DNI degrades as we vary the positions and it cannot even converge when the split point is at layer 7.

Impact of the number of split points. From equation (7), we know that the maximum time delay is determined by



(a) (b)

Figure 5. Training and testing loss curves for ResNet-110 on CIFAR-10 using multiple GPUs. **(5a)** Loss function value regarding epochs. **(5b)** Loss function value regarding computation time.

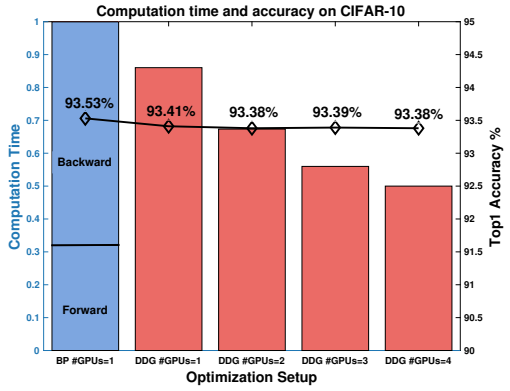


Figure 6. Computation time and the best Top 1 accuracy for ResNet-110 on the test data of CIFAR-10. The most left bar denotes the computation time using backpropagation algorithm on a GPU, where the forward time accounts for about 32%. We normalize the computation time of all optimization settings using the amount of time required by backpropagation.

the number of modules K. Theorem 2 also shows that K affects the convergence rate. In this experiment, we vary the number of split points in the network from 2 to 4 and plot the results in Figure 4. It is easy to observe that DDG performs as well as BP, regardless of the number of split points in the network. However, DNI is very unstable when we place more split points, and cannot even converge sometimes.

5.2. Optimizing Deeper Neural Networks

In this section, we employ DDG to optimize two very deep neural networks (ResNet-56 and ResNet-110) on CIFAR-10 and CIFAR-100. Each network is split into two modules at the center. We use SGD with the momentum of 0.9 and the stepsize is initialized to 0.01. Each model is trained for 300 epochs and the stepsize is divided by a factor of 10 at 150 and 225 epochs. The weight decay constant is set to 5×10^{-4}. We perform the same data augmentation as in section 5.1. Experiments are run on a single Titan X GPU.

Table 3. The best Top 1 classification accuracy (%) for ResNet-56 and ResNet-110 on the test data of CIFAR-10 and CIFAR-100.

Architecture	CIFAR-10		CIFAR-100	
	BP	DDG	BP	DDG
ResNet-56	**93.12**	93.11	69.79	**70.17**
ResNet-110	**93.53**	93.41	**71.90**	71.39

Figure 7 presents the experimental results of BP and DDG. We do not compare DNI because its performance is far worse when models are deep. Figures in the first column present the convergence of loss regarding epochs, showing that DDG and BP admit similar convergence rates. We can also observe that DDG converges faster when we compare the loss regarding computation time in the second column of Figure 7. In the experiment, the "Volatile GPU Utility" is about 70% when we train the models with BP. Our method runs on two subprocesses such that it fully leverages the computing capacity of the GPU. We can draw similar conclusions when we compare the Top 1 accuracy in the third and fourth columns of Figure 7. In Table 3, we list the best Top 1 accuracy on the test data of CIFAR-10 and CIFAR-100. We can observe that DDG can obtain comparable or better accuracy even when the network is deep.

5.3. Scaling the Number of GPUs

In this section, we split ResNet-110 into K modules and allocate them across K Titan X GPUs sequentially. We do

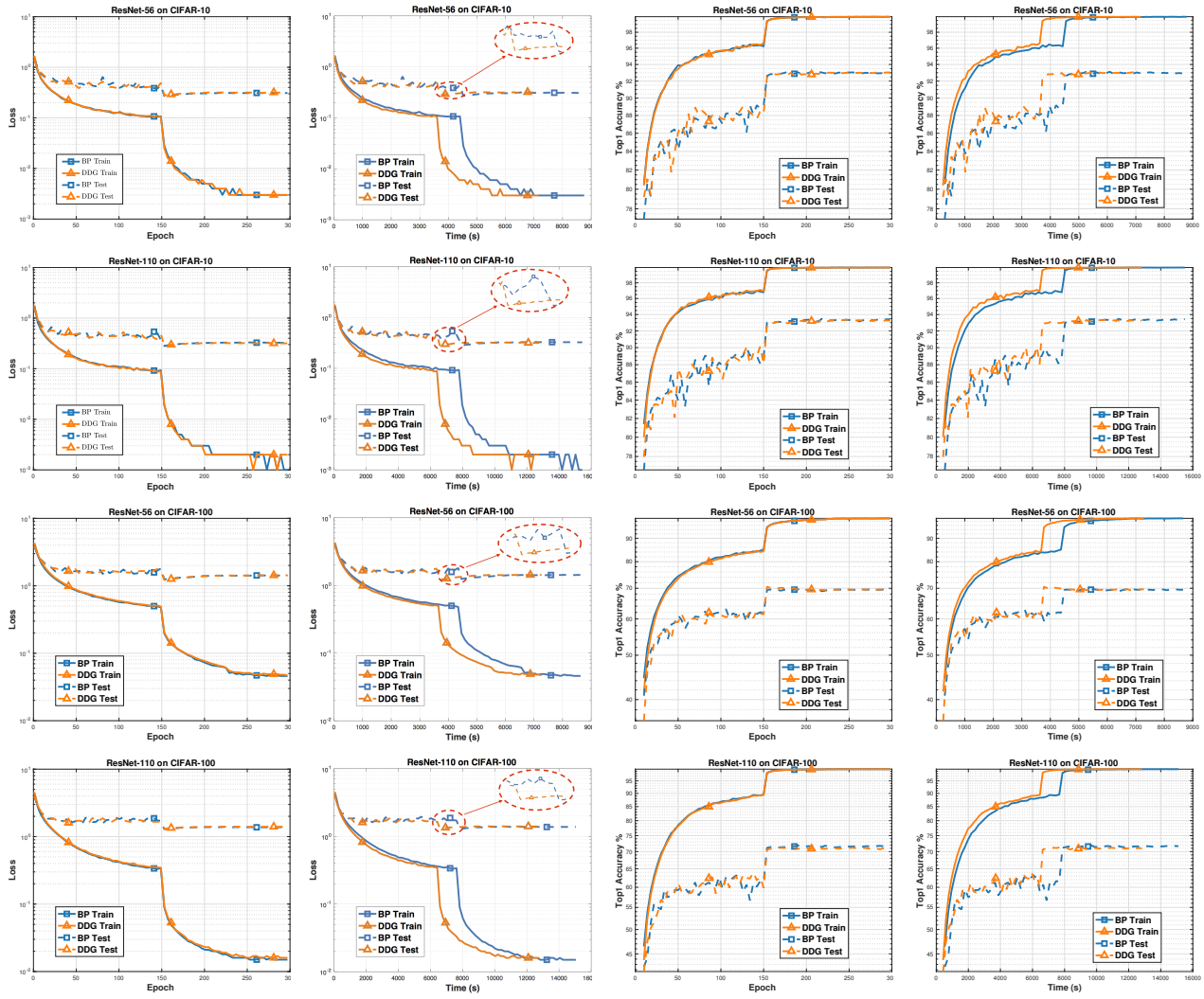


Figure 7. Training and testing curves for ResNet-56 and ResNet-110 on CIFAR-10 and CIFAR-100. **Column** 1 **and** 2 present the loss function value regrading epochs and computation time respectively; **Column** 3 **and** 4 present the Top 1 classification accuracy regrading epochs and computation time. For DDG, there is only one split point at the center of the network.

not consider filter-wise model parallelism in this experiment. The selections of the parameters in the experiment are similar to Section 5.2. From Figure 5, we know that training networks in multiple GPUs does not affect the convergence rate. For comparison, we also count the computation time of backpropagation algorithm on a single GPU. The computation time is worse when we run backpropagation algorithm on multiple GPUs because of the communication overhead. In Figure 6, we can observe that forward time only accounts for about 32% of the total computation time for backpropagation algorithm. Therefore, backward locking is the main bottleneck. In Figure 6, it is obvious that when we increase the number of GPUs from 2 to 4, our method reduces about 30% to 50% of the total computation time. In other words, DDG achieves a speedup of about 2 times without loss of accuracy when we train the networks across 4 GPUs.

6. Conclusion

In this paper, we propose decoupled parallel backpropagation algorithm, which breaks the backward locking in backpropagation algorithm using delayed gradients. We then apply the decoupled parallel backpropagation to two stochastic methods for deep learning optimization. In the theoretical section, we also provide convergence analysis and prove that the proposed method guarantees convergence to critical points for the non-convex problem. Finally, we perform experiments on deep convolutional neural networks, results verifying that our method can accelerate the training significantly without loss of accuracy.

Acknowledgement

This work was partially supported by U.S. NIH R01 AG049371, NSF IIS 1302675, IIS 1344152, DBI 1356628, IIS 1619308, IIS 1633753.

References

Balduzzi, D., Vanchinathan, H., and Buhmann, J. M. Kickback cuts backprop's red-tape: Biologically plausible credit assignment in neural networks. In *AAAI*, pp. 485–491, 2015.

Bengio, Y. et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

Carreira-Perpinan, M. and Wang, W. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, pp. 10–19, 2014.

Czarnecki, W. M., Świrszcz, G., Jaderberg, M., Osindero, S., Vinyals, O., and Kavukcuoglu, K. Understanding synthetic gradients and decoupled neural interfaces. *arXiv preprint arXiv:1703.00522*, 2017.

Eldan, R. and Shamir, O. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pp. 907–940, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hinton, G., Srivastava, N., and Swersky, K. Neural networks for machine learning-lecture 6a-overview of mini-batch gradient descent, 2012.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., and Kavukcuoglu, K. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016.

Johnson, J. Benchmarks for popular cnn models. https://github.com/jcjohnson/cnn-benchmarks, 2017.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature*, 521(7553):436–444, 2015.

Nøkland, A. Direct feedback alignment provides learning in deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 1037–1045, 2016.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Qian, N. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

Robbins, H. and Monro, S. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., and Goldstein, T. Training neural networks without gradients: A scalable admm approach. In *International Conference on Machine Learning*, pp. 2722–2731, 2016.

Telgarsky, M. Benefits of depth in neural networks. *arXiv preprint arXiv:1602.04485*, 2016.

Yadan, O., Adams, K., Taigman, Y., and Ranzato, M. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*, 2013.