CONTINUOUS PROPAGATION: LAYER-PARALLEL TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Continuous propagation is a parallel technique for training deep neural networks with batch size one at full utilization of a multiprocessor system. It enables spatially distributed computations on emerging deep learning hardware accelerators that do not impose programming limitations of contemporary GPUs. The algorithm achieves model parallelism along the depth of a deep network. The method is based on the continuous representation of the optimization process and enables sustained gradient generation during all phases of computation. We demonstrate that in addition to its increased concurrency, continuous propagation improves the convergence rate of state of the art methods while matching their accuracy.

1 INTRODUCTION

Stochastic gradient descent (SGD) with back-propagation has become a ubiquitous algorithm for training deep neural networks. Learning via gradient descent is inherently sequential because each update in parameter space requires a gradient measurement at the new point in parameter space.

$$\Theta_t = \Theta_{t-1} - \alpha \frac{\partial \mathcal{L}}{\partial \Theta}\bigg|_{\Theta_{t-1}} \tag{1}$$

One technique for parallelization is gradient averaging over a mini-batch but it does nothing to speed up the rate of sequential parameter updates. As mini-batch size increases, gradient noise is reduced. Beyond a point this causes poor generalization (Sec. 2 and Sec. 3.1).

While deep neural networks have many layers, it is not possible to process them in parallel because information passes sequentially through the layers of a network.

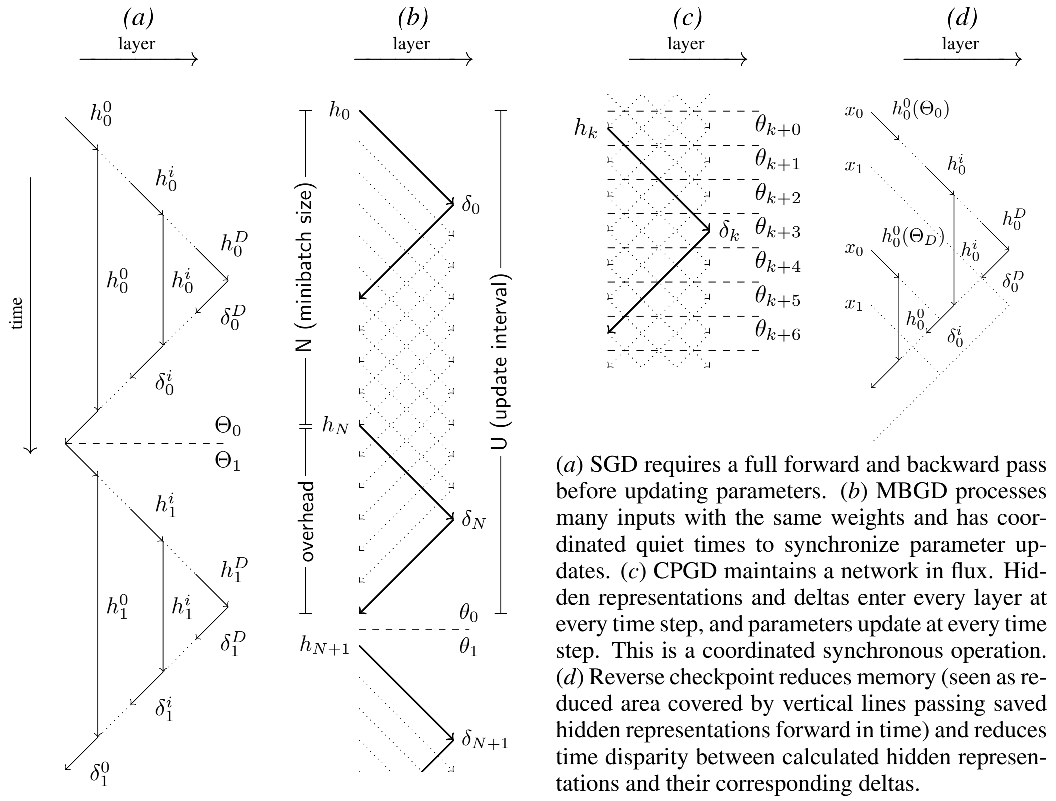
To overcome these limitations we propose *continuous propagation*—an approach that allows parameters in all layers to update simultaneously while samples propagate through network layers.

Information flows and concurrency in training algorithms can be visualized with pipeline diagrams. Figure 1(a) shows that gradient-descent algorithms require a full forward and backward pass through the network to compute a gradient estimate. Use of differential equations replaces sequential dependency with a continuous model that has *sustained gradient generation*, where all layers compute gradients at all times rather than having modal forward and backward phases (Fig. 1(c), Sec. 4).

The main advantage of this approach is that it enables *layer parallelism* by allowing each layer to learn concurrently with others without explicit synchronization. As a result, parallelization along the depth of a network allows more computing resources to be applied to training. This computational framework differs from a GPU-optimized implementation, where computations, even though performed in parallel, happen sequentially in terms of layer utilization.

While the minibatch approach waits for a group of gradient measurements to be completed, in continuous propagation the gradient estimate is used as soon as it becomes available leading to statistically superior behavior. This incremental contribution of each observation is matched by a continuous representation of the problem that allows us to formalize this approach and enables convergence analysis.

The theoretical foundation of this technique relies on a continuous differential representation of the learning process (Sec. 3.2). It is based on the observation that the time iteration equation of gradient-



(*a*) SGD requires a full forward and backward pass before updating parameters. (*b*) MBGD processes many inputs with the same weights and has coordinated quiet times to synchronize parameter updates. (*c*) CPGD maintains a network in flux. Hidden representations and deltas enter every layer at every time step, and parameters update at every time step. This is a coordinated synchronous operation. (*d*) Reverse checkpoint reduces memory (seen as reduced area covered by vertical lines passing saved hidden representations forward in time) and reduces time disparity between calculated hidden representations and their corresponding deltas.

Figure 1: Pipeline diagrams for forward and backward propagation.

descent learning algorithm (1) can be viewed as a numerical integration-approximation method of the differential system

$$\dot{\Theta} = -\frac{\partial \mathcal{L}}{\partial \Theta} \,. \tag{2}$$

Theoretical results include a formal convergence proof of the method (sec. 4.1, Appendix).

Experiments with SVHN, CIFAR-10, and CIFAR-100 image classification sets show that in addition to increased concurrency compared to state of the art algorithms, continuous propagation offers comparable accuracy and improved convergence rate expressed in epochs of training (Sec. 5).

2 Related Work

Optimization in the presence of stochastic variables was introduced in the preeminent work by Robbins & Monro (1951). Robbins' *stochastic approximation* finds roots of the expected value of an unknown stochastic function. Robbins identified efficiency as the convergence rate of the sequence of stochastic observations.

Stochastic gradient descent (SGD) extends the original gradient-descent (GD) method with a process of stochastic observations of a Lyapunov (cost) function and its gradient. A principal advantage of SGD is that the effort used to obtain a gradient is fixed and independent of the size of the input domain. This independence also allows extension to infinite training sets. In exchange, notions of convergence also have to be made probabilistic (Bottou, 1998).

Mini-batch gradient descent (MBGD) was first introduced as a hybrid approach between SGD and GD Møller (1993) in order to enjoy both the speed advantages of SGD and the convergence guarantees of GD. More recently, mini-batches are important in amortizing the cost of context switching in GPU-accelerated computation. When the state associated with model parameters (weights, gradients, momentum, etc.) are too large to fit in cache memory, time efficiency is gained by reusing the parameters that can fit in cache across a mini-batch of activations.

Asynchronous SGD (ASGD) is a parallel multi-GPU algorithm for mini-batch learning (Zhang et al., 2013; Dean et al., 2012). Its primary gain is in operational efficiency in dealing with a cluster of machines. ASGD eliminates cluster synchronization at the end of every gradient computation, which accommodates machine faults, but also causes some parameter differences between worker nodes. Synthetic Gradients (Czarnecki et al., 2017) is another approach to train a deep network asynchronously by modeling the error gradients at each layer independently.

Thematic in both MBGD and ASGD is the recovery of otherwise idle computational resources to perform additional gradient measurements without a parameter update. While the problem of recruiting many machines to assist in gradient computation is solved by relying on increasingly large batches in the SGD algorithm, it is known that mini-batch training is an inefficient use of this computing power Wilson & Martinez (2003), Keskar et al. (2016).

Recent research has explored fundamental changes to GD. Difference target propagation (Lee et al., 2015) eliminates back-propagation entirely by learning bidirectional inverse mappings between adjacent layers. Feedback alignment (Lillicrap et al., 2014), in contrast, uses fixed random matricies for the backprop phase. It depends on feedback to maintain parameters as approximate pseudo-inverses of the fixed random matricies.

Current research is not in computing more-accurate gradients but in being able to scale to larger models. For example, Shazeer et al. (2017) set a goal of training a network of one trillion parameters. Such large models make it even more important to develop efficient parallelization methods.

3 Theory

3.1 Advantages of Model Parallelism

Parallelizing a computation involves spreading its evaluation over separate computational units operating simultaneously. In consideration of deep neural networks, there are two principal ways this can be imagined. In a *model-parallel* regime, separate workers simultaneously evaluate the same network input using distinct model parameters. Conversely, in a *data-parallel* regime, separate workers simultaneously evaluate distinct network inputs using the same formal model parameters. Current scaling efforts use fine-grain model parallelism in block matrix–vector multiplication and coarse-grain data parallelism across layers and among workers in a cluster.

Parameter values used in parallel cannot have sequential dependencies. Therefore, coordination is necessary among workers responsible for the same parameters. Strict synchronization ensures identical values of corresponding parameters; loose synchronization allows some discrepancy.

While there are distinct ways to implement data parallelism, they all share the attribute that multiple gradients are evaluated at points independent of each other's outcomes. That is, the point of evaluation of the gradient is not able to benefit from learning based on the gradients that are evaluated in parallel. Since mini-batches capture this attribute well, we use increasing mini-batch sizes as a proxy for all forms of increasing data parallelism.[1]

In addition to discovering sharp minima in parameter space that lead to bad generalization, increasing mini-batch size is ineffective for scaling for three other reasons.

First, given that all gradient measurements provide independent estimates of the true gradient,

$$\left.\frac{\partial \mathcal{L}}{\partial \Theta}\right|_{x\sim X} = \mathop{\mathbb{E}}_{x\sim X}\left[\frac{\partial \mathcal{L}}{\partial \Theta}\right] + \mathcal{N}(0,\sigma)\ ; \tag{3}$$

having n samples increases the accuracy of our gradient estimate to

$$\frac{1}{n}\sum_{i=1}^{n}\left.\frac{\partial \mathcal{L}}{\partial \Theta}\right|_{x_i\sim X} = \mathop{\mathbb{E}}_{x\sim X}\left[\frac{\partial \mathcal{L}}{\partial \Theta}\right] + \mathcal{N}(0,\sigma/\sqrt{n})\ . \tag{4}$$

[1] Specially designed schemes may be able to exploit additional information from data parallelism, such as gradient variance or cost curvature. We specifically preclude second-order methods from consideration.

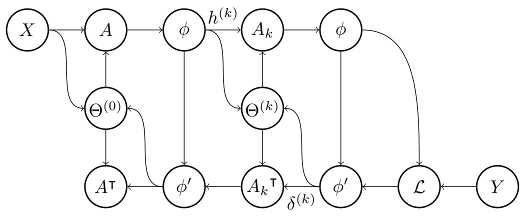


Figure 2: Feed-forward neural network

Table 1: Equations for deep network nodes

Node	Type	Function
A	Forward	$h^{(l)} = \Theta^{(l)} h^{(l-1)}$
$A^\intercal$	Backward	$\delta^{(l)} = \Theta^{(l)\intercal} \delta^{(l+1)}$
Θ	Update	$\Theta^{(l)} = \Theta_0 - \int_0^t h^{(l-1)} \times \delta^{(l)} dt$
Φ	Activation	$h^{(l)} = \phi(h^{(l-1)})$
Φ'	Tangent	$\delta^{(l)} = \phi'(h^{(l-1)}) \odot \delta^{(l+1)}$
X	Input*	$x = X_{\lfloor t/\alpha \rfloor}$
Y	Label	$y(x)$
$\mathcal{L}$	Loss	$\mathcal{L}(h^{(D)}, y)$

* $\{X_k \sim X\}$ is a sequence of random observations of X. $x(t)$ is piecewise constant.

For the same computational effort we could have taken n steps, each with a step size equal to $(\alpha/\sqrt{n})\frac{\partial \mathcal{L}}{\partial \Theta}$. Since the batch step size stayed within a neighborhood of Θ well approximated by its first-order Taylor expansion, each of the n steps of the SGD algorithm stays within this same neighborhood. However, we have now proceeded through a total distance of $(\alpha n/\sqrt{n})\frac{\partial \mathcal{L}}{\partial \Theta}$, so we are more efficient by approximately $\sqrt{n}$ (Goodfellow et al., 2016, section 8.1.3).

Second, our objective function $\mathcal{L}$ is nonlinear. Given this, the accuracy of the first-order gradient provides limited utility, owing to the high curvature of the loss surface. Beyond this, the ability to use faster learning rate α by employing larger batch size is fruitless because it is bound to be inaccurate, and an update in parameter space is necessary to assess a gradient at a new location.

Finally, much of the computational efficiency of SGD comes from the noise realized by replacing $\mathbb{E}_{x \sim X}$ with a sample estimate. Computing a larger sample estimate undoes this efficiency. The sampling noise causes model regularization. Beyond a certain point, the gain in accuracy exceeds the desired regularization, and synthetic noise must be added back to the computation to achieve the desired results.

3.2 Formulation as Differential Equations

Deep neural networks are high-dimensional parameterized functions $f(x;\Theta)$, which are often expressed as directed acyclic graphs. The back-propagation algorithm, however, is best expressed by a *cyclic* graph (Fig. 2).

The cycle in the graph is a feedback iteration: the gradients produced by the first full network evaluation change the weights used in the next iteration. This is because the iterative algorithms are discrete approximations of a continuous differential system:

$$\dot{\Theta} = -\mathop{\mathbb{E}}_{x \sim X}\left[\frac{\partial \mathcal{L}(f(x;\Theta), y(x))}{\partial \Theta}\right] + \mathcal{G}(\sigma(t)) \,, \tag{5}$$

where $\mathcal{G}$ is an unbiased continuous-noise process, with time-varying statistics σ. $\mathcal{G}$ provides regularization to allow the continuous system to model phenomena observed in discrete-time learning systems. In the discrete case, regularization is provided by the sampling procedure (SGD), by the learning rate, and by other explicit mechanisms. We choose time-dependent $\mathcal{G}$ because using a learning-rate schedule has practical importance in training deep models. Specifically, more regularization erases local high-frequency contours in parameter space. As the correct region is approached, regularization is reduced, leading to a better final solution.

Algorithm 1 Continuous Propagation

```
Input: $X$, $\Theta_0$, $\mathcal{L}$, $y$
Output: $\Theta_\infty$
  {Array access is modulo array size}
  $h[D][D]$, hidden activation storage
  $\delta[D][2]$, delta storage
  $\Theta[D]$, parameters
  $\Theta \leftarrow \Theta_0$
  for all $t \in \mathbb{N}$ do
    $h[0][t] \leftarrow x{\sim}X$
    $\delta[D-1][t] \leftarrow \mathcal{L}(h[D][t], y(h[0][t-D]))$
    for all layers $k$ in parallel do
      $h[k][t] \leftarrow \phi(\Theta[k]h[k-1][t-1])$
      $\delta[k][t] \leftarrow \phi'(h[k][t+k])\Theta^T[k]\delta[k+1][t-1])$
      $\Theta[k] \leftarrow \Theta[k] - \alpha(h[k][t+k-D] \times \delta[k][t])$
    end for
  end for
  $\Theta_\infty \leftarrow \Theta$
```

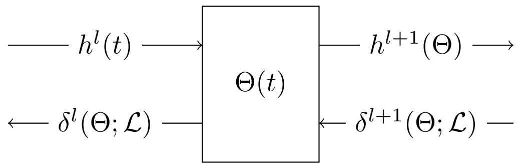

$h^l(t)$
$h^{l+1}(\Theta)$
$\Theta(t)$
$\delta^l(\Theta;\mathcal{L})$
$\delta^{l+1}(\Theta;\mathcal{L})$

Figure 3: Each processor implements layer dynamics. Param values and states (e.g., momentum) reside locally in the node. Activations and deltas stream through, modified by local params.

Algorithm 2 Local Learning Rules for MBGD

$$\nabla = \left(d^{D-l}h^{(l-1)}\right) \times \delta^{(l)}$$

$$G_t = \begin{cases} G_{t-1} + \nabla, & t - 2D + l < N \mod U \\ 0, & \text{otherwise} \end{cases}$$

$$\Theta_t = \begin{cases} \Theta_{t-1} - \frac{\alpha}{n} G_t, & t - 2D + l = N \mod U \\ \Theta_{t-1}, & \text{otherwise} \end{cases}$$

4 Continuous Propagation

Continuous propagation is derived by considering an arbitrary feed-forward neural network (Fig 2). Expressing all nodes as functions of time (Table 1), and applying function composition,[2] gives

$$h^{(k)} = \left(\prod_{i=0}^{k}{}_{\circ}\, \phi^{(i)} \circ \Theta^{(i)}\right) \circ x \,, \tag{6}$$

$$\delta^{(k)} = \left(\prod_{i=k}^{D} \phi'(h^{(i)}) \Theta^{(i)\intercal}\right) \mathcal{L}(h^{(D)}, y) \,, \tag{7}$$

$$\dot{\Theta}^{(k)} = -(h^{(k)} \times \delta^{(k)}) \,. \tag{8}$$

This factorization shows individual network layers as systems with their own local dynamics (Eq. 8; Fig. 3). Internal state Θ evolves according to stimuli h and δ to which it is subjected. The framework is general, and modifications yield analogs of other learning rules. For example, gradient descent with momentum corresponds to[3]

$$\nabla_{\text{mom}}^{(k)}(t) \equiv h^{(k)} \times \delta^{(k)} \,, \tag{9}$$

$$\nu_{\text{mom}}^{(k)}(t) \equiv \int_0^\infty e^{-\lambda\tau} \nabla_{\text{mom}}^{(k)}(t-\tau) d\tau \,, \tag{10}$$

$$\dot{\Theta}_{\text{mom}}^{(k)}(t) \equiv -\lambda \nu_{\text{mom}}^{(k)}(t) \,. \tag{11}$$

The system we are characterizing has two natural dimensions: network depth and time evolution of parameters. These are depicted in the pipeline drawings of Figure 1. Any implementation that seeks acceleration by mapping network layers to computational units separated in space must also accept

[2] $\prod_{i=0}^{k}{}_{\circ}\, f_i \equiv f_k \circ f_{k-1} \circ ... \circ f_0$.

[3] $\nu_{\text{mom}}^{(k)}$ is an IIR filter operating on ∇_{mom}.

latency communicating between these layers. Therefore, we introduce a time delay[4] between layers:

$$h^{(k)} = \left(\prod_{i=0}^{k}{}_{\circ}\, d^1 \circ \phi^{(i)} \circ \Theta^{(i)} \right) \circ x \; ; \tag{12}$$

$$\delta^{(k)} = \left(\prod_{i=k}^{D} \phi'(d^{D-k}(h^{(i)}))\Theta^{(i)\intercal} \right) \mathcal{L}(h^{(D)}, y) \; ; \tag{13}$$

$$\dot{\Theta}^{(k)} = -\left(d^{D-k}(h^{(k)}) \right) \times \delta^{(k)} \; . \tag{14}$$

The continuous-propagation algorithm (Alg. 1; Fig. 1(*c*)) is the synchronous implementation of these time-delay equations.

Notice that an input vector and its hidden representations experience model parameters at different time steps as it travels forward in the network. This difference cannot be detected by the vector itself on the way forward. It is as if we choose a fixed set of parameters from successive time steps and applied learning to the aggregate parameter state that results.

Naturally, we have the choice when the delta values return through the network on the backward pass either to use the *immediate* model parameters as they now stand or else to retrieve the historical version of model parameters[5] *anchored* to when the layer was used for the corresponding forward-pass activity. Deltas computed from the immediate network parameters use updated information corresponding to the current parameter slope. We might expect that they improve gradient descent:

$$\frac{\partial \mathcal{L}}{\partial \Theta^{(k)}} = \frac{\partial h^{(k)}}{\partial \Theta^{(k)}} \times \frac{\partial \mathcal{L}}{\partial h^{(k)}} = h^{(k-1)} \times \delta^{(k)} \; . \tag{15}$$

Our choice of $\delta^{(k)}$ in this vector product can point either in the current direction downhill or in a historical direction. We are better with the current direction so long as $h^{(k-1)}$ is uncorrelated in expectation. In practice we do not see a large difference in these approaches (Sec. 5.2).

Total memory required in this algorithm is $O(D^2)$, the same as in SGD. Traditional techniques like reverse checkpoint (Dauvergne & Hascoët, 2006) can be adapted to this regime (Fig. 1(*d*)). We have the interesting choice when using reverse checkpoint to admit immediate network weights for recomputed activations. When we do this, the recomputed activations differ from those originally computed because of parameter evolution. In addition to reverse checkpoint's normal use in reducing memory footprint, this also reduces the time disparity in parameters used for computing forward-propagating activations and backward-propagating deltas in the aligning wave fronts.

When continuous propagation is implemented as a series of local learning rules for each layer, it is capable of expressing a variety of traditional algorithms. For example, rules for MBGD are shown in Algorithm 2 (Fig. 1(*b*)). Observe that while the computation exactly matches the MBGD algorithm, the expression of synchronization and deferred parameter updates appears somewhat arbitrary.

4.1 Convergence Analysis

Continuous representation of the learning process allows us to relate the time delay used in parameter updates (12-14) to the discrete form of SGD (1). A key observation is that the simulated continuous model time Δt that elapses during a layer computation is proportional to the learning rate α. As the learning rate schedule reduces α, the model becomes asymptotically closer to traditional SGD.

Therefore, we can adapt traditional convergence proofs for online descent to continuous propagation. We extend the approach of Lian et al. (2015) to the case where each layer's parameters get different delays of updates. The asymptotic equivalence allows us to bound layer delays and demonstrate that the expected norms of gradients for each layer converge.

The Appendix presents the formal convergence proof for the continuous-propagation method under the weak assumptions that the gradient of the objective function is Lipschitzian and that the stochastic gradient is unbiased with bounded variance.

[4] $d^k f(t) \equiv f(t - k\Delta t)$, where Δt is layer latency in units of model time.

[5] To use anchored parameters, replace Equation 13 with $\delta^{(k)} = \left(\prod_{i=k}^{D} \phi'(d^{D-k}(h^{(i)}))d^{D-k}(\Theta^{(i)\intercal}) \right) \mathcal{L}(h^{(D)}, y)$.

Table 2: Hyper Parameters tested with Continuous Propagation

Momentum	$\mu_{1/2}$	half-life in epochs	$[0, 0.10]$
Normalization	-	-	None, Normalization PropagationArpit et al. (2016)
Learning Rate	α	-	$[0.001, 0.05]$
Learning Rate Decay	$\alpha_{1/2}$	half-life in epochs	CIFAR: $[12.5, 50]$ SVHN: 1
Parameter Averaging	$\theta_{1/2}$	half-life in epochs	$[0, 1]$
Data Set	-	-	SVHN, CIFAR-10, CIFAR-100
Delta rule	-	-	Anchored, Immediate
Bias	-	-	With bias, No bias allowed
Initialization	-	-	Glorot, Glorot + Orthonormal

5 Experimental Results

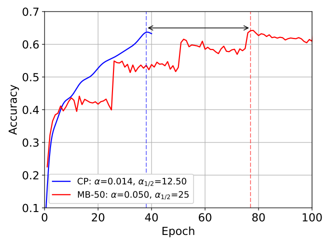
We studied continuous propagation on deep convolutional networks with the network-in-network architecture Lin et al. (2013). We observed successful training in a variety of settings shown in table 2. In our experiments we initiate the learning schedule hyperparameters based on the values derived for MBGD. We decrease the α proportionally to the square root of the batch size $\sqrt{MB}$ and increase the momentum μ to adjust for the $\sqrt{MB}$ factor in the half-life decay.

5.1 Compatible with State of the Art Methods

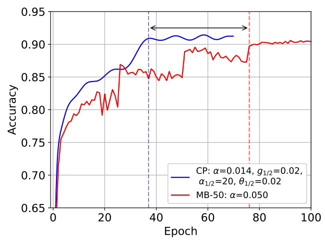
We show CP is compatible with state of the art normalization techniques. We incorporate Normalization Propagation Arpit et al. (2016) which is an adaptation of the popular Batch Normalization Ioffe & Szegedy (2015) technique that works at batch size 1.

We compare validation accuracy on SVHN, CIFAR-10 and CIFAR-100 using continuous propagation with normalization propagation with the results obtained by Arpit et al. (2016) using the same network architecture as the comparison study.

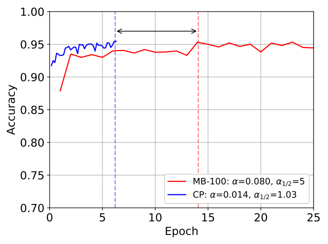
Figure 4 shows validation accuracy in these experiments. In all cases continuous propagation is able to match the validation accuracy in many fewer training epochs.



(a) CIFAR-100



(b) CIFAR-10



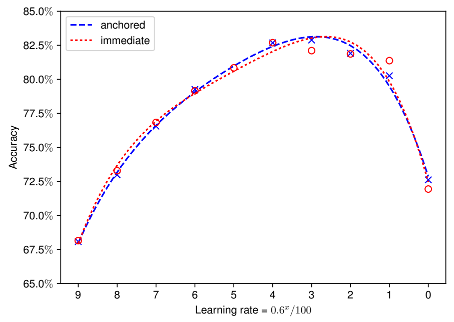
(c) SVHN

Figure 4: Validation-sample accuracy during training

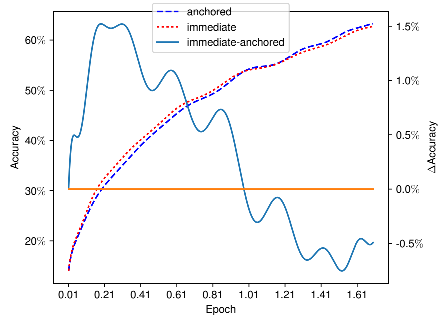
5.2 Anchored-Delta versus Immediate-Delta Rules

We train the convolution network under two conditions, following the discussion from Section 4: the anchored-delta rule, which uses the original parameters; and the immediate-delta rule, which uses the evolved parameters. The goal of this substudy is to compare the learning performance of the two methods. The immediate rule allows us to avoid using extra memory to store the model parameter values for multiple input vectors.

To facilitate comparison, we disable stability-inducing features from the network. For instance, we have no bias addition or normalization of the hidden features. We performed a search over the learning hyper parameter α with momentum fixed at $\mu_{\frac{1}{2}} = 0.0008$.



(a) Immediate- and anchored-rules achieve similar accuracy. Plot shows accuracy at epoch 20 for a sweep of learning rates.



(b) Immediate is better than anchored for first epoch. Plot shows average performance over 100 random initializations.

Figure 5: Comparison of immediate- and anchored- delta rules.

We observe a negligible difference in accuracy between these methods after 20 epochs of training (Fig. 5(a)). We noticed a trend that early in training the immediate rule seemed to out-perform the anchor rule but became slightly worse during the second epoch. Because this difference was small, we decided to run 200 randomized trials to confirm the existence of this effect (Fig. 5(b)).

6 Discussion

The computing model we have posited is generic and capable of implementing traditional MBGD-style algorithms exactly and efficiently. As in other batch schemes, our framework enjoys a property that for large batch sizes computational utilization asymptotically approaches 100%.

We realized that instead of idly waiting for batch-boundary synchronization, we could also immediately start processing the subsequent batch. This results in a strategy that is similar to ASGD. Namely, the parameter inconsistency within the pipeline is limited to no more than one batch boundary. This is also true in ASGD if the worker nodes are balanced on vector throughput. In that case, workers should never be more than one time step removed from their peers.

In considering the optimal batch size to use, we were led down a path of differential equations and rules for the learning dynamics of each layer. Research in feedback alignment shows the importance of feedback dynamics in learning. To implement local dynamics it is natural for weights to reside in physical proximity to their use. This is true in the case of biological neurons (weight encoded in synaptic strength), as it is in our model-parallel algorithm.

When considering what an optimal strategy may look like, we realize that we always have the ability to specify that a layer remain idle at a time step in order to create a global synchronization boundary. Likewise, we also have the ability to allow a weight to remain fixed instead of adopting a new value. Why would either of these strategies be ideal? As we allow data to stream through the neural network, each input from the environment and each measurement of the cost function contains some amount of useful information not yet extracted from the environment. The purpose of any learning equation is to allow the network to respond to this information. In this light both the strategy of "idly waiting" and the strategy of "keeping fixed" are rejections of the utility of this information. See the step-function specification of the MBGD algorithm (Alg. 2). These are indications of suboptimality.

We demonstrated that choosing a lower learning rate dominates using larger batch sizes. Continuous propagation allows statistically efficient learning while maintaining all the cores of a multiprocessor system busy. It permits explicit regularization terms such as L1 penalty on parameters or activations. In gradient descent, explicit regularization works by biasing the gradient in favor of regularization. Regularization's contribution to gradient is available immediately and does not require traversal through the entire network. Therefore in CP parameter update based on regularization penalty is applied before the corresponding loss gradient is applied.

We note that the hidden activity and delta arcs in our pipeline diagrams (Fig. 1) can be considered as individual vectors or batch matrices. Interpreting them as batch matrices allows investigating these techniques directly on contemporary GPU hardware. This is also a demonstration that continuous propagation can be combined with data parallelism in case network depth is exhausted.

References

Devansh Arpit, Yingbo Zhou, Bhargava U. Kota, and Venu Govindaraju. Normalization Propagation: A Parametric Technique for Removing Internal Covariate Shift in Deep Networks. *ArXiv e-prints*, March 2016.

Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

Wojciech Marian Czarnecki, Grzegorz Swirszcz, Max Jaderberg, Simon Osindero, Oriol Vinyals, and Koray Kavukcuoglu. Understanding synthetic gradients and decoupled neural interfaces. *CoRR*, abs/1703.00522, 2017. URL `http://arxiv.org/abs/1703.00522`.

Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *International Conference on Computational Science*, pp. 566–573. Springer, 2006.

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*, pp. 1223–1231, 2012.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 498–515. Springer, 2015.

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pp. 2737–2745, 2015.

Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013. URL `http://arxiv.org/abs/1312.4400`.

Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

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

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. Asynchronous stochastic gradient descent for dnn training. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6660–6663. IEEE, 2013.

Appendix: Convergence Proof

The variable $\mathbf{x}_{k,m,d}$ denotes the random variable for data specifically for the k^{th} training iteration, for the m^{th} sample with the mini-batch used for the d^{th} layer. The variable $\mathbf{\Theta}^d$ with the superscript d denotes parameters from the d^{th} layer only. The variable $\tau_{k,m,d}$ is used to denote that the gradient update for the d^{th}-layer parameter at the k^{th} iteration uses a gradient computed from the parameter $\mathbf{\Theta}^d$ at the $(k-\tau_{k,m,d})^{th}$ iteration using the m^{th} sample.

Assumption 1. *The stochastic gradient $G(\mathbf{\Theta};\mathbf{x})$ is unbiased: $\nabla f(\mathbf{\Theta}) = \mathbb{E}_{\mathbf{x}}[G(\mathbf{\Theta};\mathbf{x})]$.*

Assumption 2. *The variance of the stochastic gradient is bounded: $\mathbb{E}_{\mathbf{x}}[\|G(\mathbf{\Theta}^d;\mathbf{x}) - \nabla f(\mathbf{\Theta}^d)\|^2] \leq \sigma_d^2 \quad \forall \mathbf{\Theta}$.*

Assumption 3. *The gradient function $\nabla f(.)$ is Lipschitzian: $\|\nabla f(\mathbf{\Theta}^d) - \nabla f(\mathbf{\Gamma}^d)\|_2 \leq L_d \|\mathbf{\Theta}^d - \mathbf{\Gamma}^d\|_2 \quad \forall \mathbf{\Theta}, \mathbf{\Gamma}$.*

Assumption 4. *All the random variables in $\{\mathbf{x}_{k,m,d}\}_{k\in\{0,1,\dots,K\},m\in\{1,\dots,M\},d\in\{1,\dots,D\}}$ are independent.*

Assumption 5. *All decay variables $\tau_{k,m,d}$ have an upper bound of some fixed integer T.*

Proposition. *If the assumptions hold and the step-length sequence $\{\gamma_k\}_{k\in\{1,\dots,K\}}$ satisfies*

$$LM\gamma_k + 2L^2M^2T\gamma_k \sum_{t=1}^{T} \gamma_{k+t} \leq 1 \quad \forall k\ , \tag{16}$$

then we have the following convergence for Algorithm 1,

$$\frac{1}{\sum\limits_{k=1}^{K}\gamma_k} \sum_{k=1}^{K} \gamma_k \mathbb{E}[\|\nabla f(\mathbf{\Theta}_k)\|^2] \leq \frac{2(f(\mathbf{\Theta}_1) - f(\mathbf{\Theta}^*))}{M\sum\limits_{k=1}^{K}\gamma_k} + \frac{\sum\limits_{d=1}^{D}\sigma_d^2 \sum\limits_{k=1}^{K}\left(\gamma_k^2 M L_d + 2L_d^2M^2\gamma_k \sum\limits_{j=k-T}^{k-1}\gamma_j^2\right)}{M\sum\limits_{k=1}^{K}\gamma_k}.$$

Proof. From the Lipschitzian assumption on $f(.)$, we have

$$f(\mathbf{\Theta}_{k+1}) - f(\mathbf{\Theta}_k) \leq \langle \nabla f(\mathbf{\Theta}_k), \mathbf{\Theta}_{k+1} - \mathbf{\Theta}_k\rangle + \frac{L}{2}\|\mathbf{\Theta}_{k+1} - \mathbf{\Theta}_k\|^2. \tag{17}$$

Taking expectation with respect to $\mathbf{x}_{k,m,l}$ on both sides (notice that the left-hand side is independent of $\mathbf{x}_{k,m,l}$), we get

$$f(\mathbf{\Theta}_{k+1}) - f(\mathbf{\Theta}_k) \leq \sum_{d=1}^{D}\langle \nabla f(\mathbf{\Theta}_k^d), \sum_{m=1}^{M} G(\mathbf{\Theta}^d_{k-\tau_{k,m,d}}, \mathbf{x}_{k,m,l})\rangle + \frac{\gamma_k^2 L}{2}\|\sum_{m=1}^{M} G(\mathbf{\Theta}^d_{k-\tau_{k,m,d}}, \mathbf{x}_{k,m,l})\|^2. \tag{18}$$

Given that the assumptions hold, and following the proof of theorem 1 of Lian et al. (2015), we get

$$\begin{aligned} &f(\mathbf{\Theta}_{k+1}) - f(\mathbf{\Theta}_1) \\ &\leq \sum_{d=1}^{D} -\frac{M}{2}\sum_{k=1}^{K}\gamma_k\mathbb{E}[\|\nabla f(\mathbf{\Theta}_k^d)\|^2] + \sum_{k=1}^{K}\sigma_d^2\left(\frac{\gamma_k^2 M L_d}{2} + L_d^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right). \end{aligned} \tag{19}$$

Rearranging the terms and dividing by $\sum\limits_{k=1}^{K}\gamma_k$ on both sides, we get

$$\frac{1}{\sum\limits_{k=1}^{K}\gamma_k}\sum_{k=1}^{K}\gamma_k\mathbb{E}[\|\nabla f(\mathbf{\Theta}_k)\|^2] \leq \frac{2(f(\mathbf{\Theta}_1) - f(\mathbf{\Theta}_{k+1})) + \sum\limits_{d=1}^{D}\sigma_d^2\sum\limits_{k=1}^{K}\left(\gamma_k^2ML_d + 2L_d^2M^2\gamma_k\sum\limits_{j=k-T}^{k-1}\gamma_j^2\right)}{M\sum\limits_{k=1}^{K}\gamma_k}. \tag{20}$$

Since $f(\mathbf{\Theta}^*) \leq f(\mathbf{\Theta}_{k+1})$, we achieve our claim. $\square$