NoSync: Particle Swarm Inspired Distributed DNN Training

Mihailo Isakov and Michel A. Kinsy

Adaptive and Secure Computing Systems (ASCS) Laboratory
Department of Electrical and Computer Engineering
Boston University, Boston, MA
{mihailo, mkinsy}@bu.edu

Abstract. Training deep neural networks on big datasets remains a computational challenge. It can take hundreds of hours to perform and requires distributed computing systems to accelerate. Common distributed data-parallel approaches share a single model across multiple workers, train on different batches, aggregate gradients, and redistribute the new model. In this work, we propose *NoSync*, a particle swarm optimization inspired alternative where each worker trains a separate model, and applies pressure forcing models to converge. NoSync explores a greater portion of the parameter space and provides resilience to overfitting. It consistently offers higher accuracy compared to single workers, offers a linear speedup for smaller clusters, and is orthogonal to existing data-parallel approaches.

Keywords: deep learning, artificial neural network, distributed systems, evolutionary algorithm, particle swarm optimization

1 Introduction

Deep neural networks have shown excellent results on a number of tasks such as image recognition [8], machine translation [2], question answering [1], and game playing [16]. In his 2014 keynote on "Large Scale Deep Learning" [4], Jeffrey Dean makes the point that DNN researchers want the results of experiments quickly, and that there is a "patience threshold" they are willing to pay. As state-of-the-art networks require weeks to train with a single GPU on the ImageNet dataset, many researchers are turning to distributed systems for training. This distributed training ranges from running a model on a single machine outfitted with multiple GPUs, to using clusters with thousands of cores, novel architectures, and special interconnects [19].

Two common approaches for distributing neural networks across multiple workers are model parallelism and data parallelism. In model parallelism, network layers are split across multiple workers, and workers communicate neuron activations and gradients. In data parallelism, networks are cloned between workers, workers work on different batches and communicate parameter updates after every batch. Data parallelism is better suited for convolutional neural net parallelization, as this approach requires less network bandwidth [12].

Data parallel distributed DNN training approaches are either synchronous or asynchronous. Synchronous approaches require that all the updates are aggregated before the next training batch can begin. They suffer from low worker utilization due to locking, and are typically only employed when high network bandwidth and homogeneous hardware is available [9]. Asynchronous DNN training aims to fix some of these issues by relaxing the requirement that all workers must finish their updates before the next batch can begin. This significantly raises utilization and reduces bandwidth requirements, but introduces staleness in the system. If left unchecked, the staleness of worker models can range in tens or even hundreds of iterations [5]. This staleness negatively impacts accuracy, prompting a number of researchers to attempt to counter this effect [7][13].

We propose an alternative to the conventional data-parallel approaches. Our intuition stems from the fact that in both synchronous and asynchronous training, a model is cloned across multiple workers, wasting the majority of worker memory. We ask whether using that memory to train individual workers may give us faster convergence, and how would it impact accuracy. We propose NoSync, a Particle Swarm Optimization (PSO) inspired deep neural network training algorithm.

We summarize our contributions here:

- We propose a new type of distributed neural network training, which offers both higher accuracy compared to synchronous and asynchronous data-parallel approaches, as well as lower bandwidth requirements and good scalability.
- We show that *model averaging* can work, as long as the models do not diverge too far, and we provide an insight into the learning happening during NoSync training.
- We verify the results by training common convolutional networks on simulated systems, and show that our training has equivalent utilization and bandwidth requirements as common synchronous approaches.

2 Related work

Processing neural networks typically involves training, inference, or both (known as online training). In case of inference, distributing a neural network is trivial, as each example or batch can be processed independently. In the case of distributed training, there are two methods of parallelizing neural networks present in literature: (1) model parallelization [12][5], where different network layers or neurons are partitioned between machines and all machines work on the same data, and (2) data parallelization [9][15], where the same network model is present on all machines, but trains on different data.

Model parallelization splits a model between multiple workers, requiring the workers to transmit neuron activations for each batch. This approach is efficient in the case of fully-connected layers, where models are large and activations are small, but is very inefficient in the case of convolutional neural networks where the convolution kernels are small, but activations are large [12].

While model parallelization exploits the fact that neural networks are highly parallelizable, data parallelization attempts to parallelize the training algorithm, in this case stochastic gradient descent (SGD). In data parallel distributed DNN training, multiple workers share the same model, but work on different data.

Typically, we take a batch, split it amongst workers, aggregate the calculated gradients, and update all the models [14][5][17]. Data parallel approaches can be further broken down into synchronous and asynchronous. In synchronous data parallel training, a locking mechanism prevents each of the workers from working on stale models, requiring that all machines have identical models at all times. This approach leads to lower utilization, requiring either fast interconnects to achieve good performance [9], or a higher computation/communication ratio [19].

A simple way of increasing worker utilization is allowing the workers to work on batches independently of each other. In asynchronous training, each worker requests the newest model from a parameter server, calculates the gradients on a batch, and sends them back. These gradients are likely not applied to the same model the worker was given, but to a newer one updated by other workers, meaning that the applied update is stale. Asynchronous approaches, while faster than synchronous ones, suffer an accuracy penalty due to this staleness. Several works have attempted to minimize this loss in accuracy [7][13]. In [7], authors inversely weigh the updates by their staleness, meaning that staler updates will have less of an impact on training. While restoring accuracy, this approach does not fully utilize all the workers, as the slower workers might not contribute to training at all due to their lower learning rates. In [13], the authors show that staleness caused by asynchrony can be viewed as just an amount of implicit momentum. By tuning the momentum parameter, they restore the original accuracy while still valuing all updates equally.

Recently, several works have pushed the envelope on the minimum time required to train a network on the ImageNet dataset, ranging from 29 hours on 8 NVidia P100 GPUs [8], down to 1 hour using 256 P100 GPUs [6], and even 15 minutes using 1024 P100's [18]. All of these approaches use synchronous training and try to increase the computation to communication ratio, for example by using batches as large as 32k samples. Similar to our work, but in parallel, the authors in [20] propose training an individual model on every worker and applying elastic averaging between workers as means to prevent divergence. This development serves as further validation o f the proposed approach.

3 NoSync Training

Particle swarm optimization: Particle Swarm Optimization [10] (PSO) is a biology-inspired optimization algorithm imitating the movement of flocks of birds or swarms of insects. It searches for a function extreme by having a population of particles, each of which samples the function at a certain position. Each particle has a position and velocity, and repeatedly moves in the parameter space searching for a better extreme. PSO is gradient-insensitive, easy to parallelize, and is a good global search algorithm.

In PSO, each particle with index j at time t consists of a position x_t^j and velocity v_t^j. A particle keeps track of the best position it has encountered during the search p_t^{ij} and the swarm stores the position p_t^g of the best solution any particle has encountered during the search. PSO introduces two metaparameters: the cognitive parameter c_1 and the social parameter c_2, along with the random values $r_1, r_2 \in [0, 1]$ determined at each iteration.

Each iteration, a particle j updates its position and velocity as:

$$\begin{aligned} x_{t+1}^j &= x_t^j + v_t^j \\ v_{t+1}^j &= v_t^j + c_1 r_1 (p_t^{ij} - x_t^j) + c_2 r_2 (p_t^g - x_t^j) \end{aligned} \tag{1}$$

From equation 1, a particle maintains its speed across iterations, and accelerates towards the best local and global solution. The goal of the cognitive and social parameters is to control the amount of 'pull' applied towards the best individual and swarm solution, respectively. Initially, the swarm should give more freedom to the particles by having a small value of $c2$. Later in the search, PSO increases $c2$, forcing the particles to converge and explore the area around the best solution.

Particle swarm optimization and gradient descent: Classic gradient descent is often prone to overfitting and does not generalize very well. Adding momentum has been shown to help the search escape local minima and find good solutions. For some parameters θ, iteration t, a learning rate α, objective $J(\theta)$, and a batch of input-output pairs x_i and y_i drawn from a dataset, we can write one update as:

$$\begin{aligned} \theta^{t+1} &= \theta^t - v^t \\ v^{t+1} &= \mu v^t + \alpha \nabla_\theta J(\theta; x_i, y_i) \end{aligned} \tag{2}$$

By observing equations 1 and 2, we notice some similarities: (1) both equations maintain a position and speed, and (2) in PSO, each particle is pulled towards the best solution it has encountered ($c_1 r_1 (p_t^{ij} - x_t^j)$), while in gradient descent, a model calculates and applies the gradient, arriving at a better solution ($\alpha \nabla_\theta J(\theta; x_i, y_i)$). The third component $c_2 r_2 (p_t^g - x_t^j)$ of a PSO velocity update has no counterpart in gradient descent - it is used to pull the swarm towards the best solution any particle in the swarm has encountered. Since gradient descent only trains one solution, there is no global solution for it to be pulled towards. From this observation, we introduce a new type of neural network training which trains multiple solutions, and applies a force for them to converge.

Introducing NoSync: In NoSync, for a distributed system of w workers, we train w models, one on each worker. Each worker is trained with classic stochastic gradient descent with momentum. After every batch, we gather the n best performing models, and calculate their mean model c_m, i.e., their 'center of mass'. We then perform pulling - we move each of the w models towards this center of mass c_m. The amount of pull depends on the distance between the model and the center of mass, multiplied by the pull coefficient β. With the metaparameter β set to 0, models will freely diverge. Interpolating two models will typically produce a model whose accuracy is worse than either of the two. This is because the error function on the linear path between them is highly nonconvex. There is no reason to assume that two distant models can gain anything by being interpolated. For that reason, we apply pulling from the very start, forcing the models not to stray too far. If the models are close enough, we can safely assume that the error function between them is convex.

Pulling models: In order to prevent models from diverging, we introduce 'pulling' between workers. In a cluster of w workers, each worker i trains its model W^i on a separate batch, and afterwards sends it over the network to the parameter server. The parameter server computes the average of the models, and pulls all the workers' models towards it by a parameter β as:

$$W^i_{t+1} = (1-\beta)W^i_t + \frac{\beta}{w}\sum_{k=1}^{w} W^k_t \tag{3}$$

Parameter β is chosen so that the models do not diverge to far, but also do not converge to a single point, rendering the parallelization useless. While there is no reason to think that combining different trained models results in a network with comparable accuracy, in section 6 we show that combining or pulling models from the very start results in higher accuracies than that of single machine implementations. There exists an obvious connection between the learning rate α and the pull β: higher learning rates will permit models to diverge further, possibly breaking the above assumption about interpolating loss, and lower learning rates will lead to the models converging and not usefully exploring the parameter space.

In a one-dimensional system, let us assume that there are w particles at time t have positions p^t_i and gradients g^t_i drawn from a normal distribution $g^t_i = \mathcal{N}(0, \sigma^2_g)$. Each iteration, particle i updates its position as:

$$p^{t+1}_i = (1-\beta)(p^t_i + \alpha g^t_i) + \frac{\beta}{w}\sum_{k=1}^{w} p^t_k \tag{4}$$

Assuming that particles are initialized from a normal distribution $\mathcal{N}(0, \sigma^2_w)$, in case when the pull parameter β is $\beta = 0$, one can model the position of a particle as a random walk:

$$\begin{aligned} p^t_i &= p^0_i + \sum_{t=1}^{t} \alpha g^t_i \\ &= \mathcal{N}(0, \sigma^2_w) + \alpha \sum_{t=1}^{t} \mathcal{N}(0, \sigma^2_g) \\ &= \mathcal{N}(0, \sigma^2_w + t\alpha^2\sigma^2_g) \\ &= \mathcal{N}(0, t\alpha^2\sigma^2_g), \quad \sigma^2_w \ll t\alpha^2\sigma^2_g \end{aligned} \tag{5}$$

It follows that two particles m and n at time t will have a distance of:

$$\begin{aligned} |p^t_m - p^t_n| &= |\mathcal{N}(0, t\alpha^2\sigma^2_g) - \mathcal{N}(0, t\alpha\sigma^2_g)| \\ &= |\mathcal{N}(0, 2t\alpha^2\sigma^2_g)| \end{aligned} \tag{6}$$

The absolute value of normal value is a half-normal distribution, with the mean $\mu = \frac{\sigma\sqrt{2}}{\sqrt{\pi}}$. Hence, the average distance can be calculated as:

$$E(|p^t_m - p^t_n|) = \frac{\sqrt{2}\sqrt{2t\alpha^2\sigma^2_g}}{\sqrt{\pi}} = \frac{2\sqrt{t}\alpha\sigma_g}{\sqrt{\pi}} \tag{7}$$

From equation 7 it follows that the average distance between two points grows with the square of time. To prevent different models from diverging, we apply the pull coefficient $\beta \in [0, 1]$. With $\beta \neq 0$, the equation 7 becomes:

$$\begin{aligned} p_i^{t+1} &= (1-\beta)(p_i^t + \alpha g_i^t) + \frac{\beta}{w}\sum_{k=1}^{w} p_k^t \\ &= (1-\beta)(\mathcal{N}(0, \sigma_{p_i^t}^2) + \alpha\mathcal{N}(0, \sigma_g^2)) + \frac{\beta}{w}\sum_{k=0}^{w}\mathcal{N}(0, \sigma_{p_i^t}^2) \\ &= \mathcal{N}(0, (1-\beta)^2(\sigma_{p_i^t}^2 + \alpha^2\sigma_g^2)) + \mathcal{N}(0, \frac{\beta^2}{w^2} w\sigma_{p_i^t}^2) \\ &= \mathcal{N}(0, \alpha^2\sigma_g^2(1-\beta)^2 + \sigma_{p_i^t}^2((1-\beta)^2 + \frac{\beta^2}{w})) \end{aligned} \tag{8}$$

$$\psi = (1-\beta)^2, \quad \omega = (1-\beta)^2 + \frac{\beta^2}{w} \tag{9}$$

$$p_i^{t+1} = \mathcal{N}(0, \psi\alpha^2\sigma_g^2 + \omega\sigma_{p_i^t}^2) \tag{10}$$

In order to determine $\sigma_{p_i^t}^2$, we monitor p_i^t from time-step 0 onwards:

$$\begin{aligned} p_i^0 &= \mathcal{N}(0, \psi\alpha^2\sigma_g^2) \\ p_i^1 &= \mathcal{N}(0, \psi\alpha^2\sigma_g^2 + \omega\psi\alpha^2\sigma_g^2) \\ &\vdots \\ p_i^t &= \mathcal{N}(0, \psi\alpha^2\sigma_g^2\sum_{k=0}^{t}\omega^k) = \mathcal{N}(0, \psi\alpha^2\sigma_g^2\frac{1-\omega^t}{1-\omega}) \end{aligned} \tag{11}$$

The distance of two particles pulled by coefficient β is:

$$\begin{aligned} E(|p_i^t - p_j^t|) &= E(|\mathcal{N}(0, \psi\alpha^2\sigma_g^2\frac{1-\omega^t}{1-\omega}) - \mathcal{N}(0, \psi\alpha^2\sigma_g^2\frac{1-\omega^t}{1-\omega})|) \\ &= \frac{2(1-\beta)\alpha\sigma_g\sqrt{\frac{1-\omega^t}{1-\omega}}}{\sqrt{\pi}} \end{aligned} \tag{12}$$

Therefore, the distance between particles will grow with bigger gradient deviation σ_g, greater learning rate α, and will decrease with increasing pull β. It is also worth noting that this distance will approach infinity when $\beta \to 0$ and $t \to \inf$. Assuming that the number of workers w is large and $t \to \inf$, the distance becomes:

$$E(|p_i^t - p_j^t|) = \frac{2(1-\beta)\alpha\sigma_g}{\sqrt{\pi}\sqrt{2\beta - \beta^2}} \tag{13}$$

Assuming $\beta \ll 1$, the distance changes into:

$$E(|p_i^t - p_j^t|) = \frac{\sqrt{2}\alpha\sigma_g}{\sqrt{\pi}\sqrt{\beta}} \tag{14}$$

With the coefficient $\beta \neq 0$, the mean distance from the center of mass will stabilize, as the random gradients force the particles to diverge irrespective of the the mean distance from the center, while the pull grows linearly with the distance. By increasing β, one can reduce the size of the swarm, and by decreasing β more freedom of movement will be given to the particles. This formulation gives the user the ability to directly control the relative size of the swarm compared with weight updates. One can make sure that the each particle on average is not more than n steps from every other particle.

Dropping models: In classic PSO, each particle is pulled towards a single or multiple best optima encountered. In the above section, NoSync applies pull to the 'center of mass', for which all particles contribute. We explore the possibility of calculating the center of mass from only the n best particles, which might allow the swarm to follow the leaders and faster escape local minima. Given a set N of the n best models, we rewrite equation 3 as:

$$W^i_{t+1} = (1-\beta)W^i_t + \frac{\beta}{n}\sum_{k\in N} W^k_t \tag{15}$$

An additional benefit of this approach is bandwidth reduction - only the particles that are in the top n solutions transmit their model every iteration.

4 Exploring Learning in NoSync

Source of the accuracy increase: NoSync modifies the original synchronous training approach in three ways: (1) It does not synchronize models. Each worker trains a separate model, and periodically sends updates to the parameter server. (2) Instead of synchronizing models, effectively taking their "center of mass", NoSync only pulls them closer. This means that at any point during training, we have w different models, which allows exploring w points in the parameter space, instead of just 1. (3) While synchronous data-parallel training integrates updates from all batches, regardless of how poorly a training model performs, we only integrate the n best performing models, $n \in [1, w]$. This aggregation approach does not mean that low-performing models stop contributing to subsequent training rounds, but rather that their states are disregarded during the current iteration.

These modifications raise the question: does the accuracy increase stem from dropping bad gradients, or from training multiple models in parallel? In order to find the source of the increase in accuracy, we compare several systems:

1. A baseline single worker system;
2. A synchronous model distributed over w workers;
3. 10 models on 10 machines, where after every batch we keep only the last batch's best performing model and redistribute it. This is equivalent to setting $n = 1$ and $\beta = 1$;
4. 10 NoSync trained models, with the n parameter set to 10, i.e., we pull all workers towards the "center of mass";
5. 10 NoSync trained models, with the n parameter set to 3.

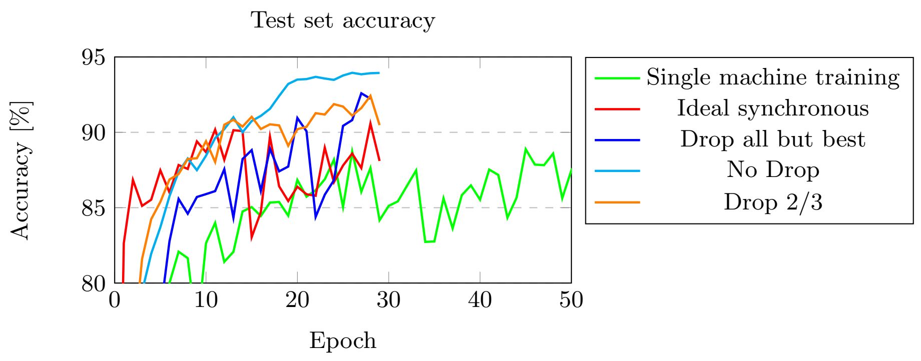

Fig. 1. Training and test accuracy of 5 systems: (green) training with a single worker, (red) classic synchronous training with 10 workers, (blue) dropping all but one, (cyan) pulling all 10 models, (orange) pulling top 3, and dropping 7 models.

In Figure 1, we present the training and test accuracy after 30 epochs of all 5 systems. The three best performing systems are NoSync with $n = 10$, $n = 3$, and $n = 1$, in that order. Evidently, sharing more models between the workers is ideal (no drop), but sharing only the best performing model still allows the system to give a higher accuracy compared to the single-machine and distributed synchronous systems. These results corroborate the fact that accuracy stems from the number of parallel models rather than model dropping. This fact also highlights the trade-off opportunity between accuracy and network communication.

We attribute the accuracy increase to three effects: (1) by training many models, a greater amount of the loss function is explored and there is a higher chance that a good solution will be found. (2) NoSync acts as a regularizer, i.e., though some models may get stuck in local optima or saddle points, other models will get the opportunity to pull out the underperforming ones. (3) Similarly as in PSO, while individual particles may follow local gradients, the whole swarm is less sensitive to nonlinearities of the loss function, and shows more stability.

Saddle points, and broad minima: Another way of understanding NoSync speedup is by observing saddle points during training. In [3], the authors argue that while saddle points will not prevent gradient descent from finding a good local minimum, getting trapped in a saddle point will significantly slow down training. This is due to the fact that, similarly to minima, the gradients in saddle points approach close to zero. Several approaches try to solve this, either by cycling the learning rate as a triangular wave, or by periodically resetting the gradient back to the staring value. NoSync combats this problem by having multiple particles. With a low enough value of β, the particles will have enough freedom and some of them will quickly fall off the saddle point. Particles which fall off will have a larger gradient than that of those trapped in the saddle point, and will pull the trapped ones out.

Next, we test out the quality of NoSync solutions compared to those acquired by conventional training. In [11], authors argue that "broader" local minima are better at generalizing than "narrow" minima. Given a minimum, we would prefer one that is robust to random changes in the parameters, which equates to it being broad. We compare the resilience of two networks to random parameter changes, one network trained with classic stochastic gradient descent, and the other with NoSync. In Figure 2, we vary the amount of noise applied to the parameters and measure the accuracy and loss on the test set. The NoSync models trained with

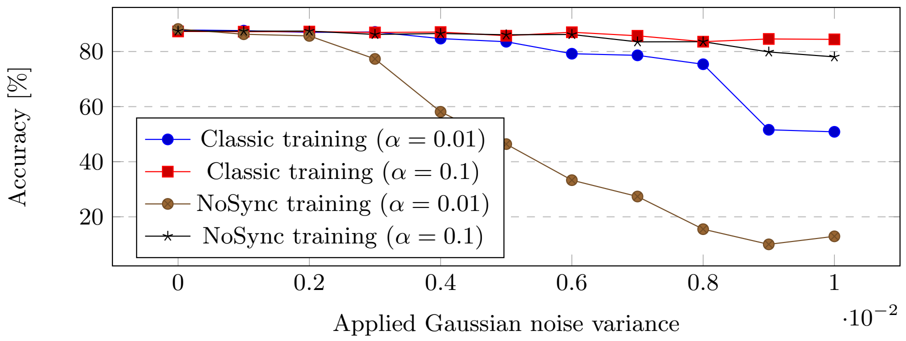

Fig. 2. Robustness to noise of networks trained with classic stochastic gradient descent, and with NoSync. Classic networks are trained for 30 epochs, and NoSync network is trained with 10 workers for 10 epochs. We vary the learning rate (0.1 and 0.01), and set the pull as $\beta = 0.1$.

smaller learning rates (0.01) are more sensitive to perturbations compared with models with larger learning rates. We attribute this effect to multiple particles early on clustering on a single minimum, and fine-tuning it instead of exploring the area. This effect is not present when the learning rate is higher (0.1), as particles will more easily diverge and populate different solutions.

5 System Design

As shown above, the NoSync method converges to higher accuracies compared to conventional approaches. In this section we propose a distributed training architecture and explore techniques for reducing network bandwidth and increasing worker utilization. A typical synchronous data-parallel system uses a number of workers and either parameter servers [5] or reduction trees [9]. Following the work in FireCaffe [9] we design a synchronous training system with $log_2 w - 1$ reduction tree levels. We pick a synchronous over an asynchronous architecture in order to simplify training and not have to consider staleness of the system. In NoSync, each worker computes the forward pass individually and calculates the accuracy on its batch, requiring no network communication. Each worker then sends its accuracy to a parameter server, which sorts the models based on their accuracies. The parameter server requests the models of the n best workers, takes their average, and broadcasts it to all workers on the network. Each worker is tasked with calculating the weighted average of its model and the broadcasted model, and uses this newly calculated model in the next batch.

NoSync and synchronous training: The NoSync method has the same performance as the classic synchronous training when we integrate all the models.

NoSync can further reduce traffic by: (1) decreasing the number of integrated models by dropping the worst performing ones, and (2) introducing stride, i.e., pulling models only every n iterations. Furthermore, if one allows some small staleness, each worker can have full utilization during training.

6 Evaluation

In the case of synchronous data-parallel training, we can prove that if randomness is removed, the system will behave exactly as a single machine implementation. This allows authors to independently monitor speedup and accuracy. In NoSync, however, our speedup stems from a modified search algorithm, and not a purely parallelized implementation of backpropagation. This means that we cannot observe accuracy and speedup in a vacuum, but must measure both together in order to determine the overall benefit of NoSync. For example, a slower NoSync implementation may nonetheless overtake an optimized synchronous one, as it may compute less epochs per second, but have faster convergence per epoch.
NoSync Accuracy: We first focus on whether our search algorithm benefits or hurts overall accuracy. In Figure 3, we compare a baseline single machine system, an ideal w-worker synchronous implementation with a w times larger aggregated batch size, and several different NoSync configurations with different numbers of workers. For testing NoSync, we train a conventional 18-layer ResNet18 network [8]. Due to GPU memory constraints, we did not train deeper networks, as multiple instances of larger networks are unable to fit into the memory of a single NVidia Titan Xp GPU.

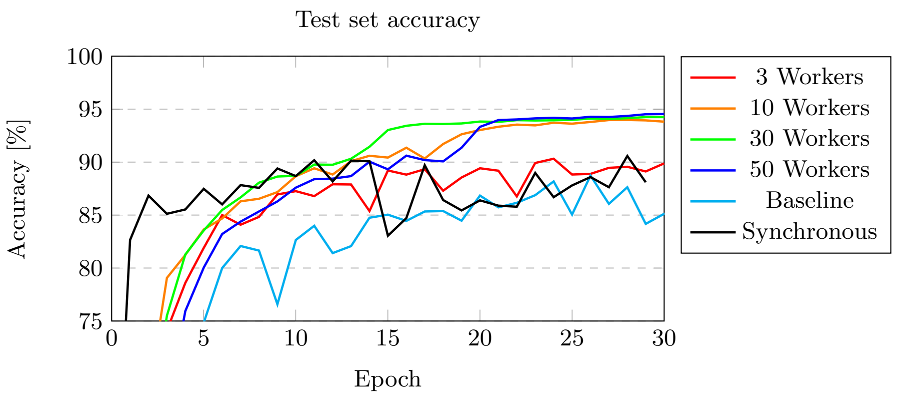

Fig. 3. Per epoch training and test accuracy of 6 systems: (red) NoSync, $w = n = 3$, (orange) NoSync, $w = n = 10$, (green) NoSync, $n = w = 30$, (blue) NoSync, $n = w = 50$, (cyan) Baseline single worker training, (black) Synchronous 10 worker training.

NoSync offers a considerably higher accuracy compared to single machine or synchronous data-parallel approaches. Additionally, we notice that NoSync with 10 workers converges as quickly as the synchronous approach, but additional workers do not speed up convergence.
Metaparameter exploration: We report that the choice of metaparameters greatly affects accuracy. In Figure 4 we compare 3 systems of 10, 30, and 50

machines, and run a grid search on the learning rate α and the pull coefficient β

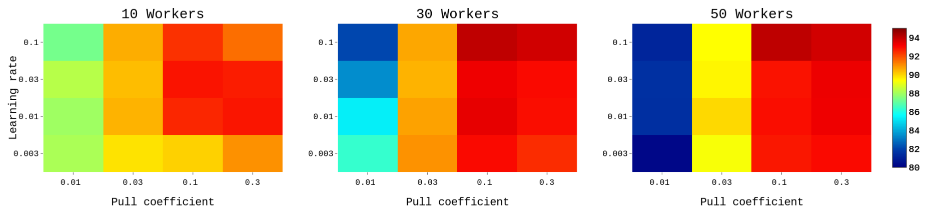

Fig. 4. Test accuracy for 3 different configurations of 10, 30, and 50 machines, training with learning rates $\alpha \in \{0.003, 0.01, 0.03, 0.1\}$ and pulls $\beta \in \{0.01, 0.03, 0.1, 0.3\}$. Each accuracy reported is the best seen on 20 epochs of training. We use a batch size of 512, momentum of 0.9.

Experiments show that the systems with smaller numbers of workers are less sensitive to the metaparameter settings. The amount of 'pull' is normalized for the number of workers, so it is reasonable that a larger cluster will occupy a larger portion of space. The larger the cluster is, the higher the chance that the loss function between each worker and the center of mass will be nonlinear, and pulling will negatively affect their performance. Therefore, we should increase the pulling force with the number of particles.

Overall Speedup: To measure the overall speedup, we measure the number of epochs until convergence for different networks, and the time per epoch for different implementations. In Figure 5, we compare the time to reach 87% accuracy for each of the systems. As we can see, adding more than 3 workers does not significantly speed up convergence.

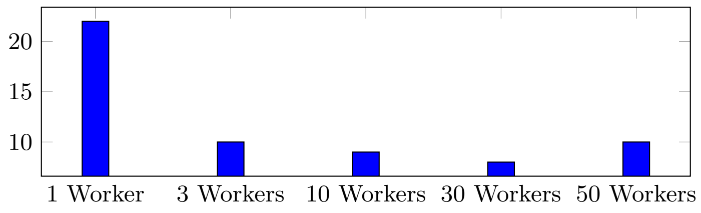

Fig. 5. Epochs until each system achieves 87% test set accuracy.

7 Conclusion

In this work, we presented an alternative distributed DNN training strategy that outperforms synchronous distributed training in terms of both accuracy and performance. We analyzed how this approach converges, and showed experimental results for it. We further proposed a system implementation, and introduced several modifications to it like adding stride and staleness. Future work will focus on providing a strict theoretical backing to the NoSync learning and an architecture exploration exploiting dropping and striding to reduce network contention.

References

1. Andreas, J., Rohrbach, M., Darrell, T., Klein, D.: Learning to compose neural networks for question answering. CoRR abs/1601.01705 (2016)
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. CoRR abs/1409.0473 (2014)
3. Dauphin, Y.N., de Vries, H., Chung, J., Bengio, Y.: Rmsprop and equilibrated adaptive learning rates for non-convex optimization. CoRR abs/1502.04390 (2015)
4. Dean, J.: Large scale deep learning (2014), `https://research.google.com/people/jeff/CIKM-keynote-Nov2014.pdf`
5. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q.V.: Large scale distributed deep networks. Advances in Neural Information Processing Systems pp. 1223–1231 (2012)
6. Goyal, P., Dollár, P., Girshick, R.B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, large minibatch SGD: training imagenet in 1 hour. CoRR abs/1706.02677 (2017)
7. Gupta, S., Zhang, W., Wang, F.: Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. Proceedings - IEEE International Conference on Data Mining, ICDM pp. 171–180 (2017)
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. CoRR abs/1512.03385 (2015)
9. Iandola, F.N., Ashraf, K., Moskewicz, M.W., Keutzer, K.: Firecaffe: near-linear acceleration of deep neural network training on compute clusters. CoRR abs/1511.00175 (2015)
10. Kennedy, J., Eberhart, R.: Particle swarm optimization. Neural Networks, 1995. Proceedings., IEEE International Conference on 4, 1942–1948 vol.4 (1995)
11. Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.: On large-batch training for deep learning: Generalization gap and sharp minima. CoRR abs/1609.04836 (2016)
12. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. CoRR abs/1404.5997 (2014)
13. Mitliagkas, I., Zhang, C., Hadjis, S., Re, C.: Asynchrony begets momentum, with an application to deep learning. 54th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2016 pp. 997–1004 (2017)
14. Niu, F., Recht, B., Re, C., Wright, S.J.: HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent pp. 1–22 (2011)
15. Paine, T., Jin, H., Yang, J., Lin, Z., Huang, T.S.: GPU asynchronous stochastic gradient descent to speed up neural network training. CoRR abs/1312.6186 (2013)
16. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D.: Mastering the game of go without human knowledge. Nature 550, 354 EP – (Oct 2017), article
17. Strom, N.: Scalable distributed DNN training using commodity GPU cloud computing. Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2015-Janua, 1488–1492 (2015)
18. Takuya Akiba, Shuji Suzuki, K.F.: Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes (2017)
19. You, Y., Zhang, Z., Hsieh, C., Demmel, J.: 100-epoch imagenet training with alexnet in 24 minutes. CoRR abs/1709.05011 (2017)
20. Zhang, S., Choromanska, A., LeCun, Y.: Deep learning with elastic averaging SGD. CoRR abs/1412.6651 (2014)