# Improving Performance for Distributed SGD using Ray
# Group 8

Jay Shah
Arizona State University

Rishabh Hatgadkar
Arizona State University

May 5, 2020

**Abstract**

We design here a hybrid architecture in order to overcome the limitations of traditional synchronous and asynchronous parameter server models by defining two new parameters: *gradient staleness* and *pull weights rule*. These parameters help build a model that is not completely synchronous or asynchronous but outperforms baseline models in terms of accuracy and training times.

***Keywords*** Distributed training, synchronous and asynchronous SGD, parameter servers, convolutional neural networks, ablation study

## 1 Introduction

In this project, we implement baseline synchronous and asynchronous Stochastic Gradient Descent (SGD) models using Ray [1] to train MNIST [5], CIFAR-10 [6], and Fashion-MNIST [7] datasets. We then perform an ablation study to design new synchronous and asynchronous SGD models by changing the parameters of the baseline models.

We hypothesize that the new SGD models that have better accuracy and training times than the baseline SGD models are hybrid models that combine aspects of both the synchronous and asynchronous models. Figure 1 illustrates what we mean by a hybrid model.

A hybrid model can be implemented by changing the parameters of a baseline synchronous model to be more asynchronous. A hybrid model can also be implemented by changing the parameters of a baseline asynchronous model to be more synchronous. Our implementation identifies and develops parameters that can increase and decrease the synchronicity and asynchronicity of a model.

### 1.1 Ray

Ray is a Python-based runtime framework that simplifies the process of developing distributed systems. It takes care of all the intra-node communication, data transfer,

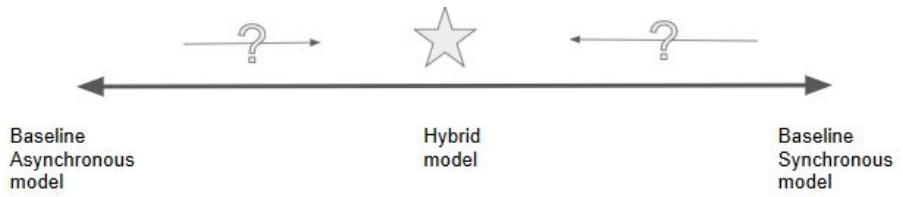


Figure 1: Hybrid model

and resistance to failure that distributed computing requires. We used Ray for its simplicity in implementing our SGD models using the parameter server (PS) architecture.

## 1.2 Parameter Server Architecture

The PS architecture is composed of two entities: *parameter servers* and *workers*. Parameter servers store a machine learning model's parameters. Each worker implements a machine learning model and computes local gradients on a mini-batch. The worker then sends the calculated gradients to the parameter servers. The parameter servers generate new weights using the received gradients. When needed, workers can pull weights from the parameter servers. There are two different PS architectures: *synchronous* and *asynchronous*.

### 1.2.1 Synchronous Parameter Server Architecture

In each training iteration of the synchronous PS architecture, all workers push their gradients to the parameter servers and pull the newly calculated weights from the parameter servers. This is illustrated in Figure 2.

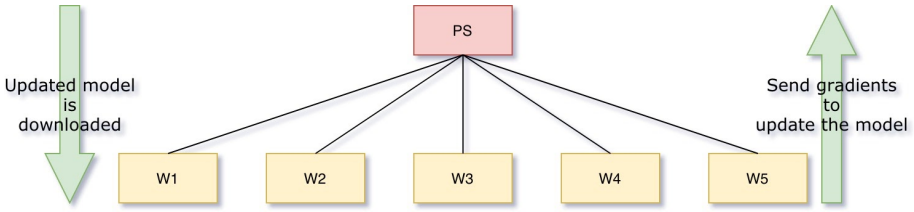


Figure 2: Synchronous PS architecture

The synchronous PS architecture has the advantage of having high accuracies, because all workers are training their models using the most up-to-date weights. The synchronous PS architecture has the disadvantage of slow training times because of having to wait for slow workers before sending the gradients to the parameter servers.

### 1.2.2 Asynchronous Parameter Server Architecture

In each training iteration of the asynchronous PS architecture, only one of the workers pushes its gradients to the parameter servers and pulls weights from the parameter servers. This is illustrated in Figure 3.

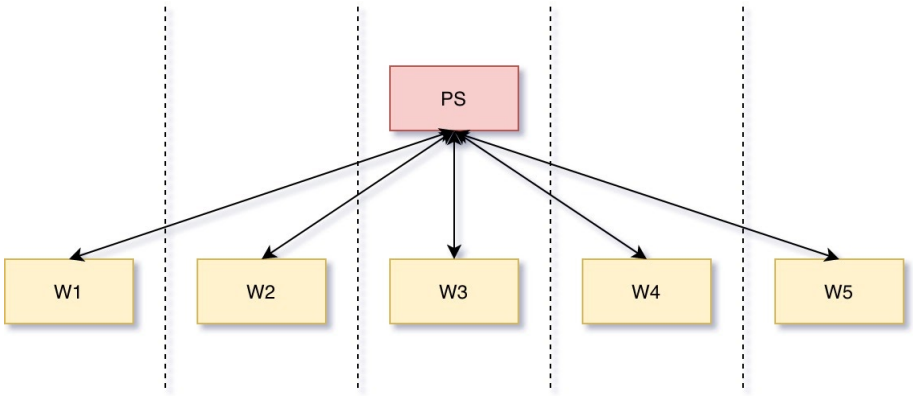


Figure 3: Asynchronous PS architecture

The asynchronous PS architecture has the advantage of having fast training times, because a worker is never blocked by other workers since it pushes gradients to the parameter servers independently of other workers. The asynchronous PS architecture has the disadvantage of having low accuracies, because slow workers would be updating the weights of the parameter servers with stale gradients since they would not be using the most up-to-date weights.

# 2 Related Work

Much research already exists to improve the performance of the PS architecture in training machine learning models. Most of the research implements PS architectures that combine aspects of both synchronous and asynchronous PS architectures.

In [3], Aji and Heafield were able to increase the gradient staleness of a synchronous SGD model by only having the workers pull weights from the parameter servers a certain number of iterations. We use this idea in our implementation to make a synchronous model more asynchronous.

In [4], Ho, et al. were able to reduce the gradient staleness of an asynchronous SGD model by bounding the number of gradient pushes a worker can make based on a staleness threshold. We use this idea in our implementation to make an asynchronous model more synchronous.

# 3 Implementation

In our implementation, we developed a PS architecture using Ray, we developed three SGD models, and we developed new parameters that influence the degrees of synchronicity and asynchronicity of a model.

## 3.1 Parameter Server Architecture using Ray

Ray's *Actors* are used to define the parameter server and workers. A Ray Actor represents an entity as an operating system process. So that means the parameter

server and workers are individual processes that are able to communicate with each other through inter-process communication.

## 3.2 Stochastic Gradient Descent Models

Each worker is running an SGD model. We used PyTorch [2] to implement our SGD models for MNIST, CIFAR-10, and Fashion-MNIST datasets. We have used simpler version of originally benchmarked architectures as the sole aim of this study is to compare the relative performance improvements and not the absolute values.

### 3.2.1 SGD Model for MNIST

The MNIST database contains 60,000 training images and 10,000 testing images and each feature vector (row in the feature matrix) consists of 784 pixels (intensities) unrolled from the original 28×28 pixels images.

- We feed these values into our first fully connected layer and then apply a ReLU activation to the nodes.
- We do this through our two fully connected layers, except for the last one where instead of a ReLU activation we return a log softmax "activation".

### 3.2.2 SGD Model for CIFAR-10

The CIFAR-10 dataset consists of 60,000 32×32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

- The first convolutional layer expects 3 input channels and will convolve 6 filters each of size 3×5×5.
- The first down-sampling layer uses max pooling with a 2x2 kernel and stride set to 2.
- The second convolutional layers expects 6 input channels and will convolve 16 filters each of size 6×5×5.
- The second down-sampling layer uses max pooling with a 2×2 kernel and stride set to 2.
- Then we flatten the tensors and put them into a dense layer, pass through a Multi-Layer Perceptron (MLP) to carry out the task of classification of our 10 categories. The last fully-connected layer uses softmax and is made up of ten nodes, one for each category in CIFAR-10.

### 3.2.3 SGD Model for Fashion-MNIST

FashionMNIST is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. The mapping of all 0-9 integers is to the target class labels.

- The model we built as our baseline has two main aspects: the feature extraction front end comprised of convolutional and pooling layers, and the classifier backend that will make a prediction.
- We have two convolution layers, each with 5×5 kernels. After each convolution layer, we have a max-pooling layer with a stride of 2. This allows us to extract the necessary features from the images.

- Then we flatten the tensors and put them into a dense layer, pass through a Multi-Layer Perceptron (MLP) to carry out the task of classification of our 10 categories.

## 3.3 Parameters

The parameters that can be modified in our models are shown in Table 1.

Table 1: Parameters and their associated variable name

| **Parameter** | **Variable name** |
|---|---|
| Batch size | `batch_size` |
| Number of workers | `num_workers` |
| Learning rate | `lr` |
| Number of workers that send updates per iteration | `num_workers_ps_update` |
| Gradient staleness tolerance | `staleness_tolerance` |
| Interval of iterations to pull weights | `pull_weights_interval_rule` |

The last three parameters in Table 1 influence the degrees of synchronicity and asynchronicity of a model. These parameters are: `num_workers_ps_update`, `staleness_tolerance`, and `pull_weights_interval_rule`.

### 3.3.1 Number of workers that send updates per iteration (`num_workers_ps_update`)

- In the synchronous SGD model, all workers push their gradients to the parameter server per iteration.
- In the asynchronous SGD model, only one worker pushes its gradients to the parameter server per iteration.

To control the number of workers that can push their gradients per iteration, the `num_workers_ps_update` parameter is introduced. The pseudocode for how the `num_workers_ps_update` operates is shown in Figure 4.

```
for each iteration:
    Get gradients from num_workers_ps_update workers
    Push the gradients to parameter server
    Same workers pull latest weights from parameter
        server
```

Figure 4: Pseudocode of `num_workers_ps_update`

From Figure 4, it is seen that only `num_workers_ps_update` workers push their gradients and pull weights in each iteration. Figure 5 illustrates how various values of `num_workers_ps_update` can be used to modify the baseline synchronous and asynchronous models.

In Figure 5, it is expected that `num_workers_ps_update` is set to the number of all workers for baseline synchronous, and `num_workers_ps_update` is set to one for baseline asynchronous.

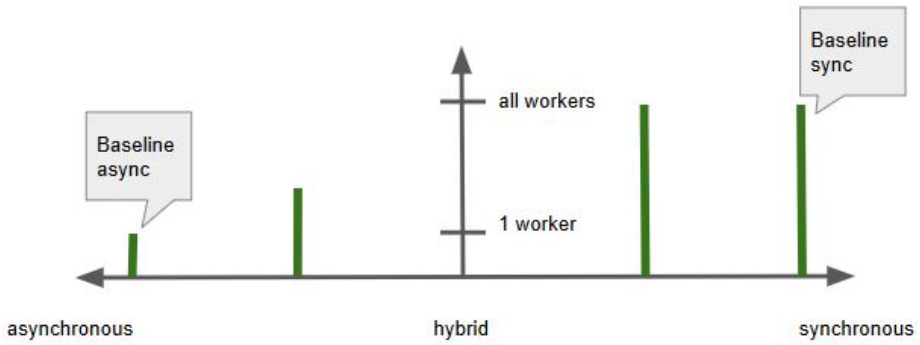


Figure 5: Behavior of modifications to `num_workers_ps_update`

- It is not possible to transform the baseline synchronous model to become more asynchronous by modifying the `num_workers_ps_update` parameter, because in a synchronous PS architecture, all workers must push their gradients per iteration.
- It is possible to transform the baseline asynchronous model to become more synchronous by increasing the value of the `num_workers_ps_update` parameter to a value greater than one and less than the number of all workers.

#### 3.3.2 Gradient staleness tolerance (`staleness_tolerance`)

- In the synchronous SGD model, the gradient staleness tolerance amongst all workers is zero, because all workers pull the weights from the parameter server per iteration.
- In the asynchronous SGD model, the gradient staleness tolerance amongst all workers is infinity, because workers work independently from each other and pull gradients at different times.

To control the staleness of gradients amongst workers, the `staleness_tolerance` parameter is introduced. The pseudocode for how the `staleness_tolerance` operates is shown in Figure 6.

From Figure 6, it is seen that only workers whose number of gradient pushes is within `staleness_tolerance` to the worker with the least number of gradient pushes will be able to push their gradients. Figure 7 illustrates how various values of `staleness_tolerance` can be used to modify the baseline synchronous and asynchronous models.

In Figure 7, it is expected that `staleness_tolerance` is set to zero for baseline synchronous, and `staleness_tolerance` is set to infinity for baseline asynchronous.

- It is not possible to transform the baseline synchronous model to become more asynchronous by modifying the `staleness_tolerance` parameter, because in a synchronous PS architecture, all workers must pull weights from the parameter server per iteration.
- It is possible to transform the baseline asynchronous model to become more synchronous by decreasing the value of the `staleness_tolerance` parameter to a value less than infinity and greater than zero.

```
for each iteration:
    min_w = the worker with minimum number of pushes to
        parameter server
    chosen_workers = []
    for each worker in all_workers:
        staleness = worker.num_pushes - min_w.num_pushes
        if staleness <= staleness_tolerance:
            chosen_workers.append(worker)
    Get gradients from workers in chosen_workers
    Push the gradients to parameter server
    Same workers pull latest weights from parameter
        server
```

Figure 6: Pseudocode of `staleness_tolerance`

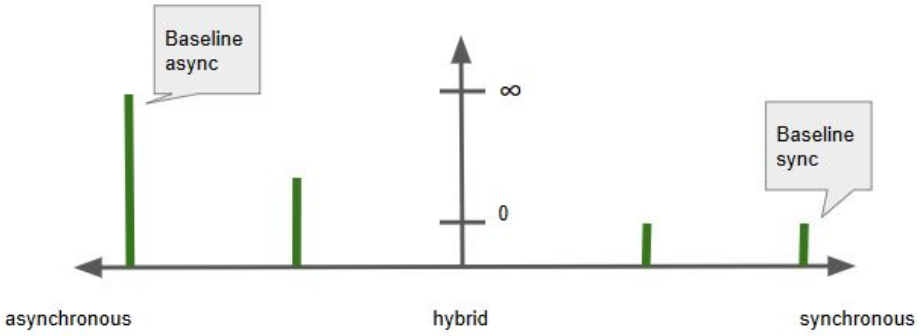


Figure 7: Behavior of modifications to `staleness_tolerance`

### 3.3.3 Interval of iterations to pull weights (`pull_weights_interval_rule`)

In both the baseline synchronous and asynchronous models, weights are pulled from the parameter server in every iteration. To control the interval of iterations to pull weights, the `pull_weights_interval_rule` parameter is introduced. The pseudocode for how the `pull_weights_interval_rule` operates is shown in Figure 8.

```
for each iteration:
    Get gradients from workers
    Push the gradients to parameter server
    if iteration % pull_weights_interval_rule == 0:
        Same workers pull latest weights from parameter
            server
```

Figure 8: Pseudocode of `pull_weights_interval_rule`

From Figure 8, it is seen that increasing the value of the `pull_weights_interval_rule` parameter will lead to fewer occurrences of pulling weights from the parameter server. Figure 9 illustrates how various values of `pull_weights_interval_rule` can be used to modify the baseline synchronous and asynchronous models.

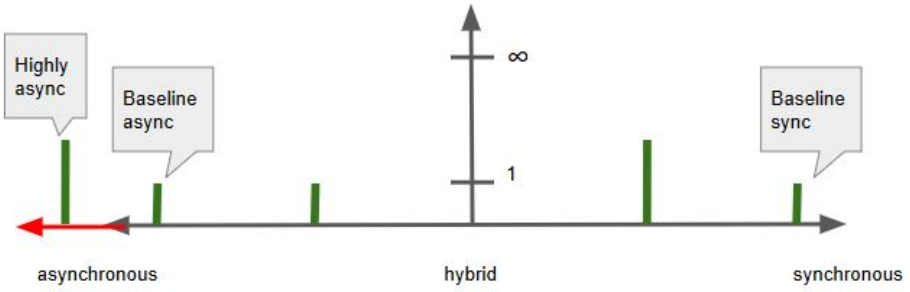


Figure 9: Behavior of modifications to `pull_weights_interval_rule`

In Figure 9, it is expected that `pull_weights_interval_rule` is set to zero for baseline synchronous and asynchronous. It is possible to transform both the baseline synchronous and asynchronous models to become more asynchronous by increasing the value of the `pull_weights_interval_rule` parameter, because it increases the chance that the workers are working with stale weights.

### 3.3.4 Summary

To summarize:

- A synchronous SGD model can be changed to become more asynchronous by increasing the value of the `pull_weights_interval_rule` parameter.
- An asynchronous SGD model can be changed to become more synchronous by increasing the value of the `num_workers_ps_update` parameter or reducing the value of the `staleness_tolerance` parameter.

# 4 Experiments

In our experiments, we first begin by implementing the PS architectures. Next, we identify the parameter values for the baseline synchronous and asynchronous SGD models. Then we measure the test accuracies and training times of the baseline models in units of percentage and seconds, respectively. Finally, we perform the ablation study to develop improved synchronous and asynchronous SGD models. We ran our experiments on two systems: *Machine A* and *Machine B*. The specifications for these systems are shown in Table 2.

Table 2: System specifications

| | **Machine A** | **Machine B** |
|---|---|---|
| **Processor** | Intel(R) Core(TM) i5-8250U | Intel(R) Core(TM) i5 |
| **Number of cores** | 4 | 2 |
| **Number of threads per core** | 2 | 2 |
| **CPU speed (MHz)** | 1800 | 2700 |
| **RAM (GB)** | 12 | 8 |

## 4.1 Setup

We developed our code using Python 3, Ray, and PyTorch. The source code is available publicly in its repository: `https://github.com/rhatgadkar/CSE598Project`. There are three source files in the repository. Each source file implements a PS architecture for one of the three datasets: MNIST, CIFAR-10, and Fashion-MNIST.

Our PS architecture contains one parameter server, and each worker trains for 500 iterations. All the other requirements for the PS architecture are user-specified. The user is prompted to enter the values for these parameters when the program initially runs. These parameters are the ones shown in Table 1.

We ran our experiments for the MNIST and CIFAR-10 datasets on Machine A, and we ran our experiments for the Fashion-MNIST dataset on Machine B.

## 4.2 Baseline Parameter Values

The values of the parameters of Table 1 for both baseline synchronous and asynchronous SGD models are shown in Table 3.

As can be seen in Table 3, the only difference between the baseline synchronous and asynchronous SGD models is in the values of the `num_workers_ps_update` and `staleness_tolerance` parameters.

## 4.3 Baseline SGD Models Performance

The performances of our baseline models for each of the three datasets are shown in Table 4, Table 5, and Table 6.

Table 3: Baseline parameter values

| | **Baseline synchronous** | **Baseline asynchronous** |
|---|---|---|
| `batch_size` | 128 | 128 |
| `num_workers` | 5 | 5 |
| `lr` | 0.03 | 0.03 |
| `num_workers_ps_update` | 5 | 1 |
| `staleness_tolerance` | 0 | $\infty$ |
| `pull_weights_interval_rule` | 1 | 1 |

Table 4: Baseline MNIST performance on Machine A

| | **Accuracy** | **Training time** |
|---|---|---|
| **Baseline synchronous** | 95.38 | 33.24 |
| **Baseline asynchronous** | 94.94 | 32.10 |

Table 5: Baseline CIFAR-10 performance on Machine A

| | **Accuracy** | **Training time** |
|---|---|---|
| **Baseline synchronous** | 48.98 | 68.86 |
| **Baseline asynchronous** | 47.29 | 65.91 |

Table 6: Baseline Fashion-MNIST performance on Machine B

| | **Accuracy** | **Training time** |
|---|---|---|
| **Baseline synchronous** | 30.91 | 202.04 |
| **Baseline asynchronous** | 29.18 | 283.07 |

## 4.4 Ablation Study

In our ablation study, we seek to improve the performance of the baseline synchronous and asynchronous SGD models by attempting to change the values of their parameters.

### 4.4.1 MNIST

Table 7 shows some of the results of the ablation study for synchronous MNIST.

Table 7: Ablation study for synchronous MNIST on Machine A

| Change to baseline parameters | Accuracy | Training time |
|---|---|---|
| num_workers=6, num_workers_ps_update=6 | 95.38 | 38.90 |
| batch_size=64 | 95.09 | 23.96 |
| pull_weights_interval_rule=2 | 93.04 | 35.18 |
| num_workers=6, num_workers_ps_update=6, batch_size=64 | 95.29 | 26.13 |

From the results of the synchronous MNIST ablation study, the following observations are seen:

- Lowering `batch_size` to 64 improves the training time to be even faster than baseline asynchronous without significantly impacting the performance.
- Increasing the value of `pull_weights_interval_rule` to make the model more asynchronous harms the accuracy and training time.
- The best set of parameters to modify to improve the performance of the synchronous MNIST SGD model is the row highlighted in red in Table 7, because the training time is significantly faster than baseline asynchronous and the accuracy is similar to baseline synchronous.

The improvement to the synchronous MNIST SGD model is still considered as a synchronous model.

Table 8 shows some of the results of the ablation study for asynchronous MNIST.

Table 8: Ablation study for asynchronous MNIST on Machine A

| Change to baseline parameters | Accuracy | Training time |
|---|---|---|
| num_workers_ps_update=2 | 95.07 | 33.20 |
| staleness_tolerance=10 | 95.36 | 32.13 |
| num_workers=4 | 95.2 | 28.97 |
| num_workers=6 | 96.02 | 36.08 |
| batch_size=64 | 94.89 | 24.03 |
| staleness_tolerance=10, num_workers=6, batch_size=64 | 95.63 | 25.86 |

From the results of the asynchronous MNIST ablation study, the following observations are seen:

- Increasing the value of `num_workers_ps_update` improves the accuracy of the asynchronous model, but also increases the training time.
- Reducing the value of `staleness_tolerance` to 10 improves the accuracy without impacting the training time.
- If the value of `num_workers` is increased to 6, the accuracy becomes even higher than baseline synchronous.
- Reducing the value of `batch_size` to 64 significantly reduces the training time.
- The best set of parameters to modify to improve the performance of the asynchronous MNIST SGD model is the row highlighted in red in Table 8, because the accuracy is higher than baseline synchronous and the training time is lower than baseline asynchronous.

The improvement to the asynchronous MNIST SGD model is considered as a hybrid model, because the value of the `staleness_tolerance` parameter is reduced.

### 4.4.2 CIFAR-10

Table 9 shows some of the results of the ablation study for synchronous CIFAR-10.

Table 9: Ablation study for synchronous CIFAR-10 on Machine A

| **Change to baseline parameters** | **Accuracy** | **Training time** |
|---|---|---|
| `num_workers`=4, `num_workers_ps_update`=4 | 46.55 | 58.42 |
| `num_workers`=6, `num_workers_ps_update`=6 | 48.50 | 76.08 |
| `lr`=0.05 | 48.79 | 67.43 |
| `batch_size`=64 | 46.42 | 49.60 |
| `pull_weights_interval_rule`=2 | 41.79 | 66.72 |
| `batch_size`=64, `lr`=0.05 | 48.29 | 43.21 |

From the results of the synchronous CIFAR-10 ablation study, the following observations are seen:

- Reducing `num_workers` and `num_workers_ps_update` decreases the accuracy to be lower than baseline asynchronous, but improves the training time to be faster than baseline asynchronous.
- Increasing `num_workers` and `num_workers_ps_update` has a similar accuracy to baseline synchronous, but increases the training time to be slower than baseline synchronous.
- Reducing `batch_size` to 64 significantly reduces the training time to be faster than baseline asynchronous, but also reduces the accuracy to be worse than baseline asynchronous.
- Increasing `lr` to 0.05 seems to keep the accuracy and training time similar to baseline synchronous.
- Increasing `pull_weights_interval_rule` worsens the accuracy to be lower than baseline asynchronous.

- The best set of parameters to modify to improve the performance of the synchronous CIFAR-10 SGD model is the row highlighted in red in Table 9, because the accuracy is similar to baseline synchronous and the training time is faster than baseline asynchronous.

The improvement to the synchronous CIFAR-10 SGD model is still considered as a synchronous model.

Table 10 shows some of the results of the ablation study for asynchronous CIFAR-10.

Table 10: Ablation study for asynchronous CIFAR-10 on Machine A

| **Change to baseline parameters** | **Accuracy** | **Training time** |
|---|---|---|
| `num_workers_ps_update`=2 | 48.56 | 62.43 |
| `staleness_tolerance`=10 | 47.80 | 69.71 |
| `lr`=0.05 | 49.78 | 60.58 |
| `batch_size`=64 | 46.26 | 42.66 |
| `lr`=0.05, `num_workers_ps_update`=2 | 48.60 | 62.59 |
| `lr`=0.05, `num_workers_ps_update`=2, `batch_size`=64 | 47.85 | 41.82 |

From the results of the asynchronous CIFAR-10 ablation study, the following observations are seen:

- Increasing the value of `num_workers_ps_update` to 2 improves the training time to be faster than baseline asynchronous and also improves the accuracy to be similar to baseline synchronous.
- Reducing the `staleness_tolerance` to 10 slightly improves the accuracy, but worsens the training time.
- Reducing the `batch_size` to 64 greatly improves the training time, but worsens the accuracy.
- Increasing the learning rate improves both accuracy and training time.
- The best set of parameters to modify to improve the performance of the asynchronous CIFAR-10 SGD model is the row highlighted in red in Table 10, because the accuracy is better than baseline asynchronous and the training time is much faster than baseline asynchronous.

The improvement to the asynchronous CIFAR-10 SGD model is considered as a hybrid model, because the value of the `num_workers_ps_update` parameter is increased.

### 4.4.3 Fashion-MNIST

Table 11 shows some of the results of the ablation study for synchronous Fashion-MNIST.

From the results of the synchronous Fashion-MNIST ablation study, the following observations are seen:

- Reducing `num_workers` and `num_workers_ps_update` does not improve the training time of baseline synchronous. The accuracy is similar to baseline synchronous though.

Table 11: Ablation study for synchronous Fashion-MNIST on Machine B

| Change to baseline parameters | Accuracy | Training time |
|---|---|---|
| num_workers=4, num_workers_ps_update=4 | 29.6 | 217.59 |
| lr=0.01 | 38.73 | 279.72 |
| batch_size=256 | 27.35 | 517.18 |
| batch_size=64 | 39.8 | 179.81 |
| pull_weights_interval_rule=3 | 33.01 | 118.57 |
| batch_size=64, num_workers=4, num_workers_ps_update=4, lr=0.01, pull_weights_interval_rule=2 | 43.87 | 144.86 |

- Reducing `lr` to 0.01 improves the accuracy, but slows the training time.
- Reducing `batch_size` to 64 greatly improves the accuracy and reduces the training time.
- Increasing `pull_weights_interval_rule` to 3 also improves the accuracy and greatly improves the training time.
- The best set of parameters to modify to improve the performance of the synchronous Fashion-MNIST SGD model is the row highlighted in red in Table 11, because the accuracy is greater than baseline synchronous and the training time is faster than baseline asynchronous and synchronous.

The improvement to the synchronous Fashion-MNIST SGD model is considered as a hybrid model, because the value of the `pull_weights_interval_rule` parameter is increased.

Table 12 shows some of the results of the ablation study for asynchronous Fashion-MNIST.

Table 12: Ablation study for asynchronous Fashion-MNIST on Machine B

| Change to baseline parameters | Accuracy | Training time |
|---|---|---|
| num_workers_ps_update=4 | 33.54 | 269.42 |
| num_workers=4 | 30.24 | 204.85 |
| staleness_tolerance=100 | 21.33 | 253 |
| lr=0.01 | 67.3 | 196.039 |
| batch_size=64 | 27 | 131.06 |
| pull_weights_interval_rule=3 | 37.76 | 107.27 |
| num_workers=4, num_workers_ps_update=3, lr=0.01, pull_weights_interval_rule=3 | 53.7 | 207.46 |

From the results of the asynchronous Fashion-MNIST ablation study, the following observations are seen:

- Decreasing `num_workers_ps_update` makes the accuracy better than baseline synchronous and also improves the training time of baseline asynchronous.

- Decreasing `num_workers` reduces the training time.
- Reducing `staleness_tolerance` does not improve the accuracy, but it improves the training time of baseline asynchronous.
- Reducing `lr` to 0.01 greatly improves both the accuracy and training time. The accuracy and training time are much better than baseline synchronous.
- Reducing `batch_size` to 64 worsens the accuracy, but greatly improves the training time.
- Increasing `pull_weights_interval_rule` to 3 improves both the accuracy and training time.
- The best set of parameters to modify to improve the performance of the asynchronous Fashion-MNIST SGD model is the row highlighted in red in Table 12, because the accuracy is much greater than baseline synchronous and the training time is similar to baseline synchronous and much faster than baseline asynchronous.

The improvement to the asynchronous Fashion-MNIST SGD model is considered as a hybrid model, because the value of the `num_workers_ps_update` parameter is increased.

### 4.5 Summary

Table 13 shows a summary of our findings for the best synchronous and asynchronous improvements to the baseline models.

Table 13: Improvements to baseline models

| **Dataset** | **Synchronous** | **Asynchronous** |
|---|---|---|
| **MNIST** | synchronous | hybrid: `staleness_tolerance` |
| **CIFAR-10** | synchronous | hybrid: `num_workers_ps_update` |
| **Fashion-MNIST** | hybrid: `pull_weights_interval_rule` | hybrid: `num_workers_ps_update` |

- The best improvements to the baseline synchronous models for MNIST and CIFAR-10 were still synchronous models. But for Fashion-MNIST, the best improvement was a hybrid model that came from increasing the value of the `pull_weights_interval_rule` parameter.
- Across all datasets, the best improvements to the baseline asynchronous models were hybrid models that either decreased the value of the `staleness_tolerance` parameter or increased the value of the `num_workers_ps_update` parameter.

## 5 Conclusion

Using a hybrid architecture that is lesser synchronous than the baseline synchronous and lesser asynchronous than the baseline asynchronous indeed achieves better results as per our initial proposition. Such a model definitely contributes towards a more

efficient training architecture in terms of time taken for computation and model updates. It combines the beneficial aspects of both versions of baseline models and tries to optimizes computation. Hence it is wise to use such a hybrid approach for model training whenever feasible for efficient training and lesser resource consumption.

Apart from that, we also found that Ray is a very easy-to-use framework to deploy distributed frameworks for model training, here, a parameter-server architecture. It can handle internal communications for such models very efficiently. Although, as per our results, we saw that in case of complex models, like Fashion-MNIST, where number of parameters can be more, Ray might suffer because of overheads of its implementation. Ray can manage the inter-communication for shallow networks, but for more deep networks Ray might need additional techniques for optimizations, and in those cases it might degrade the performance if not improve.

# References


[1] Philipp Moritz, Robert Nishihara, et al. Ray: A Distributed Framework for Emerging AI Applications. arXiv preprint arXiv:1712.05889, 2017.

[2] Adam Paszke, Sam Gross, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv preprint arXiv:1912.01703, 2019.

[3] Alham Fikri Aji and Kenneth Heafield. Making Asynchronous Stochastic Gradient Descent Work forTransformers. School of Informatics, University of Edinburgh, 2019.

[4] Qirong Ho, James Cipar, et al. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. School of Computer Science, Carnegie Mellon University, 2013.

[5] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/, 2020.

[6] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Department of Computer Science, University of Toronto, 2009.

[7] Kashif Rasul and Han Xiao. Fashion-MNIST. https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/, 2020.