ps-lite Documentation

Release 1.0

ps-lite developers

Nov 04, 2017

Contents

PS-Lite is a lightweight implementation of the parameter server. It provides asynchronous and zero-copy key-value pair communications between machines.

CHAPTER 1

Overview

The parameter server aims for high-performance distributed machine learning applications. In this framework, multiple nodes runs over multiple machines to solve machine learning problems. There are often a single schedule node, and several worker and servers nodes.

- **Worker**. A worker node performs the main computations such as reading the data and computing the gradient. It communicates with the server nodes via `push` and `pull`. For example, it pushes the computed gradient to the servers, or pulls the recent model from them.
- **Server.** A server node maintains and updates the model weights. Each node maintains only a part of the model.
- **Scheduler.** The scheduler node monitors the aliveness of other nodes. It can be also used to send control signals to other nodes and collect their progress.

1.1 Distributed Optimization

Assume we are going to solve the following

$$\min_w \sum_{i=1}^{n} f(x_i, y_i, w)$$

where *(yi, xi)* are example pairs and *w* is the weight.

We consider solve the above problem by minibatch stochastic gradient descent (SGD) with batch size *b*. At time *t*, this algorithm first randomly picks up *b* examples, and then updates the weight *w* by

$$w = w - \eta_t \sum_{i=1}^{b} \nabla f(x_{k_i}, y_{k_i}, w)$$

We give two examples to illusrate the basic idea of how to implement a distributed optimization algorithm in ps-lite.

1.1.1 Asynchronous SGD

In the first example, we extend SGD into asynchronous SGD. We let the servers maintain w, where server k gets the k-th segment of w, denoted by *wk<\sub>*. Once received gradient from a worker, the server k will update the weight it maintained:

```
t = 0;
while (Received(&grad)) {
  w_k -= eta(t) * grad;
  t++;
}
```

where the function `received` returns if received gradient from any worker node, and `eta` returns the learning rate at time t.

While for a worker, each time it dose four things

```
Read(&X, &Y);  // read a minibatch X and Y
Pull(&w);      // pull the recent weight from the servers
ComputeGrad(X, Y, w, &grad);  // compute the gradient
Push(grad);    // push the gradients to the servers
```

where ps-lite will provide function `push` and `pull` which will communicate with servers with the right part of data.

Note that asynchronous SGD is semantically different the single machine version. Since there is no communication between workers, so it is possible that the weight is updated while one worker is calculating the gradients. In other words, each worker may used the **delayed** weights. The following figure shows the communication with 2 server nodes and 3 worker nodes.

1.1.2 Synchronized SGD

Different to the asynchronous version, now we consider a synchronized version, which is semantically identical to the single machine algorithm. We use the scheduler to manage the data synchronization

```
for (t = 0, t < num_iteration; ++t) {
  for (i = 0; i < num_worker; ++i) {
     IssueComputeGrad(i, t);
  }
  for (i = 0; i < num_server; ++i) {
     IssueUpdateWeight(i, t);
  }
  WaitAllFinished();
}
```

where `IssueComputeGrad` and `IssueUpdateWeight` issue commands to worker and servers, while `WaitAllFinished` wait until all issued commands are finished.

When worker received a command, it executes the following function,

```
ExecComputeGrad(i, t) {
   Read(&X, &Y);  // read minibatch with b / num_workers examples
   Pull(&w);      // pull the recent weight from the servers
   ComputeGrad(X, Y, w, &grad);  // compute the gradient
   Push(grad);    // push the gradients to the servers
}
```

which is almost identical to asynchronous SGD but only *b/num_workers* examples are processed each time.

While for a server node, it has an additional aggregation step comparing to asynchronous SGD

```
ExecUpdateWeight(i, t) {
   for (j = 0; j < num_workers; ++j) {
      Receive(&grad);
      aggregated_grad += grad;
   }
   w_i -= eta(t) * aggregated_grad;
}
```

1.1.3 Which one to use?

Comparing to a single machine algorithm, the distributed algorithms have two additional costs, one is the data communication cost, namely sending data over the network; the other one is synchronization cost due to the imperfect load balance and performance variance cross machines. These two costs may dominate the performance for large scale applications with hundreds of machines and terabytes of data.

Assume denotations:

f	convex function
n	number of examples
m	number of workers
b	minibatch size
τ	maximal delay
T_{comm}	data communication overhead of one minibatch
T_{sync}	synchronization overhead

The trade-offs are summarized by

SGD	slowdown of convergence	additional overhead
synchronized	$\sqrt{b}$	$\frac{n}{b}(T_{\text{comm}} + T_{\text{sync}})$
asynchronous	$\sqrt{b\tau}$	$\frac{n}{mb}T_{\text{comm}}$

What we can see are

- the minibatch size trade-offs the convergence and communication cost
- the maximal allowed delay trade-offs the convergence and synchronization cost. In synchronized SGD, we have τ=*0* and therefore it suffers a large synchronization cost. While asynchronous SGD uses an infinite τ to eliminate this cost. In practice, an infinite τ is unlikely happens. But we also place a upper bound of τ to guarantee the convergence with some synchronization costs.

1.2 Further Reads

Distributed optimization algorithm is an active research topic these years. To name some of them

- Dean, NIPS'13, Li, OSDI'14 The parameter server architecture
- Langford, NIPS'09, Agarwal, NIPS'11 theoretical convergence of asynchronous SGD
- Li, NIPS'14 trade-offs with bounded maximal delays τ
- Li, KDD'14 improves the convergence rate with large minibatch size b
- Sra, AISTATS'16 asynchronous SGD adaptive to the actually delay rather than the worst maximal delay
- Li, WSDM'16 practical considerations for asynchronous SGD with the parameter server

- Chen, LearningSys'16 synchronized SGD for deep learning.

CHAPTER 2

Get Started

CHAPTER 3

Tutorials

CHAPTER 4

How To

4.1 Debug PS-Lite

One way to debug is logging all communications. We can do it by specifying the environment variable `PS_VERBOSE`:

- `PS_VERBOSE=1`: logging connection information
- `PS_VERBOSE=2`: logging all data communication information

For example, first run `make test; cd tests` in the root directory. Then

```
export PS_VERBOSE=1; ./local.sh 1 1 ./test_connection
```

Possible outputs are

```
[19:57:18] src/van.cc:72: Node Info: role=schedulerid=1, ip=127.0.0.1, port=8000
[19:57:18] src/van.cc:72: Node Info: role=worker, ip=128.2.211.110, port=58442
[19:57:18] src/van.cc:72: Node Info: role=server, ip=128.2.211.110, port=40112
[19:57:18] src/van.cc:336: assign rank=8 to node role=server, ip=128.2.211.110,
↪port=40112
[19:57:18] src/van.cc:336: assign rank=9 to node role=worker, ip=128.2.211.110,
↪port=58442
[19:57:18] src/van.cc:347: the scheduler is connected to 1 workers and 1 servers
[19:57:18] src/van.cc:354: S[8] is connected to others
[19:57:18] src/van.cc:354: W[9] is connected to others
[19:57:18] src/van.cc:296: H[1] is stopped
[19:57:18] src/van.cc:296: S[8] is stopped
[19:57:18] src/van.cc:296: W[9] is stopped
```

where `H`, `S` and `W` stand for scheduler, server, and worker respectively.

4.2 Use a Particular Network Interface

In default PS-Lite automatically chooses an available network interface. But for machines have multiple interfaces, we can specify the network interface to use by the environment variable `DMLC_INTERFACE`. For example, to use the infinite-band interface `ib0`, we can

```
export DMLC_INTERFACE=ib0; commands_to_run
```

If all PS-Lite nodes run in the same machine, we can set `DMLC_LOCAL` to use memory copy rather than the local network interface, which may improve the performance:

```
export DMLC_LOCAL=1; commands_to_run
```

4.3 Environment Variables to Start PS-Lite

This section is useful if we want to port PS-Lite to other cluster resource managers besides the provided ones such as `ssh`, `mpirun`, `yarn` and `sge`.

To start a PS-Lite node, we need to give proper values to the following environment variables.

- `DMLC_NUM_WORKER` : the number of workers
- `DMLC_NUM_SERVER` : the number of servers
- `DMLC_ROLE` : the role of the current node, can be `worker`, `server`, or `scheduler`
- `DMLC_PS_ROOT_URI` : the ip or hostname of the scheduler node
- `DMLC_PS_ROOT_PORT` : the port that the scheduler node is listening

4.4 Retransmission for Unreliable Network

It's not uncommon that a message disappear when sending from one node to another node. The program hangs when a critical message is not delivered successfully. In that case, we can let PS-Lite send an additional ACK for each message, and resend that message if the ACK is not received within a given time. To enable this feature, we can set the environment variables

- `PS_RESEND` : if or not enable retransmission. Default is 0.
- `PS_RESEND_TIMEOUT` : timeout in millisecond if an ACK message if not received. PS-Lite then will resend that message. Default is 1000.

We can set `PS_DROP_MSG`, the percent of probability to drop a received message, for testing. For example, `PS_DROP_MSG=10` will let a node drop a received message with 10% probability.

CHAPTER 5

APIs

The data communicated are presented as key-value pairs, where the key might be the `uint64_t` (defined by `ps::Key`) feature index and the value might be the according `float` gradient.

1. Basic synchronization functions: \ref ps::KVWorker::Push, \ref ps::KVWorker::Pull, and \ref ps::KVWorker::Wait
2. Dynamic length value push and pull: \ref ps::KVWorker::VPush and \ref ps::KVWorker::VPull
3. Zero-copy versions: \ref ps::KVWorker::ZPush, \ref ps::KVWorker::ZPull, \ref ps::KVWorker::ZVPush and \ref ps::KVWorker::ZVPull

often server *i* handles the keys (feature indices) within the i-th segment of [0, uint64_max]. The server node allows user-defined handles to process the `push` and `pull` requests from the workers.

1. Online key-value store \ref ps::OnlineServer
2. Example user-defined value: \ref ps::IVal
3. Example user-defined handle: \ref ps::IOnlineHandle

also We can also implement

, which is often used to monitor and control the progress of the machine learning application. It also can be used to deal with node failures. See an example in asynchronous SGD.

template <typename *Val*>
struct `ps::`**`KVPairs`**

the structure for a list of key-value pairs

The keys must be unique and sorted in an increasing order. The length of a value can be more than one. If *lens* is empty, then the length of a value is determined by `k=vals.size()/keys.size()`. The *i-th* KV pair is then

```
{keys[i], (vals[i*k], ..., vals[(i+1)*k-1])}
```

If *lens* is given, then `lens[i]` is the length of the *i-th* value. Let

```
n = lens[0] + .. + lens[i-1]
```

then the *i-th* KV pair is presented as

```
{keys[i], (vals[n], ..., vals[lens[i]+n-1])}
```

Public Members

SArray<Key> **keys**

empty constructor

the list of keys

SArray<Val> **vals**

the according values

SArray<int> **lens**

the according value lengths (could be empty)

Index

P