PARL

Release 1.2.3

Apr 02, 2020

Contents

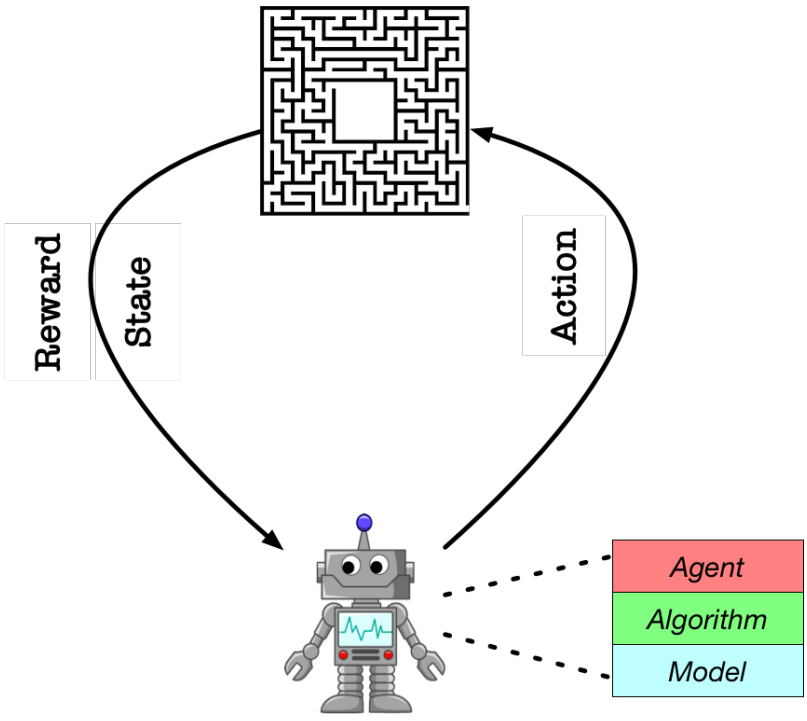
PARL is a flexible, distributed and object-oriented programming reinforcement learning framework.

Object Oriented Programming	Distributed Training

`class MLPModel(parl.Model):`
`  def __init__(self, act_dim):`
`  self.fc1 = layers.fc(size=10)`
`  self.fc2 = layers.fc(size=act_dim)`
 	
`  def forward(self, obs):`
`  out = self.fc1(obs)`
`  out = self.fc2(out)`
`  return out`
 	
`model = MLPModel()`
`target_model = copy.deepcopy(model)`
	`# Absolute multi-thread programming`
	`# witout the GIL limitation`
	
	`@parl.remote_class`
	`class HelloWorld(object):`
	`  def sum(self, a, b):`
	`  return a + b`
	
	`parl.connect('localhost:8003')`
	`obj = HelloWorld()`
	`ans = obj.sum(a, b)`

CHAPTER 1

Abstractions



PARL aims to build an **agent** for training algorithms to perform complex tasks.
The main abstractions introduced by PARL that are used to build an agent recursively are the following:

- **Model** is abstracted to construct the forward network which defines a policy network or critic network given state as input.
- **Algorithm** describes the mechanism to update parameters in the *model* and often contains at least one model.

- **Agent**, a data bridge between the *environment* and the *algorithm*, is responsible for data I/O with the outside environment and describes data preprocessing before feeding data into the training process.

1.1 Installation

1.1.1 Dependencies

- Python 2.7 or 3.5+.
- PaddlePaddle >=1.5.1 (**Optional**, if you only want to use APIs related to parallelization alone)

1.1.2 Install

PARL is distributed on PyPI and can be installed with pip:

```
pip install parl
```

or install from source:

```
pip install --upgrade git+https://github.com/PaddlePaddle/PARL.git
```

1.2 Features

1. Reproducible

We provide algorithms that reproduce stably the results of many influential reinforcement learning algorithms.

2. Large Scale

Ability to support high-performance parallelization of training with thousands of CPUs and multi-GPUs.

3. Reusable

Algorithms provided in the repository can be directly adapted to new tasks by defining a forward network and training mechanism will be built automatically.

4. Extensible

Build new algorithms quickly by inheriting the abstract class in the framework.

1.3 Getting Started

Goal of this tutorial:

- Understand PARL's abstraction at a high level
- Train an agent to solve the Cartpole problem with Policy Gradient algorithm

This tutorial assumes that you have a basic familiarity of policy gradient.

1.3.1 Model

First, let's build a `Model` that predicts an action given the observation. As an objective-oriented programming framework, we build models on the top of `parl.Model` and implement the `forward` function.

Here, we construct a neural network with two fully connected layers.

```
import parl
from parl import layers

class CartpoleModel(parl.Model):
    def __init__(self, act_dim):
        act_dim = act_dim
        hid1_size = act_dim * 10

        self.fc1 = layers.fc(size=hid1_size, act='tanh')
        self.fc2 = layers.fc(size=act_dim, act='softmax')

    def forward(self, obs):
        out = self.fc1(obs)
        out = self.fc2(out)
        return out
```

1.3.2 Algorithm

`Algorithm` will update the parameters of the model passed to it. In general, we define the loss function in `Algorithm`. In this tutorial, we solve the benchmark *Cartpole* using the *Policy Graident* algorithm, which has been implemented in our repository. Thus, we can simply use this algorithm by importting it from `parl.algorithms`.

We have also published various algorithms in PARL, please visit this page for more detail. For those who want to implement a new algorithm, please follow this tutorial.

```
model = CartpoleModel(act_dim=2)
algorithm = parl.algorithms.PolicyGradient(model, lr=1e-3)
```

Note that each `algorithm` should have two functions implemented:

- `learn`

  updates the model's parameters given transition data

- `predict`

  predicts an action given current environmental state.

1.3.3 Agent

Now we pass the algorithm to an agent, which is used to interact with the environment to generate training data. Users should build their agents on the top of `parl.Agent` and implement four functions:

- `build_program`

  define programs of fluid. In general, two programs are built here, one for prediction and the other for training.

- `learn`

  preprocess transition data and feed it into the training program.

- `predict`

  feed current environmental state into the prediction program and return an exectuive action.

- `sample`

  this function is usually used for exploration, fed with current state.

```
class CartpoleAgent(parl.Agent):
    def __init__(self, algorithm, obs_dim, act_dim):
        self.obs_dim = obs_dim
        self.act_dim = act_dim
        super(CartpoleAgent, self).__init__(algorithm)

    def build_program(self):
        self.pred_program = fluid.Program()
        self.train_program = fluid.Program()

        with fluid.program_guard(self.pred_program):
            obs = layers.data(
                name='obs', shape=[self.obs_dim], dtype='float32')
            self.act_prob = self.alg.predict(obs)

        with fluid.program_guard(self.train_program):
            obs = layers.data(
                name='obs', shape=[self.obs_dim], dtype='float32')
            act = layers.data(name='act', shape=[1], dtype='int64')
            reward = layers.data(name='reward', shape=[], dtype='float32')
            self.cost = self.alg.learn(obs, act, reward)

    def sample(self, obs):
        obs = np.expand_dims(obs, axis=0)
        act_prob = self.fluid_executor.run(
            self.pred_program,
            feed={'obs': obs.astype('float32')},
            fetch_list=[self.act_prob])[0]
        act_prob = np.squeeze(act_prob, axis=0)
        act = np.random.choice(range(self.act_dim), p=act_prob)
        return act

    def predict(self, obs):
        obs = np.expand_dims(obs, axis=0)
        act_prob = self.fluid_executor.run(
            self.pred_program,
            feed={'obs': obs.astype('float32')},
            fetch_list=[self.act_prob])[0]
        act_prob = np.squeeze(act_prob, axis=0)
        act = np.argmax(act_prob)
```

(continues on next page)

(continued from previous page)

```
        return act

    def learn(self, obs, act, reward):
        act = np.expand_dims(act, axis=-1)
        feed = {
            'obs': obs.astype('float32'),
            'act': act.astype('int64'),
            'reward': reward.astype('float32')
        }
        cost = self.fluid_executor.run(
            self.train_program, feed=feed, fetch_list=[self.cost])[0]
        return cost
```

1.3.4 Start Training

First, let's build an `agent`. As the code shown below, we usually build a model, an algorithm and finally agent.

```
model = CartpoleModel(act_dim=2)
alg = parl.algorithms.PolicyGradient(model, lr=1e-3)
agent = CartpoleAgent(alg, obs_dim=OBS_DIM, act_dim=2)
```

Then we use this agent to interact with the environment, and run around 1000 episodes for training, after which this agent can solve the problem.

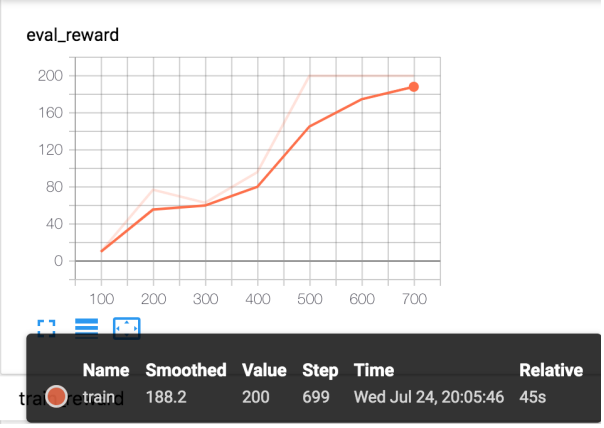
```
def run_episode(env, agent, train_or_test='train'):
    obs_list, action_list, reward_list = [], [], []
    obs = env.reset()
    while True:
        obs_list.append(obs)
        if train_or_test == 'train':
            action = agent.sample(obs)
        else:
            action = agent.predict(obs)
        action_list.append(action)

        obs, reward, done, info = env.step(action)
        reward_list.append(reward)

        if done:
            break
    return obs_list, action_list, reward_list

env = gym.make("CartPole-v0")
for i in range(1000):
      obs_list, action_list, reward_list = run_episode(env, agent)
      if i % 10 == 0:
          logger.info("Episode {}, Reward Sum {}.".format(i, sum(reward_list)))

      batch_obs = np.array(obs_list)
      batch_action = np.array(action_list)
      batch_reward = calc_discount_norm_reward(reward_list, GAMMA)

      agent.learn(batch_obs, batch_action, batch_reward)
      if (i + 1) % 100 == 0:
          _, _, reward_list = run_episode(env, agent, train_or_test='test')
```

(continues on next page)

(continued from previous page)

```
        total_reward = np.sum(reward_list)
        logger.info('Test reward: {}'.format(total_reward))
```

1.3.5 Summary



In this tutorial, we have shown how to build an agent step-by-step to solve the *Cartpole* problem.

The whole training code could be found here. Have a try quickly by running several commands:

```
# Install dependencies
pip install paddlepaddle

pip install gym
git clone https://github.com/PaddlePaddle/PARL.git
cd PARL
pip install .

# Train model
cd examples/QuickStart/
python train.py
```

1.4 Create Customized Algorithms

Goal of this tutorial:

- Learn how to implement your own algorithms.

1.4.1 Overview

To build a new algorithm, you need to inherit class `parl.Algorithm` and implement three basic functions: `sample`, `predict` and `learn`.

1.4.2 Methods

- `__init__`

  As algorithms update weights of the models, this method needs to define some models inherited from `parl.Model`, like `self.model` in this example. You can also set some hyperparameters in this method, like learning_rate, reward_decay and action_dimension, which might be used in the following steps.

- `predict`

  This function defines how to choose actions. For instance, you can use a policy model to predict actions.

- `sample`

  Based on `predict` method, `sample` generates actions with noises. Use this method to do exploration if needed.

- `learn`

  Define loss function in `learn` method, which will be used to update weights of `self.model`.

1.4.3 Example: DQN

This example shows how to implement DQN algorithm based on class `parl.Algorithm` according to the steps mentioned above.

Within class `DQN(Algorithm)`, we define the following methods:

- __init__(self, model, act_dim=None, gamma=None, lr=None)

  We define `self.model` and `self.target_model` of DQN in this method, which are instances of class `parl.Model`. And we also set hyperparameters act_dim, gamma and lr here. We will use these parameters in `learn` method.

```
def __init__(self,
             model,
             act_dim=None,
             gamma=None,
             lr=None):
    """ DQN algorithm

    Args:
        model (parl.Model): model defining forward network of Q function
        hyperparas (dict): (deprecated) dict of hyper parameters.
        act_dim (int): dimension of the action space
        gamma (float): discounted factor for reward computation.
        lr (float): learning rate.
    """
    self.model = model
    self.target_model = copy.deepcopy(model)

    assert isinstance(act_dim, int)
    assert isinstance(gamma, float)
    assert isinstance(lr, float)
    self.act_dim = act_dim
    self.gamma = gamma
    self.lr = lr
```

- predict(self, obs)

We use the forward network defined in `self.model` here, which uses observations to predict action values directly.

```
def predict(self, obs):
        """ use value model self.model to predict the action value
        """
        return self.model.value(obs)
```

- learn(self, obs, action, reward, next_obs, terminal)

  `learn` method calculates the cost of value function according to the predict value and the target value. `Agent` will use the cost to update weights in `self.model`.

```
def learn(self, obs, action, reward, next_obs, terminal):
    """ update value model self.model with DQN algorithm
    """

    pred_value = self.model.value(obs)
    next_pred_value = self.target_model.value(next_obs)
    best_v = layers.reduce_max(next_pred_value, dim=1)
    best_v.stop_gradient = True
    target = reward + (
        1.0 - layers.cast(terminal, dtype='float32')) * self.gamma * best_v

    action_onehot = layers.one_hot(action, self.act_dim)
    action_onehot = layers.cast(action_onehot, dtype='float32')
    pred_action_value = layers.reduce_sum(
        layers.elementwise_mul(action_onehot, pred_value), dim=1)
    cost = layers.square_error_cost(pred_action_value, target)
    cost = layers.reduce_mean(cost)
    optimizer = fluid.optimizer.Adam(self.lr, epsilon=1e-3)
    optimizer.minimize(cost)
    return cost
```

- sync_target(self)

  Use this method to synchronize the weights in `self.target_model` with those in `self.model`. This is the step used in DQN algorithm.

```
def sync_target(self, gpu_id=None):
    """ sync weights of self.model to self.target_model
    """
    self.model.sync_weights_to(self.target_model)
```

1.5 Save and Restore Parameters

Goal of this tutorial:

- Learn how to save and restore parameters.

1.5.1 Example

Sometimes we need to save the parameters into a file and reuse them later on. PARL provides operators to save parameters to a file and restore parameters from a file easily. You only need several lines to implement this.

Here is a demonstration of usage:

```
agent = AtariAgent()
# save the parameters of agent to ./model.ckpt
agent.save('./model.ckpt')
# restore the parameters from ./model.ckpt to agent
agent.restore('./model.ckpt')

# restore the parameters from ./model.ckpt to another_agent
another_agent =  AtariAgent()
another_agent.restore('./model.ckpt')
```

1.6 Overview

1.6.1 Easy-to-use

With a single @parl.remote_class decorator, users can implement parallel training easily, and do not have to care about stuff of multi-processes, network communication.

1.6.2 High performance

@parl.remote_class enable us to achieve real multi-thread computation efficiency without modifying our codes. As shown in figure (a), python's original multi-thread computation performs poorly due to the limitation of the GIL, while PARL empowers us to realize real parallel computation efficiency.

1.6.3 Web UI for computation resources

PARL provides a web monitor to watch the status of any resources connected to the cluster. Users can view the cluster status at a WEB UI. It shows the detailed information for each worker(e.g, memory used) and each task submitted.

1.6.4 Supporting vairous frameworks

PARL for distributed training is compatible with any other frameworks, like tensorflow, pytorch and mxnet. By adding *@parl.remote_class* decorator to their codes, users can easily convert their codes to distributed computation.

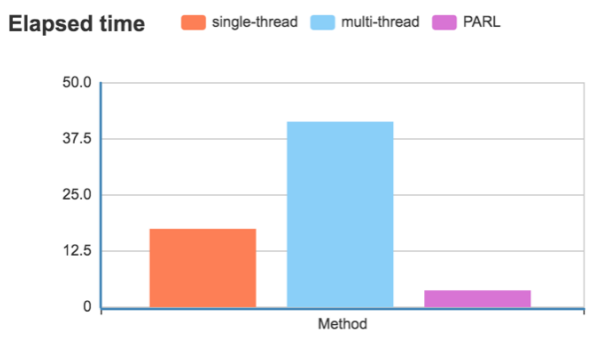
1.6.5 Why PARL

High throughput

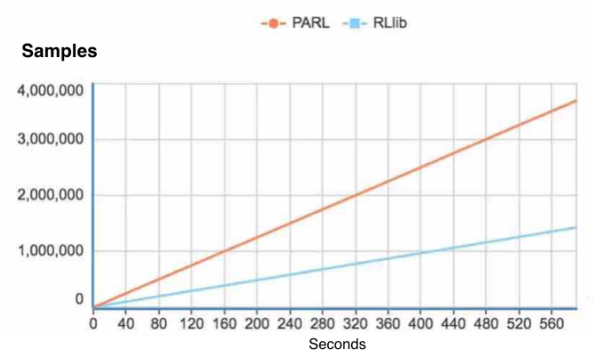
PARL uses a point-to-point connection for network communication in the cluster. Unlike other framework like RLlib which replies on redis for communication, PARL is able to achieve much higher throughput. The results can be found in figure (b). With the same implementation in IMPALA, PARL achieved an increase of 160% on data throughout over Ray(RLlib).

Automatic deployment

Unlike other parallel frameworks which fail to import modules from external file, PARL will automatically package all related files and send them to remote machines.



(a) computation efficiency comparison



(b) sample efficiency comparison

1.7 Cluster Setup

1.7.1 Setup Command

This tutorial demonstrates how to set up a cluster.

To start a PARL cluster, we can execute the following two `xparl` commands:

```
xparl start --port 6006
```

This command starts a master node to manage computation resources and adds the local CPUs to the cluster. We use the port *6006* for demonstration, and it can be any available port.

1.7.2 Adding More Resources

Note: If you have only one machine, you can ignore this part.

If you would like to add more CPUs(computation resources) to the cluster, run the following command on other machines.

```
xparl connect --address localhost:6006
```

It starts a worker node that provides CPUs of the machine for the master. A worker will use all the CPUs by default. If you wish to specify the number of CPUs to be used, run the command with `--cpu_num <cpu_num>` (e.g.——cpu_num 10).

Note that the command `xparl connect` can be run at any time, at any machine to add more CPUs to the cluster.

1.7.3 Example

Here we give an example demonstrating how to use `@parl.remote_class` for parallel computation.

```
import parl

@parl.remote_class
class Actor(object):
```

(continues on next page)

(continued from previous page)

```
    def hello_world(self):
        print("Hello world.")

    def add(self, a, b):
        return a + b

# Connect to the master node.
parl.connect("localhost:6006")

actor = Actor()
actor.hello_world()# no log in the current terminal, as the computation is placed in␣
↪the cluster.
actor.add(1, 2)  # return 3
```

1.7.4 Shutdown the Cluster

run `xparl stop` at the machine that runs as a master node to stop the cluster processes. Worker nodes at different machines will exit automatically after the master node is stopped.

1.7.5 Further Reading

Now we know how to set up a cluster and use this cluster by simply adding `@parl.remote_class`.
In next_tutorial, we will show how this decorator help us implement the **real** multi-thread computation in Python, breaking the limitation of Python Global Interpreter Lock(GIL).

1.8 Recommended Practice

This tutorial shows how to use `@parl.remote_class` to implement parallel computation with **multi-threads**.

Python has poor performance in multi-threading because of the GIL , and we always find that multi-thread programming in Python cannot bring any benefits of the running speed, unlike other program languages such as `C++` and `JAVA`.

Here we reveal the performance of Python threads. At first, let's run the following code:

```
class A(object):
    def run(self):
        ans = 0
        for i in range(100000000):
            ans += i
a = A()
for _ in range(5):
    a.run()
```

This code takes **17.46** seconds to finish counting from 1 to 1e8 for five times.
Now let's implement a thread-based code using the Python library, `threading`, as shown below.

```
import threading

class A(object):
    def run(self):
        ans = 0
        for i in range(100000000):
            ans += i
threads = []
for _ in range(5):
    a = A()
    th = threading.Thread(target=a.run)
    th.start()
    threads.append(th)
for th in threads:
    th.join()
```

It takes **41.35** seconds, much slower than previous code that finish counting serially. As the performance is limited by the GIL, there is only a single CPU running the task, and the CPU has to spend additional time on switching tasks between different threads.

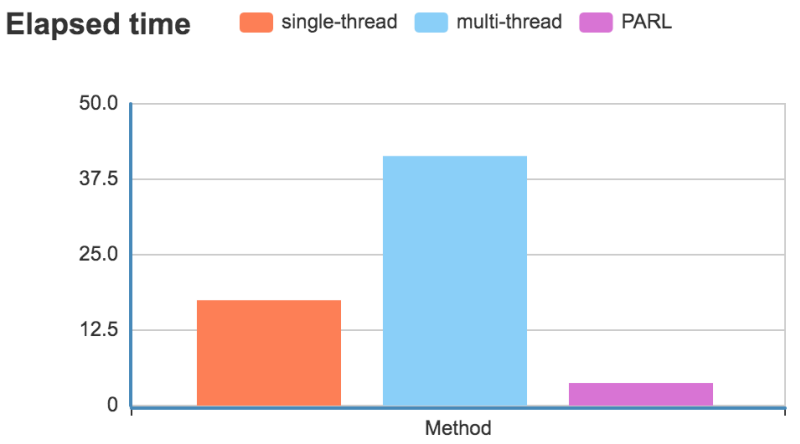
Finally, let's try to use PARL:

```
import threading
import parl

@parl.remote_class
class A(object):
    def run(self):
        ans = 0
        for i in range(100000000):
            ans += i
threads = []
parl.connect("localhost:6006")
for _ in range(5):
    a = A()
    th = threading.Thread(target=a.run)
    th.start()
```

(continues on next page)

(continued from previous page)

```
    threads.append(th)
for th in threads:
    th.join()
```



Only **3.74** seconds are needed !!! As you can see from the code above, it is the `@parl.remote_class` that makes the change happen. By simply adding this decorator, we achieved real multi-thread computation.

1.9 Implemented Algorithms

1.9.1 Policy Gradient

class PolicyGradient(*model*, *hyperparas=None*, *lr=None*)

Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *lr=None*)

Policy Gradient algorithm

Parameters

- **model** (*parl.Model*) – forward network of the policy.
- **hyperparas** (*dict*) – (deprecated) dict of hyper parameters.
- **lr** (*float*) – learning rate of the policy model.

define_learn(*obs*, *action*, *reward*)

update policy model self.model with policy gradient algorithm

Deprecated since version 1.2: This will be removed in 1.3, please use *learn* instead.

define_predict(*obs*)

use policy model self.model to predict the action probability

Deprecated since version 1.2: This will be removed in 1.3, please use *predict* instead.

learn(*obs*, *action*, *reward*)

update policy model self.model with policy gradient algorithm

predict(*obs*)

use policy model self.model to predict the action probability

1.9.2 DQN

class DQN(*model*, *hyperparas=None*, *act_dim=None*, *gamma=None*)
Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *act_dim=None*, *gamma=None*)
DQN algorithm

Parameters

- **model** (*parl.Model*) – model defining forward network of Q function
- **hyperparas** (*dict*) – (deprecated) dict of hyper parameters.
- **act_dim** (*int*) – dimension of the action space
- **gamma** (*float*) – discounted factor for reward computation.
- **lr** (*float*) – learning rate.

define_learn(*obs*, *action*, *reward*, *next_obs*, *terminal*, *learning_rate*)
Deprecated since version 1.2: This will be removed in 1.3, please use *learn* instead.

define_predict(*obs*)
use value model self.model to predict the action value

Deprecated since version 1.2: This will be removed in 1.3, please use *predict* instead.

learn(*obs*, *action*, *reward*, *next_obs*, *terminal*, *learning_rate*)
update value model self.model with DQN algorithm

predict(*obs*)
use value model self.model to predict the action value

sync_target(*gpu_id=None*)
sync weights of self.model to self.target_model

1.9.3 DDPG

class DDPG(*model*, *hyperparas=None*, *gamma=None*, *tau=None*, *actor_lr=None*, *critic_lr=None*)
Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *gamma=None*, *tau=None*, *actor_lr=None*, *critic_lr=None*)
DDPG algorithm

Parameters

- **model** (*parl.Model*) – forward network of actor and critic. The function get_actor_params() of model should be implemented.
- **hyperparas** (*dict*) – (deprecated) dict of hyper parameters.
- **gamma** (*float*) – discounted factor for reward computation.
- **tau** (*float*) – decay coefficient when updating the weights of self.target_model with self.model
- **actor_lr** (*float*) – learning rate of the actor model
- **critic_lr** (*float*) – learning rate of the critic model

define_learn(*obs*, *action*, *reward*, *next_obs*, *terminal*)
update actor and critic model with DDPG algorithm

Deprecated since version 1.2: This will be removed in 1.3, please use *learn* instead.

define_predict(*obs*)
use actor model of self.model to predict the action

Deprecated since version 1.2: This will be removed in 1.3, please use *predict* instead.

learn(*obs*, *action*, *reward*, *next_obs*, *terminal*)
update actor and critic model with DDPG algorithm

predict(*obs*)
use actor model of self.model to predict the action

1.9.4 PPO

class PPO(*model*, *hyperparas=None*, *act_dim=None*, *policy_lr=None*, *value_lr=None*, *epsilon=0.2*)
Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *act_dim=None*, *policy_lr=None*, *value_lr=None*, *epsilon=0.2*)
PPO algorithm

Parameters

- **model** (*parl.Model*) – model defining forward network of policy and value.
- **hyperparas** (*dict*) – (deprecated) dict of hyper parameters.
- **act_dim** (*float*) – dimension of the action space.
- **policy_lr** (*float*) – learning rate of the policy model.
- **value_lr** (*float*) – learning rate of the value model.
- **epsilon** (*float*) – epsilon used in the CLIP loss (default 0.2).

define_policy_learn(*obs*, *actions*, *advantages*, *beta=None*)
Learn policy model with:

Deprecated since version 1.2: This will be removed in 1.3, please use *policy_learn* instead. 1. CLIP loss: Clipped Surrogate Objective 2. KLPEN loss: Adaptive KL Penalty Objective See: https://arxiv.org/pdf/1707.02286.pdf

Parameters

- **obs** – Tensor, (batch_size, obs_dim)
- **actions** – Tensor, (batch_size, act_dim)
- **advantages** – Tensor (batch_size,)
- **beta** – Tensor (1) or None if None, use CLIP Loss; else, use KLPEN loss.

define_predict(*obs*)
Use policy model of self.model to predict means and logvars of actions

Deprecated since version 1.2: This will be removed in 1.3, please use *predict* instead.

define_sample(*obs*)
Use the policy model of self.model to sample actions

Deprecated since version 1.2: This will be removed in 1.3, please use *sample* instead.

define_value_learn(*obs*, *val*)
Learn value model with square error cost

Deprecated since version 1.2: This will be removed in 1.3, please use *value_learn* instead.

define_value_predict(*obs*)
Use value model of self.model to predict value of obs

Deprecated since version 1.2: This will be removed in 1.3, please use *value_predict* instead.

policy_learn(*obs*, *actions*, *advantages*, *beta=None*)

Learn policy model with:

1. CLIP loss: Clipped Surrogate Objective
2. KLPEN loss: Adaptive KL Penalty Objective

See: https://arxiv.org/pdf/1707.02286.pdf

Parameters

- **obs** – Tensor, (batch_size, obs_dim)
- **actions** – Tensor, (batch_size, act_dim)
- **advantages** – Tensor (batch_size,)
- **beta** – Tensor (1) or None if None, use CLIP Loss; else, use KLPEN loss.

predict(*obs*)
Use the policy model of self.model to predict means and logvars of actions

sample(*obs*)
Use the policy model of self.model to sample actions

sync_old_policy(*gpu_id=None*)
Synchronize weights of self.model.policy_model to self.old_policy_model

value_learn(*obs*, *val*)
Learn the value model with square error cost

value_predict(*obs*)
Use value model of self.model to predict value of obs

1.9.5 IMPALA

class IMPALA(*model*, *hyperparas=None*, *sample_batch_steps=None*, *gamma=None*, *vf_loss_coeff=None*, *clip_rho_threshold=None*, *clip_pg_rho_threshold=None*)
Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *sample_batch_steps=None*, *gamma=None*, *vf_loss_coeff=None*, *clip_rho_threshold=None*, *clip_pg_rho_threshold=None*)
IMPALA algorithm

Args: model (parl.Model): forward network of policy and value hyperparas (dict): (deprecated) dict of hyper parameters. sample_batch_steps (int): steps of each environment sampling. gamma (float): discounted factor for reward computation. vf_loss_coeff (float): coefficient of the value function loss. clip_rho_threshold (float): clipping threshold for importance weights (rho). clip_pg_rho_threshold (float): clipping threshold on rho_s in

ho_s delta log pi(alx) (r + gamma v_{s+1} - V(x_s)).

learn(*obs*, *actions*, *behaviour_logits*, *rewards*, *dones*, *learning_rate*, *entropy_coeff*)

Parameters

- **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

- **actions** – An int64 tensor of shape [B].
- **behaviour_logits** – A float32 tensor of shape [B, NUM_ACTIONS].
- **rewards** – A float32 tensor of shape [B].
- **dones** – A float32 tensor of shape [B].
- **learning_rate** – float scalar of learning rate.
- **entropy_coeff** – float scalar of entropy coefficient.

predict(*obs*)

Parameters **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

sample(*obs*)

Parameters **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

1.9.6 A3C

class A3C(*model*, *hyperparas=None*, *vf_loss_coeff=None*)

Bases: *parl.core.fluid.algorithm.Algorithm*

__init__(*model*, *hyperparas=None*, *vf_loss_coeff=None*)

A3C/A2C algorithm

Parameters

- **model** (*parl.Model*) – forward network of policy and value
- **hyperparas** (*dict*) – (deprecated) dict of hyper parameters.
- **vf_loss_coeff** (*float*) – coefficient of the value function loss

learn(*obs*, *actions*, *advantages*, *target_values*, *learning_rate*, *entropy_coeff*)

Parameters

- **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.
- **actions** – An int64 tensor of shape [B].
- **advantages** – A float32 tensor of shape [B].
- **target_values** – A float32 tensor of shape [B].
- **learning_rate** – float scalar of learning rate.
- **entropy_coeff** – float scalar of entropy coefficient.

predict(*obs*)

Parameters **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

sample(*obs*)

Parameters **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

value(*obs*)

Parameters **obs** – An float32 tensor of shape ([B] + observation_space). E.g. [B, C, H, W] in atari.

1.10 parl.Model

class Model(*model_id=None*)

alias: `parl.Model`
alias: `parl.core.fluid.agent.Model`

`Model` is a base class of PARL for the neural network. A `Model` is usually a policy or Q-value function, which predicts an action or an estimate according to the environmental observation.

To track all the layers , users are required to implement neural networks with the layers from `parl.layers` (e.g., parl.layers.fc). These layers has the same APIs as fluid.layers.

`Model` supports duplicating a `Model` instance in a pythonic way:

```
copied_model = copy.deepcopy(model)
```

Example:

```
import parl

class Policy(parl.Model):
    def __init__(self):
        self.fc = parl.layers.fc(size=12, act='softmax')

    def policy(self, obs):
        out = self.fc(obs)
        return out

policy = Policy()
copied_policy = copy.deepcopy(model)
```

Variables **model_id** (*str*) – each model instance has its uniqe model_id.

Public Functions:

- `sync_weights_to`: synchronize parameters of the current model to another model.
- `get_weights`: return a list containing all the parameters of the current model.
- `set_weights`: copy parameters from `set_weights()` to the model.
- `forward`: define the computations of a neural network. **Should** be overridden by all subclasses.
- `parameters`: return a list containting names of parameters of the model.
- `set_model_id`: set `model_id` of current model explicitly.

- `get_model_id`: return the `model_id` of current model.

get_params()
Return a Python list containing parameters of current model.

Deprecated since version 1.2: This will be removed in 1.3, please use *get_weights* instead.

Returns a Python list containing parameters of the current model.

Return type parameters

get_weights()
Returns a Python list containing parameters of current model.

Returns: a Python list containing the parameters of current model.

parameter_names
Get names of all parameters in this `Model`.

Deprecated since version 1.2: This will be removed in 1.3, please use *parameters* instead.

Only parameters created by `parl.layers` are included. The order of parameter names is consistent among different instances of the same *Model*.

Returns list of string containing parameter names of all parameters.

Return type param_names(list)

parameters()
Get names of all parameters in this `Model`.

Only parameters created by `parl.layers` are included. The order of parameter names is consistent among different instances of the same *Model*.

Returns list of string containing parameter names of all parameters

Return type param_names(list)

Example:

```
model = Model()
model.parameters()

# output:
['fc0.w0', 'fc0.bias0']
```

set_params(*params*, *gpu_id=None*)
Set parameters in the model with params.

Deprecated since version 1.2: This will be removed in 1.3, please use *set_weights* instead.

Parameters **params** (*List*) – List of numpy array .

set_weights(*weights*)
Copy parameters from `set_weights()` to the model.

Parameters **weights** (*list*) – a Python list containing the parameters.

sync_params_to(*target_net*, *gpu_id=None*, *decay=0.0*, *share_vars_parallel_executor=None*)
Synchronize parameters in the model to another model (target_net).

Deprecated since version 1.2: This will be removed in 1.3, please use *sync_weights_to* instead.

target_net_weights = decay * target_net_weights + (1 - decay) * source_net_weights

Parameters

- **target_model** (*parl.Model*) – an instance of `Model` that has the same neural network architecture as the current model.
- **decay** (*float*) – the rate of decline in copying parameters. 0 if no parameters decay when synchronizing the parameters.
- **share_vars_parallel_executor** (*fluid.ParallelExecutor*) – Optional. If not None, will use fluid.ParallelExecutor to run program instead of fluid.Executor

sync_weights_to(*target_model*, *decay=0.0*, *share_vars_parallel_executor=None*)

Synchronize parameters of current model to another model.

To speed up the synchronizing process, it will create a program implicitly to finish the process. It also stores a program as the cache to avoid creating program repeatedly.

target_model_weights = decay * target_model_weights + (1 - decay) * current_model_weights

Parameters

- **target_model** (*parl.Model*) – an instance of `Model` that has the same neural network architecture as the current model.
- **decay** (*float*) – the rate of decline in copying parameters. 0 if no parameters decay when synchronizing the parameters.
- **share_vars_parallel_executor** (*fluid.ParallelExecutor*) – Optional. If not None, will use `fluid.ParallelExecutor` to run program instead of `fluid.Executor`.

Example:

```
import copy
# create a model that has the same neural network structures.
target_model = copy.deepcopy(model)

# after initilizing the parameters ...
model.sync_weights_to(target_mdodel)
```

Note: Before calling `sync_weights_to`, parameters of the model must have been initialized.

1.11 parl.Algorithm

class Algorithm(*model=None*, *hyperparas=None*)

alias: `parl.Algorithm`
alias: `parl.core.fluid.algorithm.Algorithm`

`Algorithm` defines the way how to update the parameters of the `Model`. This is where we define loss functions and the optimizer of the neural network. An `Algorithm` has at least a model.

PARL has implemented various algorithms(DQN/DDPG/PPO/A3C/IMPALA) that can be reused quickly, which can be accessed with `parl.algorithms`.

Example:

```
import parl

model = Model()
dqn = parl.algorithms.DQN(model, lr=1e-3)
```

Variables **model** (parl.Model) – a neural network that represents a policy or a Q-value function.

Pulic Functions:

- get_weights: return a Python dictionary containing parameters of the current model.
- set_weights: copy parameters from get_weights() to the model.
- sample: return a noisy action to perform exploration according to the policy.
- predict: return an action given current observation.
- learn: define the loss function and create an optimizer to minized the loss.

Note: Algorithm defines all its computation inside a fluid.Program, such that the returns of functions(*sample*, *predict*, *learn*) are tensors. Agent also has functions like *sample*, *predict*, and *learn*, but they return numpy array for the agent.

__init__(*model=None*, *hyperparas=None*)

Parameters

- **model** (parl.Model) – a neural network that represents a policy or a Q-value function.
- **hyperparas** (*dict*) – a dict storing the hyper-parameters relative to training.

get_params()

Get parameters of self.model.

Deprecated since version 1.2: This will be removed in 1.3, please use *get_weights* instead.

Returns a Python List containing the parameters of self.model.

Return type params(dict)

learn(**args*, ***kwargs*)

Define the loss function and create an optimizer to minize the loss.

predict(**args*, ***kwargs*)

Refine the predicting process, e.g,. use the policy model to predict actions.

sample(**args*, ***kwargs*)

Define the sampling process. This function returns an action with noise to perform exploration.

set_params(*params*)

Set parameters from get_params to the model.

Deprecated since version 1.2: This will be removed in 1.3, please use *set_weights* instead.

Parameters **params** (*dict*) – a Python List containing the parameters of self.model.

1.12 parl.Agent

class Agent(*algorithm*, *gpu_id=None*)

alias: `parl.Agent`
alias: `parl.core.fluid.agent.Agent`

Agent is one of the three basic classes of PARL.

It is responsible for interacting with the environment and collecting data for training the policy.
To implement a customized `Agent`, users can:

```
import parl

class MyAgent(parl.Agent):
    def __init__(self, algorithm, act_dim):
        super(MyAgent, self).__init__(algorithm)
        self.act_dim = act_dim
```

This class will initialize the neural network parameters automatically, and provides an executor for users to run the programs (self.fluid_executor).

Variables

- **gpu_id** (*int*) – deprecated. specify which GPU to be used. -1 if to use the CPU.
- **fluid_executor** (*fluid.Executor*) – executor for running programs of the agent.
- **alg** (*parl.algorithm*) – algorithm of this agent.

Public Functions:

- `build_program` (**abstract function**): build various programs for the agent to interact with outer environment.
- `get_weights`: return a Python dictionary containing all the parameters of self.alg.
- `set_weights`: copy parameters from `set_weights()` to this agent.
- `sample`: return a noisy action to perform exploration according to the policy.
- `predict`: return an action given current observation.
- `learn`: update the parameters of self.alg using the *learn_program* defined in *build_program()*.
- `save`: save parameters of the `agent` to a given path.
- `restore`: restore previous saved parameters from a given path.

__init__(*algorithm*, *gpu_id=None*)
Build programs by calling the method `self.build_program()` and run initialization function of `fluid.default_startup_program()`.

Parameters

- **algorithm** (*parl.Algorithm*) – an instance of *parl.Algorithm*. This algorithm is then passed to *self.alg*.

- **gpu_id** (*int*) – deprecated. specify which GPU to be used. -1 if to use the CPU.

build_program()
Build various programs here with the learn, predict, sample functions of the algorithm.

Note:

Users **must** implement this function in an `Agent`.
This function will be called automatically in the initialization function.

To build a program, you must do the following:

a. Create a fluid program with `fluid.program_guard()`;

b. Define data layers for feeding the data;

c. Build various programs(e.g., learn_program, predict_program) with data layers defined in step b.

Example:

```
self.pred_program = fluid.Program()

with fluid.program_guard(self.pred_program):
    obs = layers.data(
        name='obs', shape=[self.obs_dim], dtype='float32')
    self.act_prob = self.alg.predict(obs)
```

get_params()
Returns a Python dictionary containing the whole parameters of self.alg.

Deprecated since version 1.2: This will be removed in 1.3, please use *get_weights* instead.

Returns a Python List containing the parameters of self.alg.

learn(**args*, ***kwargs*)
The training interface for `Agent`. This function feeds the training data into the learn_program defined in `build_program()`.

predict(**args*, ***kwargs*)
Predict an action when given the observation of the environment.

This function feeds the observation into the prediction program defined in `build_program()`. It is often used in the evaluation stage.

restore(*save_path*, *program=None*)
Restore previously saved parameters. This method requires a program that describes the network structure. The save_path argument is typically a value previously passed to `save_params()`.

Parameters

- **save_path** (*str*) – path where parameters were previously saved.
- **program** (*fluid.Program*) – program that describes the neural network structure. If None, will use self.learn_program.

Raises `ValueError` – if program is None and self.learn_program does not exist.

Example:

```
agent = AtariAgent()
agent.save('./model.ckpt')
agent.restore('./model.ckpt')
```

sample(**args*, ***kwargs*)

Return an action with noise when given the observation of the environment.

In general, this function is used in train process as noise is added to the action to preform exploration.

save(*save_path*, *program=None*)

Save parameters.

Parameters

- **save_path** (*str*) – where to save the parameters.
- **program** (*fluid.Program*) – program that describes the neural network structure. If None, will use self.learn_program.

Raises ValueError – if program is None and self.learn_program does not exist.

Example:

```
agent = AtariAgent()
agent.save('./model.ckpt')
```

set_params(*params*)

Copy parameters from get_params() into this agent.

Deprecated since version 1.2: This will be removed in 1.3, please use *set_weights* instead.

Parameters **params** (*dict*) – a Python List containing the parameters of self.alg.

Python Module Index

p

Index

Symbols

A

B

D

G

I

L

M

P

R

S

V