Machin

Release 0.3.4

Iffi

Apr 19, 2021

CONTENTS

Welcome to the main documentation of **Machin** library.

CHAPTER

ONE

ABOUT

1.1 About

Machin is a reinforcement library purely based on pytorch, it is designed with three things in mind:

1. **Easy to understand.**
2. **Easy to extend.**
3. **Easy to reuse.**

The first goal is achieved through clear structure design, robust document, and concise description of use cases. The second goal is achieved through adding an extra layer upon basic apis provided in the distributed module of pytorch, this layer offers additional fault tolerance mechanism and eliminates hassles occurring in distributed programming. The last goal is the result of modular designs, careful api arrangements, and experiences gathered from other similar projects.

Compared to other versatile and powerful reinforcement learning frameworks, Machin tries to offer a pleasant programming experience, smoothing out as many obstacles involved in reinforcement learning and distributed programming as possible. Some essential functions such as automated tuning and neural architecture search are not offered in this package, we strongly recommend you take a look at these amazing projects and take a piggyback ride:

- ray tune
- tpot
- nni

CHAPTER

TWO

INSTALLATION

Machin is hosted on PyPI, currently it requires:

1. python >= 3.5
2. torch >= 1.5.0

If you are using PIP to manage your python packages, you may directly type:

```
pip install machin
```

If you are using conda to manage your python packages, you are suggested to create a virtual environment first, to prevent PIP changes your packages without letting conda know:

```
conda create -n some_env pip
conda activate some_env
pip install machin
```

CHAPTER

THREE

TUTORIALS AND EXAMPLES

3.1 Tutorials

3.1.1 Your first program

Author: Muhan Li

Full code: Github

This tutorial will guide you to create your first Deep Q Learning (DQN) agent on the CartPole-v0 task. from the OpenAI Gym.

Preface

Some sections of this tutorial are copied and adapted from the Reinforcement Learning (DQN) Tutorial , written by **Author**: Adam Paszke, credits attributed to him.

Task

The agent has to decide between two actions - moving the cart left or right - so that the pole attached to it stays upright. You can find an official leaderboard with various algorithms and visualizations at the Gym website.

Fig. 3.1: cartpole

As the agent observes the current state of the environment and chooses an action, the environment *transitions* to a new state, and also returns a reward that indicates the consequences of the action. In this task, rewards are +1 for every incremental timestep and the environment terminates if the pole falls over too far or the cart moves more then 2.4 units away from center. This means better performing scenarios will run for longer duration, accumulating larger return.

The CartPole task is designed so that the inputs to the agent are 4 real values representing the environment state (position, velocity, etc.). However, neural networks can solve the task purely by looking at the scene, so we'll use a patch of the screen centered on the cart as an input. Because of this, our results aren't directly comparable to the ones from the official leaderboard - our task is much harder. Unfortunately this does slow down the training, because we have to render all the frames.

Strictly speaking, we will present the state as the difference between the current screen patch and the previous one. This will allow the agent to take the velocity of the pole into account from one image.

A theoretical explanation of DQN

Our environment is deterministic, so all equations presented here are also formulated deterministically for the sake of simplicity. In the reinforcement learning literature, they would also contain expectations over stochastic transitions in the environment.

Our aim will be to train a policy that tries to maximize the discounted, cumulative reward $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$, where R_{t_0} is also known as the *return*. The discount, γ, should be a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future that it can be fairly confident about.

The main idea behind Q-learning is that if we had a function $Q^*: State \times Action \rightarrow \mathbb{R}$, that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi^*(s) = \underset{a}{\mathrm{argmax}} \ Q^*(s, a)$$

However, we don't know everything about the world, so we don't have access to Q^*. But, since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*.

For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation:

$$Q^{\pi}(s, a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

The difference between the two sides of the equality is known as the temporal difference error, δ:

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a))$$

To minimise this error, we will use the common MSE loss.

DQN framework

The DQN framework is defined in *machin.frame.algorithms.dqn*, you may import it with the following statements:

```python
from machin.frame.algorithms import DQN
# Or with the following statement
from machin.frame.algorithms.dqn import DQN
```

DQN framework is one of the three major types of model-free reinforcement methods supported by Machin. To initialize it, you must at least provide a Q network, a target Q network, an optimizer used to optimize the first Q network, and a criterion used to determine distance between the estimated Q value and the target Q value we would like to reach:

```python
def __init__(self,
             qnet: Union[NeuralNetworkModule, nn.Module],
             qnet_target: Union[NeuralNetworkModule, nn.Module],
             optimizer: Callable,
             criterion: Callable,
             *_,
             lr_scheduler: Callable = None,
             lr_scheduler_args: Tuple[Tuple] = None,
             lr_scheduler_kwargs: Tuple[Dict] = None,
             batch_size: int = 100,
             update_rate: float = 0.005,
             learning_rate: float = 0.001,
```

(continues on next page)

(continued from previous page)

```
                discount: float = 0.99,
                gradient_max: float = np.inf,
                replay_size: int = 500000,
                replay_device: Union[str, t.device] = "cpu",
                replay_buffer: Buffer = None,
                mode: str = "double",
                visualize: bool = False,
                visualize_dir: str = "",
                **__):...
```

Your Q network

DQN framework supports multiple `mode` s, the `mode` parameter could be one of "vanilla", "fixed_target" or "double", for more detailed explanations on these `mode` s, please refer to *DQN*.

Depending on the Q framework `mode`, your network configurations might be a little different, by generally speaking, your Q network should accept a state, and then output estimated Q values for each action. A simple example would be:

```
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, some_state):
        a = t.relu(self.fc1(some_state))
        a = t.relu(self.fc2(a))
        return self.fc3(a)
```

Please take care of the function signature of `forward`, because the name of its arguments will be examined when the DQN framework tries to perform a forward operation on your Q network, during training or inference.

Now, please remember the name of the state argument: **"some_state"**.

Optimizer and criterion

In order to optimize your model, you must specify an optimizer and a criterion. Usually the optimizer is `torch.optim.Adam`. We are going to use the good old MSE loss `nn.MSELoss` here.

We have all the ingredients required to start the ignition sequence of the DQN framework, lets mix these parts together:

```
q_net = QNet(observe_dim, action_num)
q_net_t = QNet(c.observe_dim, c.action_num)
dqn = DQN(q_net, q_net_t,
          t.optim.Adam,
          nn.MSELoss(reduction='sum'))
```

The framework might will print two warnings for not setting the input/output device of Q networks, but lets ignore that for now. You may quite Machin down either by:

```
# to mark the input/output device Manually
# will not work if you move your model to other devices
# after wrapping

q_net = static_module_wrapper(q_net, "cpu", "cpu")
q_net_t = static_module_wrapper(q_net_t, "cpu", "cpu")
```

Or by:

```
# to mark the input/output device Automatically
# will not work if you model locates on multiple devices

q_net = dynamic_module_wrapper(q_net)
q_net_t = dynamic_module_wrapper(q_net_t)
```

static_module_wrapper and *dynamic_module_wrapper* can be imported from *machin.model.nets*

Store a step

The DQN framework has encapsulated a replay buffer inside, in order to interact with the internal replay buffer, you may use either one of the following APIs, according to your needs:

```
dqn.store_transition(transition: Union[Transition, Dict])
dqn.store_episode(episode: List[Union[Transition, Dict]])
```

`store_transition` stores a single transition step in your MDP process, while `store_episode` stores all transitions inside a MDP process.

When you are using other frameworks, these two APIs may both be supported, or only one of them is supported, depending on the internal implementations of frameworks, and requirements of algorithms.

Now lets take DQN as an example, each `Transition` object describes a single step of a MDP process, and constitutes of 5 attributes:

1. state: State observed by your agent when transition begins.
2. action: Action taken by your agent in this transition step.
3. next_state: Next state observed by your agent, when action is taken.
4. reward: Incremental reward given to your agent, due to the taken action.
5. terminal: Whether the next state is the terminal state of current MDP.

Suppose the observation dimension of your agent is 5, contiguous, within range $(-\infty, +\infty)$, and total number of available discreet actions is 3, then an example transition step would be:

```
# some states observed by your agent
old_state = state = t.zeros([1, 5])

# suppose action taken by your agent is 2, available actions are 0, 1, 2
action = t.full([1, 1], 2, dtype=t.int)

dqn.store_transition({
    "state": {"some_state": old_state},
    "action": {"action": action},
    "next_state": {"some_state": state},
    "reward": 0.1,
```

(continues on next page)

(continued from previous page)

```
    "terminal": False
})
```

Please take note that the sub key of attribute “state” and “next_state” must match the name of the state argument **“some_state”** in your Q network mentioned above. And the sub key of attribute “action” must be **“action”**.

We will come back to this seemingly strange name requirement in the *Buffer* section of *Data flow in machin*. For now, please make sure that shapes and dictionary keys of your tensors are **exactly the same** as the example.

Update

It is very easy to perform an update step, just call:

```
dqn.update()
```

on the framework instance you have just created.

Full training setup

With all the necessary parts, we can construct a full training program now:

```
from machin.frame.algorithms import DQN
from machin.utils.logging import default_logger as logger
import torch as t
import torch.nn as nn
import gym

# configurations
env = gym.make("CartPole-v0")
observe_dim = 4
action_num = 2
max_episodes = 1000
max_steps = 200
solved_reward = 190
solved_repeat = 5


# model definition
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, some_state):
        a = t.relu(self.fc1(some_state))
        a = t.relu(self.fc2(a))
        return self.fc3(a)


if __name__ == "__main__":
    q_net = QNet(observe_dim, action_num)
```

(continues on next page)

(continued from previous page)

```
    q_net_t = QNet(observe_dim, action_num)
    dqn = DQN(q_net, q_net_t,
              t.optim.Adam,
              nn.MSELoss(reduction='sum'))

    episode, step, reward_fulfilled = 0, 0, 0
    smoothed_total_reward = 0
    terminal = False

    while episode < max_episodes:
        episode += 1
        total_reward = 0
        terminal = False
        step = 0
        state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)

        while not terminal and step <= max_steps:
            step += 1
            with t.no_grad():
                old_state = state
                # agent model inference
                action = dqn.act_discrete_with_noise(
                    {"some_state": old_state}
                )
                state, reward, terminal, _ = env.step(action.item())
                state = t.tensor(state, dtype=t.float32).view(1, observe_dim)
                total_reward += reward

                dqn.store_transition({
                    "state": {"some_state": old_state},
                    "action": {"action": action},
                    "next_state": {"some_state": state},
                    "reward": reward,
                    "terminal": terminal or step == max_steps
                })

        # update, update more if episode is longer, else less
        if episode > 100:
            for _ in range(step):
                dqn.update()

        # show reward
        smoothed_total_reward = (smoothed_total_reward * 0.9 +
                                 total_reward * 0.1)
        logger.info("Episode {} total reward={:.2f}"
                    .format(episode, smoothed_total_reward))

        if smoothed_total_reward > solved_reward:
            reward_fulfilled += 1
            if reward_fulfilled >= solved_repeat:
                logger.info("Environment solved!")
                exit(0)
        else:
            reward_fulfilled = 0
```

And your Q network should will be successfully trained within about 300 episodes:

```
[2020-07-26 22:45:53,764] <INFO>:default_logger:Episode 226 total reward=188.18
[2020-07-26 22:45:54,405] <INFO>:default_logger:Episode 227 total reward=189.36
[2020-07-26 22:45:55,091] <INFO>:default_logger:Episode 228 total reward=190.42
[2020-07-26 22:45:55,729] <INFO>:default_logger:Episode 229 total reward=191.38
[2020-07-26 22:45:56,372] <INFO>:default_logger:Episode 230 total reward=192.24
[2020-07-26 22:45:57,012] <INFO>:default_logger:Episode 231 total reward=193.02
[2020-07-26 22:45:57,658] <INFO>:default_logger:Episode 232 total reward=193.72
[2020-07-26 22:45:57,658] <INFO>:default_logger:Environment solved!
```

3.1.2 Data flow in machin

Author: Muhan Li

Data flow is the major thing you should be very careful with while using the Machin library. Especially:

1. Data types
2. Tensor shapes
3. How to correctly store transitions.
4. How to correctly update your model.

If you are using the distributed algorithms, such as *A3C*, *IMPALA*, etc. You should additionally take care of:

5. How to setup the distributed world correctly.
6. How to setup the distributed framework correctly.
7. How to perform synchronization, pass data, between processes.

In this tutorial, we are not going to cover the distributed part, and will focus on the data flow in single agent RL algorithms.

The big picture

To give you a general idea of the data flow model in single agent RL algorithms, we will take the DQN framework as an example and use a diagram to illustrate everything:

Fig. 3.2: Data flow in DQN

There are mainly three types of arrows in the diagram:

1. The normal grey arrow: Represents data passed to functions by arguments, keyword arguments, etc. And data returned by functions.
2. The dashed gray arrow: The dashed gray arrow between the Q network and the target Q network means "soft_update", which updates the parameters of the target Q network by interpolating target Q and online Q.
3. The circle gray arrows: There are two circled gray arrows:

   N_State —() QNet_target
   State —() QNet

   These two circle gray arrows are special network calls named "safe_call", "safe_call"

is an enhanced keyword-argument-like caller, it will inspect arguments of the "forward"
function of your network, and fillin sub-keys of **major attributes** like **action**,
state, **next_state** from the batched sample. What major attributes are going to
be used by "safe_call" depends on the used RL framework and the model.

If sub-keys defined in major attributes are present in arguments of the "forward" function.
then the corresponding sub-values will be used. Otherwise they would be ignored.

"safe_call" will also inspect the input/output device of your model, or try to automatically
determine them, if they are not specified. Therefore, the tensors stored as sub-values of
these major attributes could be correctly moved to the target device.

Dive deeper

So what is happening under the hood exactly? How do we pass the observations and actions to the framework, then expect it to perform some magical operation and train our models behind the scenes? In this section, we are going to cover all of these questions in the following order, this order is also the direction of our data flow:

Transition -> Buffer -> Algorithm

Transition

Now let's take a step back and reexamine the process of a MDP (Markov Decision Process). A MDP process could be described as a chain of **transition steps**.

Fig. 3.3: MDP (Markov Decision Process)

In Machin, we store each transition step as a *`TransitionBase`* object, this class manages all data of a user defined transition step, by categorizing data into three types: major attribute, sub attribute and custom attribute.

1. Major attribute: `Dict[str, t.Tensor]`, used to describe complex state and action information.
2. Sub attributes: `Union[Scalar, t.Tensor]`, used to store less complex states such as reward, terminal status, etc.
3. Custom attributes: `Any`, used to store custom data structures describing environmental specific states, **must not have tensors** inside.

the default transition implementation is *`Transition`*, which have 5 attributes:

1. state (major attribute)
2. action (major attribute)
3. next_state (major attribute)
4. reward (sub attribute)
5. terminal (sub attribute)

Note:: The first dimension of **tensors** stored in major attributes and sub attributes must mean batch size (Scalar sub attributes are safe). Currently, the constructor of the default transition implementation *`Transition`* **requires batch size to be 1**, all algorithms are only tested and validated with batch size equals to 1. Scalar type custom attributes, like reward and terminal, will be considered as a tensor with shape `[1, 1]`.

Now that we have a very general transition data structure, which supports storing:

1. complex state information, such as visual(RGB-D), audio, physical(position, velocity, etc.), internal states of recurrent networks, etc.
2. complex action information, whether discreet or contiguous, single space or a combination of multitude of spaces, by storing them in different keys of the dictionary.
3. complex reward, whether scalar reward or vectorized reward.

We may use this class to store the transition steps of a full MDP. *Transition* can be constructed like:

```
old_state = state = t.zeros([1, 5])
action = t.zeros([1, 2])
transition = {
    "state": {"state": old_state},
    "action": {"action": action},
    "next_state": {"state": state},
    "reward": 1.0,
    "terminal": False
}
transition = Transition(**transition)
```

During *Transition* instance initialization, tensors stored in major and sub attributes will **be cloned then detached**, custom attributes will be **deep copied**.

Transition also supports `Transition.to()` method to move internal tensors to the target pytorch device.

Buffer

Buffers (replay memory) is one of the core parts of the Machin library. Machin provides a sophisticated but clear implementation of replay memory, to accommodate the needs of different frameworks. In *The big picture* section, we have showed that the buffer instance encapsulated in the DQN framework has two major APIs: "append" and "sample",

Append

Append is encapsulated by every framework, in their "store_*" APIs, some frameworks might will add new attributes to the constructed transition object in there "store_*" APIs, then call the "append" API of the buffer to add one or more transition objects to the buffer.

There are multiple buffer implementations, the basic *Buffer* class implements a simple ring buffer. *PrioritizedBuffer* extends on the the basic *Buffer* class with a prioritized weight tree. Distributed buffers are more interesting and complex because data are distributed on all process members.

In conclusion, the "append" API just stores one or more transition objects into the buffer, there are many internal events happening behind the scenes, and you need not worry about them.

Sample

Sampling is the first step performed in almost every frameworks, it may look like:

```
batch_size, (state, action, reward, next_state, terminal, others) = \
        self.replay_buffer.sample_batch(self.batch_size,
                                        concatenate_samples,
                                        sample_method="random_unique",
                                        sample_attrs=[
                                            "state", "action",
                                            "reward", "next_state",
                                            "terminal", "*"
                                        ])
```

What secret actions does this segment of code perform internally? Well, nothing other than "sampling" and "concatenation". Argument `sample_method` indicates the sample selection method, `sample_attrs` indicates which attributes of each sample we are going to acquire, "*" is a wildcard selector picking up all unspecified attributes.

Then what does "concatenation" mean? To put it simply, it will only affect "major attributes" and "sub attributes" of each sample, if you have specified `additional_concat_attrs`, then custom attributes can also be concatenated into a tensor. We may use a graph to explain this process happening in the basic *Buffer* class:

Fig. 3.4: Buffer sampling & concatenation process

Apart from the simplest *Buffer*, there is also *PrioritizedBuffer* (for prioritized experience replay), *DistributedBuffer* used in *IMPALA*, and *DistributedPrioritizedBuffer* used in *DQNApex* and *DDPGApex*.

We will not discuss about the internal implementations of distributed buffers here.

Algorithm

Now that algorithms have got samples from buffers, they can start training their models. The three types of model free RL algorithms supported by Machin have three respective internal data path.

For more detailed descriptions of data paths and model requirements of all RL algorithms, please refer to *Algorithm model requirements*.

In order to bridge the gap between models and algorithms, Machin uses a function named `safe_call()` to pass data from algorithms to your models, and uses different class methods defined in algorithms like *DDPG.action_transform_function()* to pack up raw data from your models before using them in the algorithm framework. With this design, Machin is able to achieve API consistency between algorithms while maintaining code simplicity.

Again, lets take the classic *DQN* framework as an example, we will use `mode="double"` here, so that a double DQN framework will be initialized, the models used in the *DQN* framework are Q networks. Q networks should accept a `state` and return `value` for each possible discreet action, ideally we would like to define the model according to this description **exactly**, like the one below, which accepts a single `state` argument in its `forward()` function, and returns a value tensor:

```
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
```

(continues on next page)

(continued from previous page)

```
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, state):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        return self.fc3(a)
```

And now in the *DQN.update()* method, we have sampled a batch of state, action, next_state etc, to train this Q network:

```
batch_size, (state, action, reward, next_state, terminal, others) = \
    self.replay_buffer.sample_batch(self.batch_size,
                                    concatenate_samples,
                                    sample_method="random_unique",
                                    sample_attrs=[
                                        "state", "action",
                                        "reward", "next_state",
                                        "terminal", "*"
                                    ])
```

where major attributes like state, action, next_state are **dictionaries of tensors**, while sub attributes like reward and terminal are two tensors of shape [batch_size, 1], we will ignore others for now, because if you are not inheriting from the DQN framework and write your own DQN.reward_func(), others does nothing.

In order to get the target Q value, which is used as an value estimation of the next state, we must use the Q network / the target Q network to criticize the sampled next_state:

```
q_value = self.criticize(state)
```

DQN.criticize() internally calls safe_call():

```
if use_target:
    return safe_call(self.qnet_target, state)
else:
    return safe_call(self.qnet, state)
```

safe_call() is a relatively complex function, it does the following things in general:

1. Check input & output device of your model, if they are not defined, try to
   automatically determine them by checking locations of all parameters.
2. Check argument names of the forward method of your model, this step will fail
   if it is not defined or your model is a JIT model complied by pytorch.
3. Try to resolve values of arguments by looking them up in the passed dictionaries,
   Additional keys in dictionaries that does not belong to args will be ignored.

Therefore, the sampled state must have the required key: "state", and "state" is the first argument (exclude self) of QNet.forwrad.

After forwarding, the Q network will pass predicted Q values back to the DQN framework, and data path is complete, the result Q values of next step will be passed to DQN.reward_func() to calculate target Q values, and then new values will be used to train the online Q network.

Summary

Generally speaking, Just treat all above process as an “advanced kwargs call”, During sampling, you will interact with your environment, and store some state tensors as values in a dictionary:

```
old_state = state = t.zeros([1, 5])
action = t.zeros([1, 2])
for _ in range(100):
    dqn.store_transition({
        "state": {"state": old_state},
        "action": {"action": action},
        "next_state": {"state": state},
        "reward": 1.0,
        "terminal": False
    })
```

Then during training, you will invoke the update method of your framework, and it will concatenate states, actions, and next states in the **first dimension**:

```
batch_size, (state, action, reward, next_state, terminal, others) = \
        self.replay_buffer.sample_batch(self.batch_size,
                                        concatenate_samples,
                                        sample_method="random_unique",
                                        sample_attrs=[
                                            "state", "action",
                                            "reward", "next_state",
                                            "terminal", "*"
                                        ])

# state = {"state": t.zeros([batch_size, 5])}
# action = {"action": t.zeros([batch_size, 2])}
# next_state = {"state": t.zeros([batch_size, 5])}
```

Then states, actions, and next states will be passed to your networks, **safely**, since tensors will be automatically moved to your model’s input device, and input device can be automatically determined or manually specified:

```
# DQN
q_value = self.criticize(state)

# DDPG
next_value = self.criticize(next_state, next_action, True)

# PPO
__, new_action_log_prob, new_action_entropy, *_ = \
                self.eval_act(state, action)
...
value = self.criticize(state)
```

And criticized values will be used to update your networks, done.

3.1.3 Parallel, distributed

Author: Muhan Li

This tutorial is going to give you a brief overview of how to write parallel & distributed programs, with Machin.

What are they?

Parallel means a set of computation processes executed simultaneously, whether synchronous or asynchronous.

Distributed means a system whose components are located on different entities, which are usually computers connected by networks.

Overview

Parallel

Traditional perspective

From the perspective of traditional parallel computation, there are many levels of parallelism, supported by Machin, based on PyTorch, from fine to coarse:

1. Element level parallelism, based on multidimensional tensor computations.
2. Task level parallelism, achieved by multi-threading, either provided by python threads, or the JIT fork mechanism of PyTorch (with no GIL).
3. Task level parallelism, achieved by multi-processing, either on the same node, or on different nodes.

For element level parallelism, we can either use existing tensor operators, or use more flexible operators such as `torch.einsum` to make customized operators, or write our own CUDA kernels. We can even use `torch.jit` to compile our models and get some performance improvements over plain python APIs. Machin doesn't provide any utility in this area.

For based task level parallelism, the basic python libraries, such as `threading` and `multiprocessing` already provide enough functions to achieve the latter two parallelisms. Machin provides the following enhancements:

1. Watch for exceptions happening in threads/processes.
2. Process/Thread pools with local function execution ability, accurate control over tensor serialization policy.
3. Process/Thread pools with contexts, allow users to pass hard-to-construct objects before executing tasks.
4. Inter-process queues with accurate control over tensor serialization policy.

Neural network perspective

From the perspective of neural networks, there are some parallelism paradigms we would like to achieve, with traditional parallel architectures:

1. Model level parallelism in small batch inference of many small models.
2. Model level parallelism in large batch inference of one potentially huge model.
3. Model level "parallelism" in storing an extremely large model across multiple devices or nodes.

Currently, there is no perfect way to deal with the first scenario, because threads in python are constrained by GIL, while processes are too slow. In *MADDPG*, Machin choose to utilize the JIT function provided by pytorch, and use compiled JIT models to work around the GIL restriction, this method is proved to have about 50% speed advantage over regular thread pools.

The second scenario could be dealt with `DistributedDataParallel` in PyTorch, by splitting the large batch into several smaller batches, then perform inference on different processes asynchronously.

The last scenario is also known as "model sharding", which means split a huge model up into several smaller models. It would be more favorable to users if this could be done automatically by the framework. However, due to the design of PyTorch, where tensors, not models, are real entities bound to device, it is not possible to achieve this function directly, with PyTorch, as of version 1.5.0. Machin currently does not provide automatic model sharding as well, but our internal implementation do support implementing such a feature, this feature might will be added in the future. Currently, Machin only provides automatic assignment of (splitted) models, with *`ModelAssigner`*.

Reinforcement learning perspective

When it comes to RL algorithms, these parallelisms are usually required:

1. Environment parallelism, where multiple same environments are executed synchronously in parallel, to produce larger batches of observations.
2. Agent parallelism, where multiple agents are learning synchronously or asynchronously, like *`A3C`*, *`DQNApex`*.
3. Agent parallelism in multi-agent algorithms, where multiple agents of different types are learning synchronously or asynchronously, like *`MADDPG`*

Machin provides parallel environment wrappers for the first scenario, like *`openai_gym.ParallelWrapperSubProc`*, which starts multiple worker processes, create an environment instance in each worker, then send commands and receive responses in batches.

The second scenario is more tricky, since agents are usually distributed across "parataxis" (same-level) processes, and on multiple nodes rather than "hypotaxis" sub-processes started in a process pool, on the same node. We will discuss this part in the *Distributed* section.

The third scenario depends on the RL algorithm framework, for `MADDPG`, each agent corresponds to a pair of separate actor and critic, in this case, only task level parallelism based threads could be used to solve the problem, because it is hard to create batches, caused by parameter and model architecture difference. But if we are using single agent RL algorithms such as `DDPG` to train a group of homogeneous agents, then batching is preferred due its efficientcy.

Distributed

Distributed is awesome, as well as extremely painful to deal with, hard to design, and even harder to debug, because applications are often required to have some crucial features like consistency, availability, partition-tolerance, and good performance.

Currently, since Machin relies on the PyTorch RPC framework, it does not provide any distribute mechanism able to guarantee any part of consistency, availability or partition-tolerance, due to some limitations in the PyTorch RPC framework, as of version 1.5.0.

What Machin provide is a more advanced set of RPC APIs: an implementation of RPC groups (namespace), on which you can register a service with `register` or share a resource with `pair`, like the code below:

```
self.group.pair(server_name,
                OrderedServerSimple(self.server_name, self.group))
self.group.register(server_name + "/_push_service", self._push_service)
self.group.register(server_name + "/_pull_service", self._pull_service)
```

This "DNS" like mechanism enables Machin to abstract away "name"s of processes, and a specific server process, instead, every process who wants to access the service/resource are faced with a registration table. This table could be different, depending on the actual process running the service, and the internal implementation of the service. With this design, Machin is able to provide some general distributed implementations such as *`DistributedBuffer`*, `DistributedPrioritizedBuffer`, *`PushPullGradServer`*, etc.

Apart from this, Machin just provides a thin layer of incapsulation over the somewhat complex APIs of `torch.distributed` and `torch.distributed.rpc`, to make them less confusing.

Examples

In order to fully understand all the functions provided `machin.parallel`, we should read some detailed use cases, this part **requires proficiency with but not a deep understanding of**:

1. `threading` library of python
2. `multiprocessing` library of python
3. `torch.distributed` module
4. `torch.distributed.rpc` module

If below examples are not enough for you, please refer to tests

Multi-threading examples

Waiting on multiple events

Detect exception thrown in a sub-thread

Using thread pools and context thread pools

Multi-processing examples

Serialization

Detect exception thrown in a sub-process

Using pools and context pools

Distributed examples

Distributed world and collective group

Distributed RPC examples

Distributed world and rpc group

A simple key-value server with version tracking

Model parallel examples

Assigning models automatically

3.1.4 Unleash distributed power

Author: Muhan Li

In this tutorial, we are going to try out some distributed single agent RL algorithms, they are:

1. `A3C`
2. `DQNApex` and `DDPGApex`
3. `IMPALA`

Now let's begin!

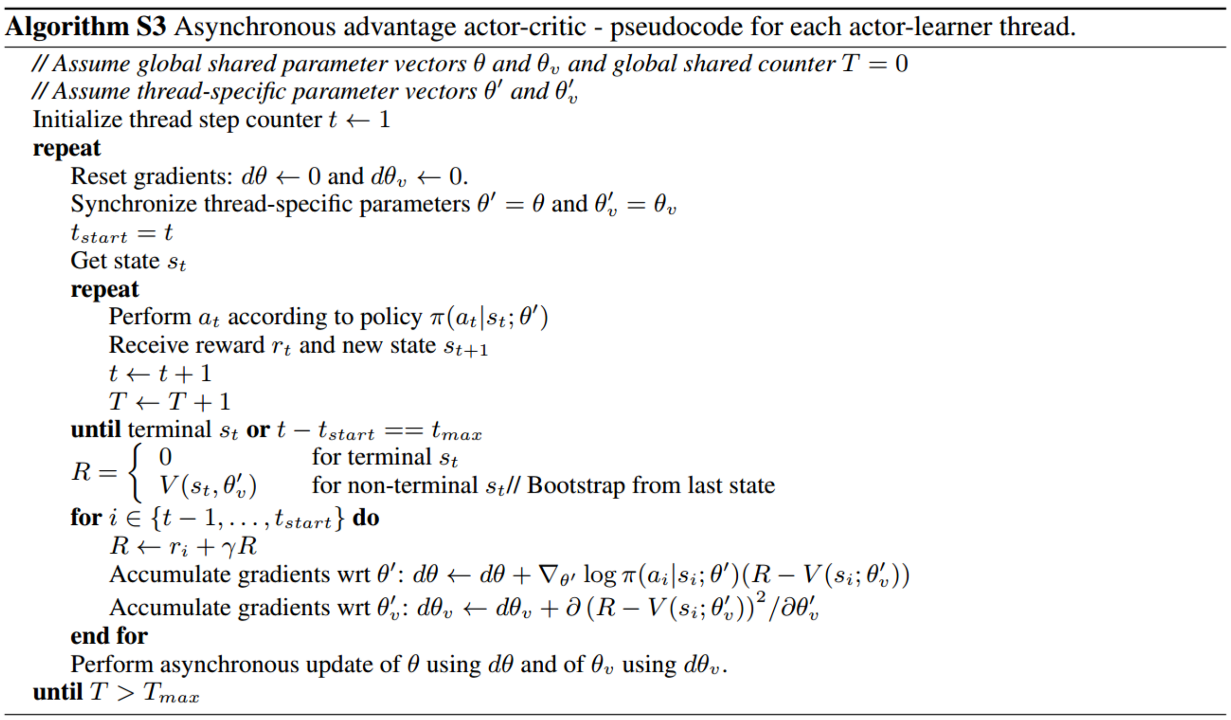
A3C

Full code: A3C code

A3C is the simplest distributed RL algorithm, among them all. We can describe its implementation with the following graph:

Fig. 3.5: A3C architecture

And a segment of pseudo code:



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \dots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

Fig. 3.6: A3C pesudo code

A3C is basically a bunch of A2C agents with a gradient reduction server. A3C(A2C) agents will interact with their environment simulators, train their local actors and critics, then push gradients to the gradient reduction server, the gradient reduction server will apply reduced gradients to its internal models (managed actor and critic network), then push the updated parameters to a key-value server. Agents will be able to pull the newest parameters and continue updating.

All A3C agents are fully asynchronous, gradient pushing & parameter pulling are asynchronous as well.

We will use the "CartPole-v0" environment from OpenAI Gym as an example, the actor network and critic network are as follows:

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_num):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        probs = t.softmax(self.fc3(a), dim=1)
        dist = Categorical(probs=probs)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act.flatten())
        return act, act_log_prob, act_entropy


class Critic(nn.Module):
    def __init__(self, state_dim):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state):
        v = t.relu(self.fc1(state))
        v = t.relu(self.fc2(v))
        v = self.fc3(v)
        return v
```

In order to initialize the `A3C` framework, we need to first initialize **the** distributed world:

```
# initlize distributed world first
_world = World(world_size=3, rank=rank,
               name=str(rank), rpc_timeout=20)
```

then provide a `PushPullGradServer` to it, Machin provides some helpful utility functions to aid inexperienced users initialize the distributed environment easily:

```
from machin.frame.helpers.servers import grad_server_helper
servers = grad_server_helper(
    [lambda: Actor(observe_dim, action_num),
```

(continues on next page)

(continued from previous page)

```
     lambda: Critic(observe_dim)],
    learning_rate=5e-3
)
```

Note all helpers from `machin.frame.helpers.servers` requires all processes in the distributed world to enter.

Finally we can compose the complete setup of *A3C*:

```
# initlize distributed world first
_world = World(world_size=3, rank=rank,
               name=str(rank), rpc_timeout=20)

actor = Actor(observe_dim, action_num)
critic = Critic(observe_dim)

# in all test scenarios, all processes will be used as reducers
servers = grad_server_helper(
    [lambda: Actor(observe_dim, action_num),
     lambda: Critic(observe_dim)],
    learning_rate=5e-3
)
a3c = A3C(actor, critic,
          nn.MSELoss(reduction='sum'),
          servers)
```

And start training, just as the A2C algorithm:

```
# manually control syncing to improve performance
a3c.set_sync(False)

# begin training
episode, step, reward_fulfilled = 0, 0, 0
smoothed_total_reward = 0
terminal = False

while episode < max_episodes:
    episode += 1
    total_reward = 0
    terminal = False
    step = 0

    state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)

    # manually pull the newest parameters
    a3c.manual_sync()
    tmp_observations = []
    while not terminal and step <= max_steps:
        step += 1
        with t.no_grad():
            old_state = state
            # agent model inference
            action = a3c.act({"state": old_state})[0]
            state, reward, terminal, _ = env.step(action.item())
            state = t.tensor(state, dtype=t.float32).view(1, observe_dim)
            total_reward += reward

            tmp_observations.append({
```

(continues on next page)

(continued from previous page)

```
                "state": {"state": old_state},
                "action": {"action": action},
                "next_state": {"state": state},
                "reward": reward,
                "terminal": terminal or step == max_steps
            })

    # update
    a3c.store_episode(tmp_observations)
    a3c.update()

    # show reward
    smoothed_total_reward = (smoothed_total_reward * 0.9 +
                             total_reward * 0.1)
    logger.info("Process {} Episode {} total reward={:.2f}"
                .format(rank, episode, smoothed_total_reward))

    if smoothed_total_reward > solved_reward:
        reward_fulfilled += 1
        if reward_fulfilled >= solved_repeat:
            logger.info("Environment solved!")
            # will cause torch RPC to complain
            # since other processes may have not finished yet.
            # just for demonstration.
            exit(0)
    else:
        reward_fulfilled = 0
```

A3C agents should will be successfully trained within about 1500 episodes, they converge much slower than A2C agents:

```
[2020-07-31 00:21:37,690] <INFO>:default_logger:Process 1 Episode 1346 total␣
↪reward=184.91
[2020-07-31 00:21:37,723] <INFO>:default_logger:Process 0 Episode 1366 total␣
↪reward=171.22
[2020-07-31 00:21:37,813] <INFO>:default_logger:Process 2 Episode 1345 total␣
↪reward=190.73
[2020-07-31 00:21:37,903] <INFO>:default_logger:Process 1 Episode 1347 total␣
↪reward=186.41
[2020-07-31 00:21:37,928] <INFO>:default_logger:Process 0 Episode 1367 total␣
↪reward=174.10
[2020-07-31 00:21:38,000] <INFO>:default_logger:Process 2 Episode 1346 total␣
↪reward=191.66
[2020-07-31 00:21:38,000] <INFO>:default_logger:Environment solved!
```

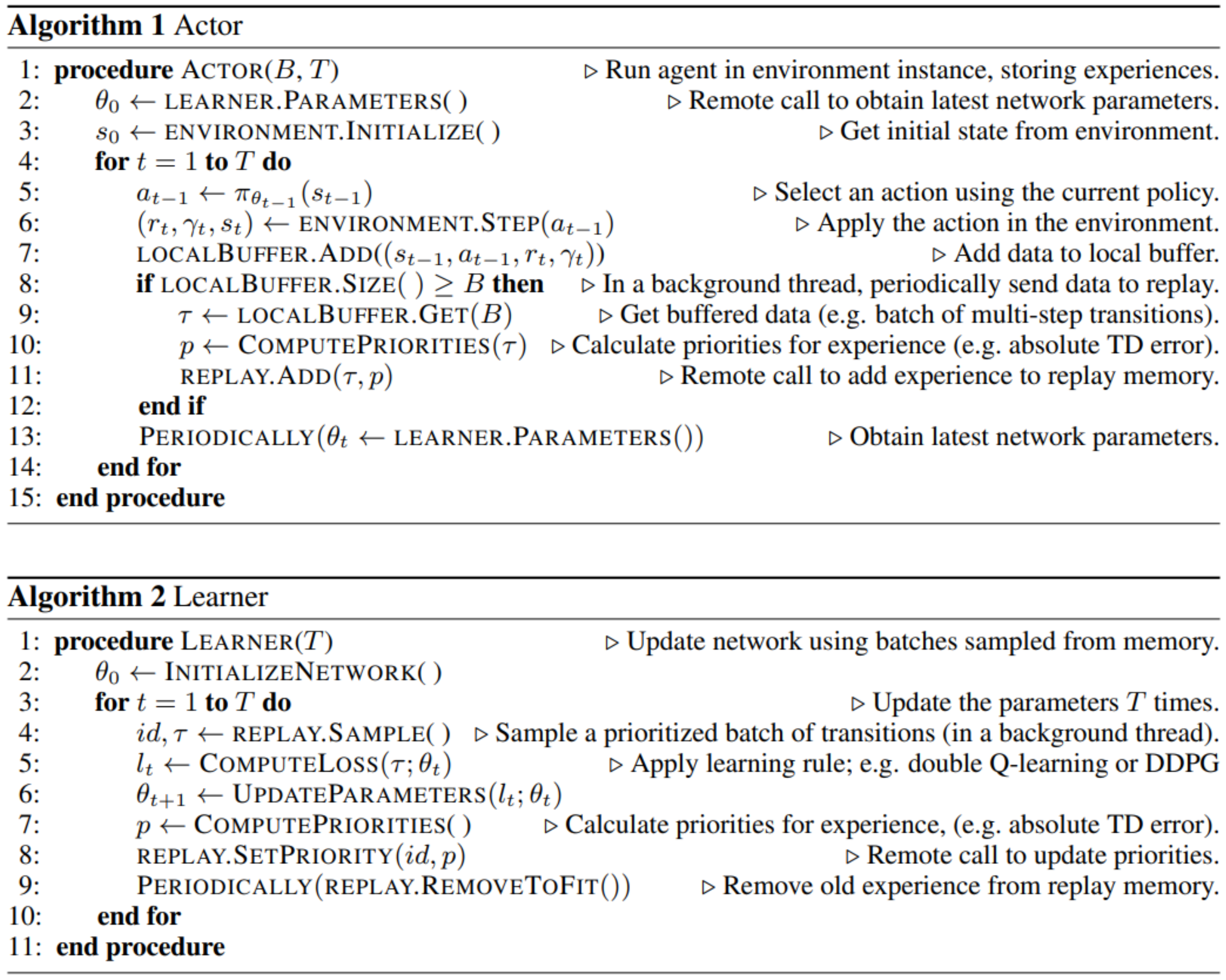
DQNApex and DDPGApex

Full code: DQNApex code

`DQNApex` and `DDPGApex` are actually based on the same architecture, therefore in this section, we are going to take `DQNApex` as an example, its distributed architecture could be described in the following graph:

Fig. 3.7: DQNApex architecture

And the pseudo code in essay:



Algorithm 1 Actor

```
1:  procedure ACTOR($B, T$)    ▷ Run agent in environment instance, storing experiences.
2:      $\theta_0 \leftarrow$ LEARNER.PARAMETERS( )    ▷ Remote call to obtain latest network parameters.
3:      $s_0 \leftarrow$ ENVIRONMENT.INITIALIZE( )    ▷ Get initial state from environment.
4:      for $t = 1$ to $T$ do
5:          $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$    ▷ Select an action using the current policy.
6:          $(r_t, \gamma_t, s_t) \leftarrow$ ENVIRONMENT.STEP($a_{t-1}$)    ▷ Apply the action in the environment.
7:          LOCALBUFFER.ADD($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$)    ▷ Add data to local buffer.
8:          if LOCALBUFFER.SIZE( ) $\geq B$ then    ▷ In a background thread, periodically send data to replay.
9:              $\tau \leftarrow$ LOCALBUFFER.GET($B$)    ▷ Get buffered data (e.g. batch of multi-step transitions).
10:             $p \leftarrow$ COMPUTEPRIORITIES($\tau$)    ▷ Calculate priorities for experience (e.g. absolute TD error).
11:             REPLAY.ADD($\tau, p$)    ▷ Remote call to add experience to replay memory.
12:         end if
13:         PERIODICALLY($\theta_t \leftarrow$ LEARNER.PARAMETERS())    ▷ Obtain latest network parameters.
14:     end for
15: end procedure
```

Algorithm 2 Learner

```
1:  procedure LEARNER($T$)    ▷ Update network using batches sampled from memory.
2:      $\theta_0 \leftarrow$ INITIALIZENETWORK( )
3:      for $t = 1$ to $T$ do    ▷ Update the parameters $T$ times.
4:          $id, \tau \leftarrow$ REPLAY.SAMPLE( )    ▷ Sample a prioritized batch of transitions (in a background thread).
5:          $l_t \leftarrow$ COMPUTELOSS($\tau; \theta_t$)    ▷ Apply learning rule; e.g. double Q-learning or DDPG
6:          $\theta_{t+1} \leftarrow$ UPDATEPARAMETERS($l_t; \theta_t$)
7:          $p \leftarrow$ COMPUTEPRIORITIES( )    ▷ Calculate priorities for experience, (e.g. absolute TD error).
8:          REPLAY.SETPRIORITY($id, p$)    ▷ Remote call to update priorities.
9:          PERIODICALLY(REPLAY.REMOVETOFIT())    ▷ Remove old experience from replay memory.
10:     end for
11: end procedure
```

Fig. 3.8: DQN-Apex pesudo code

The Apex architecture decouples the sampling and updating process with the prioritized replay buffer. There could be several implementations, such as:

1. using a central replay buffer on a single process
2. using a distributed buffer, with a central stopping signal.
3. using a distributed buffer, each buffer with a separate lock.

Machin choose the third implementation because it is most efficient:

#1 is slow because each appending requires a RPC process to update the global weight tree, and it also doesn't scale when the number of workers(samplers) grows too large, such as 100+ workers.

The central lock used in #2 is meant to protect the importance sampling-updating process, so each buffer maintains a local weight tree, during sampling, the learner will signal "STOP" to all workers, and signal "START" to all workers when importance weight update is completed, however, this design does not truly decouples learning and sampling, therefore most of the time workers are just hanging and wait for the learner to complete updaing.

#3 design is the best, because each append operation is completely local (only needs to acquire a local lock), and global sampling is complete decoupled from local appending (because lock are immediately released after returning sampled data, and not till update complete) as show in figure Fig. 3.7.

However, it could be very tricky to implement this process, because appending is still happening after sampling and before the learner finishes updating importance sampling weights (is-weights), therefore Machin uses a **taint table**, which is essentially a table full of auto increment counters, each counter maps to an entry slot in the lower ring buffer, and is incremented if the entry has been replaced with new entries. This replacement should will not be very often if the buffer has enough space, (100000+), therefore guarantee the correctness of importance weight update.

There is one thing to note, it could be indefinitely long for learner to calculate the virtual global weight sum tree using the root node of all local weight sum trees as leaves, therefore at the time of sampling, the used weight sum of local trees is already outdated, and sampling probability of each tree should have changed. However, if the size of each local buffer is large enough, then the ratio of difference between the old collected local weight sums and current weight sums should be acceptable.

Now that we know how the Apex framework is designed, we may try an example. We will use the "CartPole-v0" environment from OpenAI Gym as an example, the Q networks is as follows:

```python
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, some_state):
        a = t.relu(self.fc1(some_state))
        a = t.relu(self.fc2(a))
        return self.fc3(a)
```

Because apex frameworks relies on the `DistributedPrioritizedBuffer`, the learner needs to know the position and service name of each local buffer, as show in figure Fig. 3.7, in order to initialize the `Apex` framework, we need to provide a RPC process group, where all learner(s) and workers will live on:

```python
apex_group = world.create_rpc_group("apex", ["0", "1", "2"])
```

And we will also provide a model server on which learner(s) will store the newest parameters and workers will pull the newest parameters from the server. This kind of parameter server is different from the `PushPullGradServer` used above, and we will name it as `PushPullModelServer`, Currently, each `PushPullModelServer` **only**

manages one model per server instance, and since there is only one model needs to be shared in DQN (the online Q network), we only need one model server instance:

```
servers = model_server_helper(model_num=1)
dqn_apex = DQNApex(q_net, q_net_t,
                   t.optim.Adam,
                   nn.MSELoss(reduction='sum'),
                   apex_group,
                   (servers[0],),
                   replay_device=c.device,
                   replay_size=c.replay_size)
```

The tasks of learner(s) and workers are quite a bit different, since learner(s) only needs to update their internal models repeatedly, using samples from workers' buffers, and workers only need to do update-sample-update-sample. . . , they will run different branches in the main program.

Maybe you want to ask, why are we using **learner(s)**, isn't the original essay stating that there is only one learner and multiple workers? The answer is: Machin supports using *DistributedDataParallel* (*DataParallel* is also supported) from PyTorch inside DQNApex, so that you may distribute the updating task across **multiple learner processes**, if your models is way too **large** to be computed by a single process. It is not sensible to using this technique with small models, but for pure demonstration purpose, we will use it here:

```
if rank in (2, 3):
    # learner_group.group is the wrapped torch.distributed.ProcessGroup
    learner_group = world.create_collective_group(ranks=[2, 3])

    # wrap the model with DistributedDataParallel
    # if current process is learner process 2 or 3
    q_net = DistributedDataParallel(module=QNet(observe_dim, action_num),
                                    process_group=learner_group.group)
    q_net_t = DistributedDataParallel(module=QNet(observe_dim, action_num),
                                      process_group=learner_group.group)
else:
    q_net = QNet(observe_dim, action_num)
    q_net_t = QNet(observe_dim, action_num)

# we may use a smaller batch size to train if we are using
# DistributedDataParallel
dqn_apex = DQNApex(q_net, q_net_t,
                   t.optim.Adam,
                   nn.MSELoss(reduction='sum'),
                   apex_group,
                   (servers[0],),
                   batch_size=50)
```

The main part of the training process is as follows:

```
# synchronize all processes in the group, make sure
# distributed buffer has been created on all processes
# in apex_group
apex_group.barrier()

# manually control syncing to improve performance
dqn_apex.set_sync(False)
if rank in (0, 1):
    # Process 0 and 1 are workers(samplers)
    # begin training
    episode, step, reward_fulfilled = 0, 0, 0
```

(continues on next page)

(continued from previous page)

```
    smoothed_total_reward = 0

    while episode < max_episodes:
        # sleep to wait for learners keep up
        sleep(0.1)
        episode += 1
        total_reward = 0
        terminal = False
        step = 0

        state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)

        # manually pull the newest parameters
        dqn_apex.manual_sync()
        while not terminal and step <= max_steps:
            step += 1
            with t.no_grad():
                old_state = state
                # agent model inference
                action = dqn_apex.act_discrete_with_noise(
                    {"state": old_state}
                )
                state, reward, terminal, _ = env.step(action.item())
                state = t.tensor(state, dtype=t.float32)\
                    .view(1, observe_dim)
                total_reward += reward

                dqn_apex.store_transition({
                    "state": {"state": old_state},
                    "action": {"action": action},
                    "next_state": {"state": state},
                    "reward": reward,
                    "terminal": terminal or step == max_steps
                })

        smoothed_total_reward = (smoothed_total_reward * 0.9 +
                                 total_reward * 0.1)
        logger.info("Process {} Episode {} total reward={:.2f}"
                    .format(rank, episode, smoothed_total_reward))

        if smoothed_total_reward > solved_reward:
            reward_fulfilled += 1
            if reward_fulfilled >= solved_repeat:
                logger.info("Environment solved!")

                # will cause torch RPC to complain
                # since other processes may have not finished yet.
                # just for demonstration.
                exit(0)
        else:
            reward_fulfilled = 0

elif rank in (2, 3):
    # wait for enough samples
    while dqn_apex.replay_buffer.all_size() < 500:
        sleep(0.1)
    while True:
```

(continues on next page)

(continued from previous page)

```
        dqn_apex.update()
```

Result:

```
[2020-08-01 12:51:04,323] <INFO>:default_logger:Process 1 Episode 756 total␣
↪reward=192.42
[2020-08-01 12:51:04,335] <INFO>:default_logger:Process 0 Episode 738 total␣
↪reward=187.58
[2020-08-01 12:51:04,557] <INFO>:default_logger:Process 1 Episode 757 total␣
↪reward=193.17
[2020-08-01 12:51:04,603] <INFO>:default_logger:Process 0 Episode 739 total␣
↪reward=188.72
[2020-08-01 12:51:04,789] <INFO>:default_logger:Process 1 Episode 758 total␣
↪reward=193.86
[2020-08-01 12:51:04,789] <INFO>:default_logger:Environment solved!
```

IMPALA

Full code: IMPALA code

The *IMPALA* algorithm has the same parallel architecture as *DQNApex* and *DDPGApex* do, the only difference is that the internal distributed buffer it is using is a simple distributed buffer, with no distributed prioritized tree:

Fig. 3.9: Impala architecture

The pseudo code of the *IMPALA* algorithm is as follows:

where the $VTrace$ function is defined as:

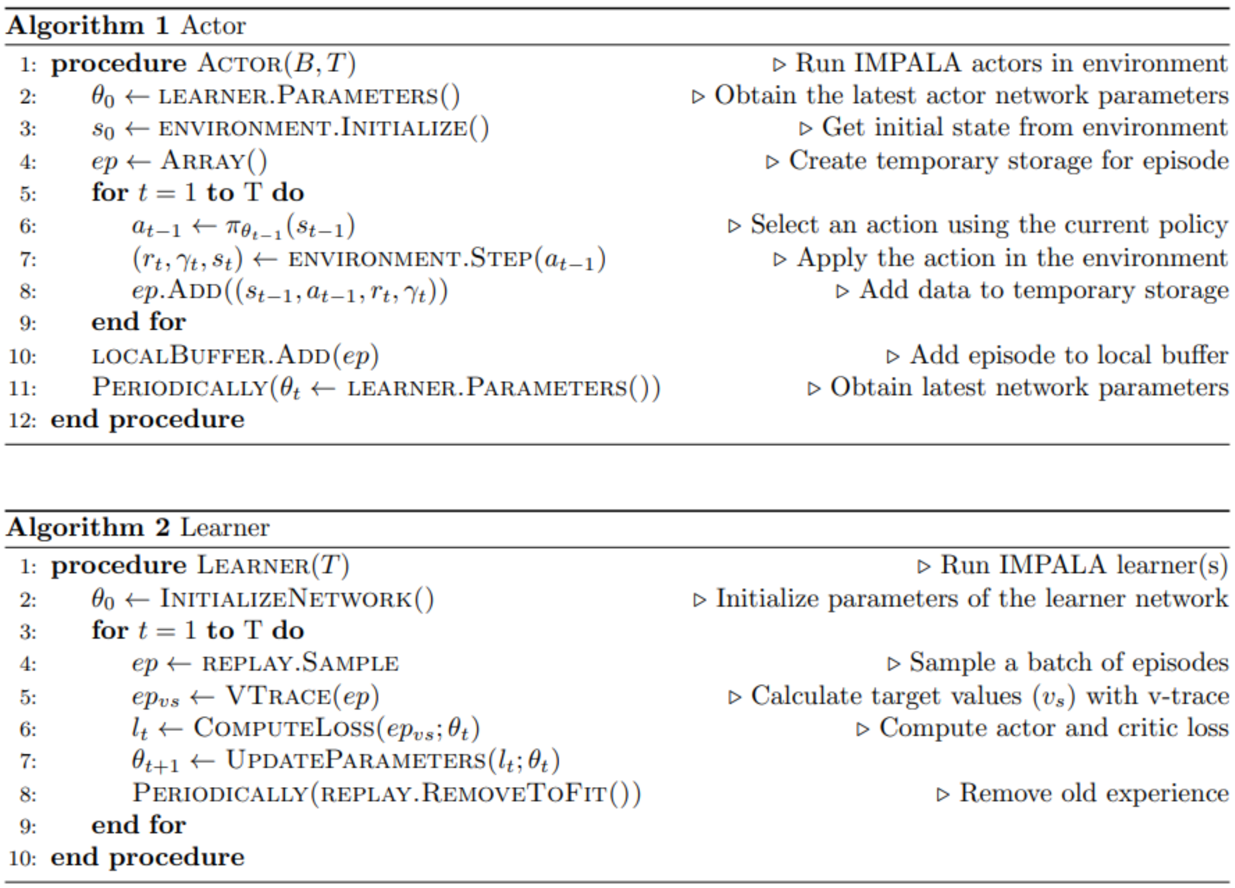
$$v_s = V(X_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} (\prod_{i=s}^{t-1} c_i) \delta_t V$$
$$\delta_t V = \rho_t (r_t + \gamma V(x_{t+1}) - V(x_t))$$
$$\rho_t = min(\rho_{max}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$$
$$c_i = min(c_{max}, \frac{\pi(a_i|x_i)}{\mu(a_i|x_i)})$$

In order to initialize the *IMPALA* framework, we need to pass two accessors to two individual `PushPullModelServer` to the framework, and take note that *IMPALA* **does not support** storing a single step**, since v-trace calculation requires sampling complete episodes, all transition objects in the *IMPALA* buffer **are episodes rather than steps**, therefore the used `batch_size` is set to `2`, which is much smaller then `50` used in *DQNApex*:

```
if rank in (2, 3):
    # learner_group.group is the wrapped torch.distributed.ProcessGroup
    learner_group = world.create_collective_group(ranks=[2, 3])

    # wrap the model with DistributedDataParallel
    # if current process is learner process 2 or 3
    actor = DistributedDataParallel(module=Actor(observe_dim, action_num),
                                    process_group=learner_group.group)
    critic = DistributedDataParallel(module=Critic(observe_dim),
```

(continues on next page)



Algorithm 1 Actor

```
1: procedure ACTOR($B, T$)                                   ▷ Run IMPALA actors in environment
2:     $\theta_0 \leftarrow$ LEARNER.PARAMETERS()               ▷ Obtain the latest actor network parameters
3:     $s_0 \leftarrow$ ENVIRONMENT.INITIALIZE()                ▷ Get initial state from environment
4:     $ep \leftarrow$ ARRAY()                                  ▷ Create temporary storage for episode
5:     for $t = 1$ to T do
6:         $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$     ▷ Select an action using the current policy
7:         $(r_t, \gamma_t, s_t) \leftarrow$ ENVIRONMENT.STEP($a_{t-1}$)   ▷ Apply the action in the environment
8:         $ep$.ADD($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$)         ▷ Add data to temporary storage
9:     end for
10:    LOCALBUFFER.ADD($ep$)                                    ▷ Add episode to local buffer
11:    PERIODICALLY($\theta_t \leftarrow$ LEARNER.PARAMETERS())  ▷ Obtain latest network parameters
12: end procedure
```

Algorithm 2 Learner

```
1: procedure LEARNER($T$)                                    ▷ Run IMPALA learner(s)
2:     $\theta_0 \leftarrow$ INITIALIZENETWORK()                ▷ Initialize parameters of the learner network
3:     for $t = 1$ to T do
4:         $ep \leftarrow$ REPLAY.SAMPLE                        ▷ Sample a batch of episodes
5:         $ep_{vs} \leftarrow$ VTRACE($ep$)                     ▷ Calculate target values ($v_s$) with v-trace
6:         $l_t \leftarrow$ COMPUTELOSS($ep_{vs}; \theta_t$)     ▷ Compute actor and critic loss
7:         $\theta_{t+1} \leftarrow$ UPDATEPARAMETERS($l_t; \theta_t$)
8:         PERIODICALLY(REPLAY.REMOVETOFIT())                   ▷ Remove old experience
9:     end for
10: end procedure
```

Fig. 3.10: Impala pesudo code

(continued from previous page)

```
                                        process_group=learner_group.group)
else:
    actor = Actor(observe_dim, action_num)
    critic = Critic(observe_dim)

# we may use a smaller batch size to train if we are using
# DistributedDataParallel

# note: since the impala framework is storing a whole
# episode as a single sample, we should wait for a smaller number
impala = IMPALA(actor, critic,
                t.optim.Adam,
                nn.MSELoss(reduction='sum'),
                impala_group,
                servers,
                batch_size=2)
```

The main part of the training process is almost the same as that of *DQNApex*:

```
# synchronize all processes in the group, make sure
# distributed buffer has been created on all processes in apex_group
impala_group.barrier()

# manually control syncing to improve performance
impala.set_sync(False)
if rank in (0, 1):
    # Process 0 and 1 are workers(samplers)
    # begin training
    episode, step, reward_fulfilled = 0, 0, 0
    smoothed_total_reward = 0

    while episode < max_episodes:
        # sleep to wait for learners keep up
        sleep(0.1)
        episode += 1
        total_reward = 0
        terminal = False
        step = 0

        state = t.tensor(env.reset(), dtype=t.float32).view(1, observe_dim)

        # manually pull the newest parameters
        impala.manual_sync()
        tmp_observations = []
        while not terminal and step <= max_steps:
            step += 1
            with t.no_grad():
                old_state = state
                # agent model inference
                action, action_log_prob, *_ = \
                    impala.act({"state": old_state})
                state, reward, terminal, _ = env.step(action.item())
                state = t.tensor(state, dtype=t.float32) \
                    .view(1, observe_dim)
                total_reward += reward

                tmp_observations.append({
```

(continues on next page)

(continued from previous page)

```
                    "state": {"state": old_state},
                    "action": {"action": action},
                    "next_state": {"state": state},
                    "reward": reward,
                    "action_log_prob": action_log_prob.item(),
                    "terminal": terminal or step == max_steps
                })

        impala.store_episode(tmp_observations)
        smoothed_total_reward = (smoothed_total_reward * 0.9 +
                                 total_reward * 0.1)
        logger.info("Process {} Episode {} total reward={:.2f}"
                    .format(rank, episode, smoothed_total_reward))

        if smoothed_total_reward > solved_reward:
            reward_fulfilled += 1
            if reward_fulfilled >= solved_repeat:
                logger.info("Environment solved!")

                # will cause torch RPC to complain
                # since other processes may have not finished yet.
                # just for demonstration.
                exit(0)
        else:
            reward_fulfilled = 0

elif rank in (2, 3):
    # wait for enough samples
    # note: since the impala framework is storing a whole
    # episode as a single sample, we should wait for a smaller number
    while impala.replay_buffer.all_size() < 5:
        sleep(0.1)
    while True:
        impala.update()
```

IMPALA converges very fast, usually within 150 episodes:

```
[2020-08-01 23:25:34,861] <INFO>:default_logger:Process 1 Episode 72 total reward=185.
↪32
[2020-08-01 23:25:35,057] <INFO>:default_logger:Process 1 Episode 73 total reward=186.
↪79
[2020-08-01 23:25:35,060] <INFO>:default_logger:Process 0 Episode 70 total reward=193.
↪28
[2020-08-01 23:25:35,257] <INFO>:default_logger:Process 1 Episode 74 total reward=188.
↪11
[2020-08-01 23:25:35,261] <INFO>:default_logger:Process 0 Episode 71 total reward=193.
↪95
[2020-08-01 23:25:35,261] <INFO>:default_logger:Environment solved!
```

3.1.5 Recurrent networks

Author: Muhan Li

Full code 1: DQN

Full code 2: DRQN

Full code 3: PPO

Full code 4: RPPO

Preface

In this tutorial, we are going to try and implement the recurrent architecture in DQN and PPO architecture, the original architecture of "DRQN" was described in Deep Recurrent Q-Learning for Partially Observable MDPs, for the sake of simplicity and this tutorial will discard the CNN part used to process Atari game screens, instead, we will directly access the internal 128 bytes of RAM of tested Atari games.

Now, in order to implement the recurrent architecture, we should have a solid grasp of the following related aspects in advance:

1. DQN framework
2. Recurrent neural networks, LSTM and GRU
3. MDP and POMDP

Recurrent networks were introduced into the reinforcement learning field to deal with POMDP models, in which agents are not able to observe the full state of the environment, and they have to rely on their internal memories of their past observations. The essay used Atari games as the benchmark suite, they compared DRQN with DQN in multiple scenarios, and shows that DRQN has significant advantage over DQN in the frostbite game, while performing about as good as / fail to compete with DQN in many other Atari games.

For offline reinforcement learning frameworks relying on the "replay memory", like `DQN` and `DDPG`, the tricky bit is that by the time of sampling, the trained models (online network and target network) are already different from the model used to interact with the environment and produce samples, authors of the essay suggested two ways of updating, both ways requires to provide a contiguous period of samples to the network to compute hidden states, and back propagation through time.

For online reinforcement learning frameworks such as `A2C` and `PPO` with no replaying mechanism, there is no need to specifically recalculate hidden states, because by the time of training, the stored samples are still generated by a actor network equal to (when update iteration=0)/ very close to (when update iteration > 0) the trained network. Therefore, hidden states can be stored along with other observations,

We are going to show the detailed recurrent implementations in the above two reinforcement learning categories, using `DQNPer` and `PPO` respectively.

Network architecture

Used network architectures are in the following graph:

Fig. 3.11: Network architectures

Design overview

DQN and DRQN

Warning: Compared to the implementation provided in this repo, our implementation of DRQN is **significantly more inefficient**, and potentially has **different result** because:

1. Duplicate states are stored for (history_depth - 1) times.
2. Only the last step in the bootstrapped random updates is performed, Q values evaluated in previous steps are not used.

You may implement your own framework to overcome these shortcomings using the utilities provided by Machin.

Authors of the original paper choose to train the LSTM layer along with the CNN layers, in order to deal with the "hidden state" input of the LSTM layer, they proposed two methods:

1. Bootstrapped sequential updates
2. Bootstrapped random updates

"Sequential updates" use the recurrent Q network to train through a whole episode, then BPTT (back propagate through time). "Random updates" samples a random period of length "unrolled_time_steps" instead of a whole episode, other details are the same.

In order to achieve this with the *DQNPer* framework, we will have to store the history observations for each transition, since the internal replay buffer does not store episodic boundaries between transitions:

```
old_history = history.get()
new_history = history.append(state).get()
drqn.store_transition({
    "state": {"history_mem": old_history},
    "action": {"action": action},
    "next_state": {"history_mem": new_history},
    "reward": reward,
    "terminal": terminal
})
```

Then we will also have to define two branches inside the `forward` function of our recurrent Q network, one branch for normal action sampling and another branch for training:

```
def forward(self, mem=None, hidden=None, history_mem=None):
    if mem is not None:
        # use `mem`, `hidden`, in sampling
        ...
    else:
        # use `history_mem`, in updating
        ...
```

We will show the details in the implementation section of this tutorial.

PPO and RPPO

PPO is much easier to deal with, if we **do not BPTT**, then we just need to store hidden states along with other states like:

```
tmp_observations.append({
    "state": {"mem": old_state, "hidden": old_hidden},
    "action": {"action": action},
    "next_state": {"mem": state, "hidden": hidden},
    "reward": reward,
    "terminal": terminal
})
```

However, not using BPTT will lose most benefits of recurrence, if you would like to use this method, then you need to implement your own framework sampling entire episodes and not timesteps from the replay buffer. Then zero-pad the sampled episodes so they are all the same length. Finally let your recurrent network go through the sampled episodes and calculate log probs/actions/hidden states. You may refer to this repo for more information.

Implementation

History

We are going to design a *History* class which allow users to store new states by *append()* and returns a fixed-length trajectory by *get()*, if there are not enough states to form a complete trajectory, then zero will be used to form paddings:

```
class History:
def __init__(self, history_depth, state_shape):
    self.history = [t.zeros(state_shape) for _ in range(history_depth)]
    self.state_shape = state_shape

def append(self, state):
    assert (t.is_tensor(state) and
            state.dtype == t.float32 and
            tuple(state.shape) == self.state_shape)
    self.history.append(state)
    self.history.pop(0)
    return self

def get(self):
    # size: (1, history_depth, ...)
    return t.cat(self.history, dim=0).unsqueeze(0)
```

DQN

The Q network will accept a transition trajectory of length *history_depth*, and returns a Q value tensor:

```
class QNet(nn.Module):
def __init__(self, history_depth, action_num):
    super(QNet, self).__init__()
    self.fc1 = nn.Linear(128 * history_depth, 256)
    self.fc2 = nn.Linear(256, 256)
    self.fc3 = nn.Linear(256, action_num)
```

(continues on next page)

(continued from previous page)

```
def forward(self, mem):
    return self.fc3(t.relu(
        self.fc2(t.relu(
            self.fc1(mem.flatten(start_dim=1))
        ))
    ))
```

In order to provide sampled trajectories to the network, we just need to store "history" instead of "state":

```
while not terminal:
    step += 1
    with t.no_grad():
        history.append(state)
        # agent model inference
        action = dqn.act_discrete_with_noise(
            {"mem": history.get()}
        )

        # info is {"ale.lives": self.ale.lives()}, not used here
        state, reward, terminal, _ = env.step(action.item())
        state = convert(state)
        total_reward += reward
        old_history = history.get()
        new_history = history.append(state).get()
        dqn.store_transition({
            "state": {"mem": old_history},
            "action": {"action": action},
            "next_state": {"mem": new_history},
            "reward": reward,
            "terminal": terminal
        })
```

DRQN

DRQN network is a little bit more complex:

```
class RecurrentQNet(nn.Module):
    def __init__(self, action_num):
        super(RecurrentQNet, self).__init__()
        self.gru = nn.GRU(128, 256, batch_first=True)
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, action_num)

    def forward(self, mem=None, hidden=None, history_mem=None):
        if mem is not None:
            # in sampling
            a, h = self.gru(mem.unsqueeze(1), hidden)
            return self.fc2(t.relu(
                self.fc1(t.relu(
                    a.flatten(start_dim=1)
                ))
            )), h
        else:
            # in updating
            batch_size = history_mem.shape[0]
```

(continues on next page)

(continued from previous page)

```
            seq_length = history_mem.shape[1]
            hidden = t.zeros([1, batch_size, 256],
                             device=history_mem.device)
            for i in range(seq_length):
                _, hidden = self.gru(history_mem[:, i].unsqueeze(1), hidden)
            # a[:, -1] = h
            return self.fc2(t.relu(
                self.fc1(t.relu(
                    hidden.transpose(0, 1).flatten(start_dim=1)
                ))
            ))
```

As you can see, the forward method is divided into two parts, the first part is for normal acting, where users will pass hidden states to the network manually and get actions during sampling:

```
hidden = t.zeros([1, 1, 256])
state = convert(env.reset())
history = History(history_depth, (1, 128))

while not terminal:
    step += 1
    with t.no_grad():
        old_state = state
        history.append(state)
        # agent model inference
        action, hidden = drqn.act_discrete_with_noise(
            {"mem": old_state, "hidden": hidden}
        )
```

The second part is used during updating, where the *DQNPer* framework will provide a batch of trajectories to the network and get Q value tensor for **last state** in each trajectory:

```
old_history = history.get()
new_history = history.append(state).get()
drqn.store_transition({
    "state": {"history_mem": old_history},
    "action": {"action": action},
    "next_state": {"history_mem": new_history},
    "reward": reward,
    "terminal": terminal
})
```

PPO

PPO is the same as *DQN*, the actor network and critic network will accept a trajectory and return an action/value:

```
class Actor(nn.Module):
    def __init__(self, history_depth, action_num):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(128 * history_depth, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, action_num)

    def forward(self, mem, action=None):
        a = t.relu(self.fc1(mem.flatten(start_dim=1)))
```

(continues on next page)

(continued from previous page)

```
        a = t.relu(self.fc2(a))
        probs = t.softmax(self.fc3(a), dim=1)
        dist = Categorical(probs=probs)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act.flatten())
        return act, act_log_prob, act_entropy


class Critic(nn.Module):
    def __init__(self, history_depth):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(128 * history_depth, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 1)

    def forward(self, mem):
        v = t.relu(self.fc1(mem.flatten(start_dim=1)))
        v = t.relu(self.fc2(v))
        v = self.fc3(v)
        return v
```

RPPO

RPPO actor will accept a hidden state, critic will accept one state instead of a trajectory comprised of multiple states:

```
class RecurrentActor(nn.Module):
    def __init__(self, action_num):
        super(RecurrentActor, self).__init__()
        self.gru = nn.GRU(128, 256, batch_first=True)
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, action_num)

    def forward(self, mem, hidden, action=None):
        hidden = hidden.transpose(0, 1)
        a, hidden = self.gru(mem.unsqueeze(1), hidden)
        a = self.fc2(t.relu(
            self.fc1(t.relu(a.flatten(start_dim=1)))
        ))
        probs = t.softmax(a, dim=1)
        dist = Categorical(probs=probs)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act.flatten())
        return act, act_log_prob, act_entropy, hidden


class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.fc1 = nn.Linear(128, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 1)

    def forward(self, mem):
        v = t.relu(self.fc1(mem))
        v = t.relu(self.fc2(v))
        v = self.fc3(v)
        return v
```

Test results

Note: These test results are put here for pure demonstration purpose, they are not intended for statistical comparision.

It seems that the DRQN implementation is extremely unstable, DQN is not quite stable as well, especially when *history_depth > 1*. PPO learns a little bit better than DQN when *history_depth = 1*, but it is able to cross the 300 boundary when *history_depth = 4*, RPPO is also able to overcome the 300 boundary after 6000 episodes. Since learning rate is fine tuned, performance of all frameworks drop considerably after some point.

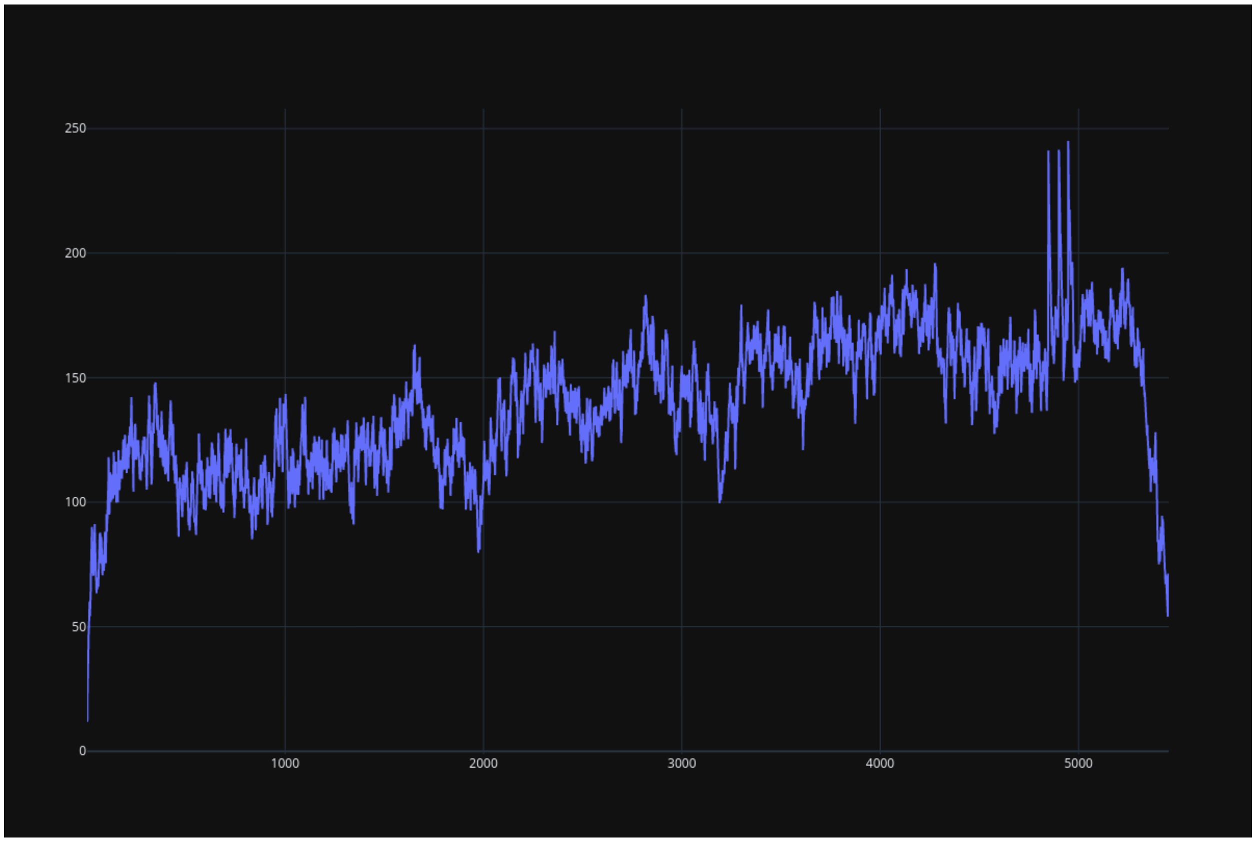


Fig. 3.12: DQN result

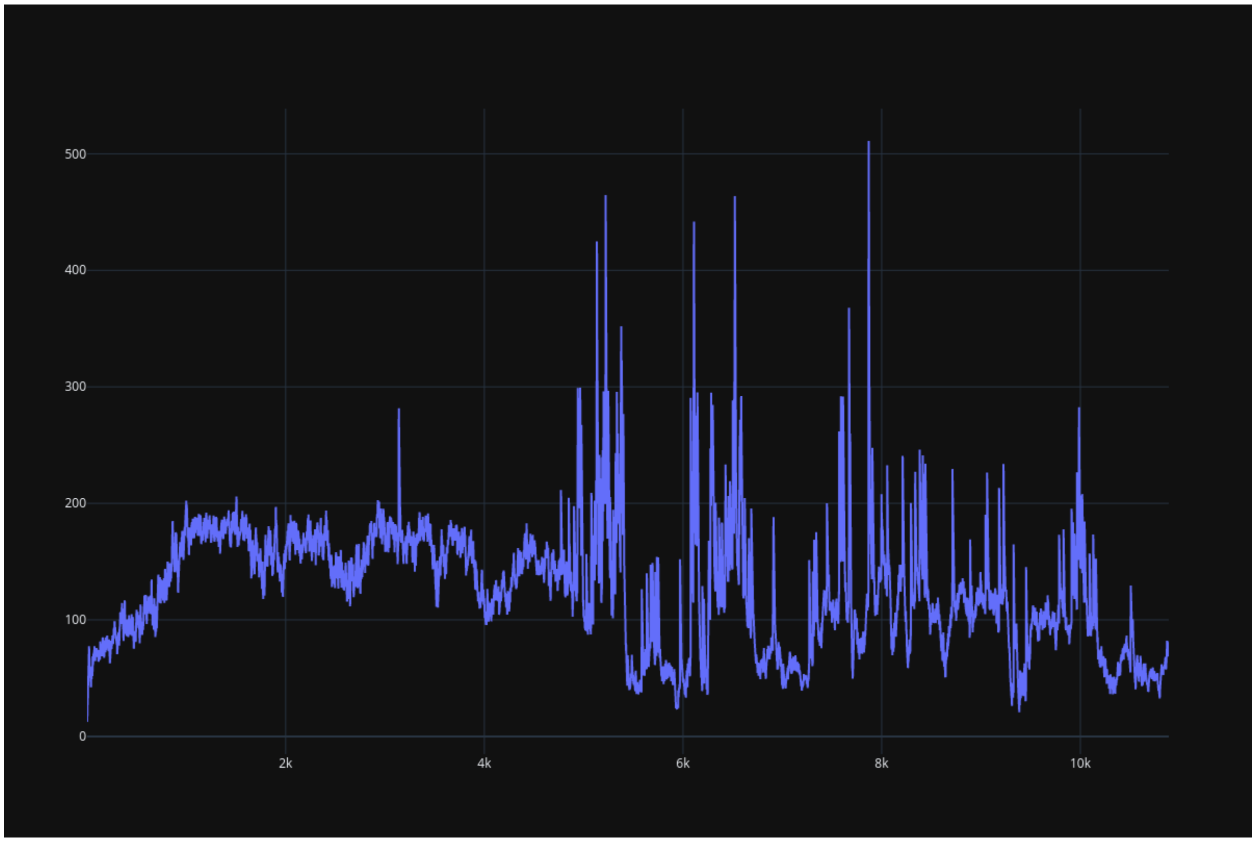


Fig. 3.13: DRQN result

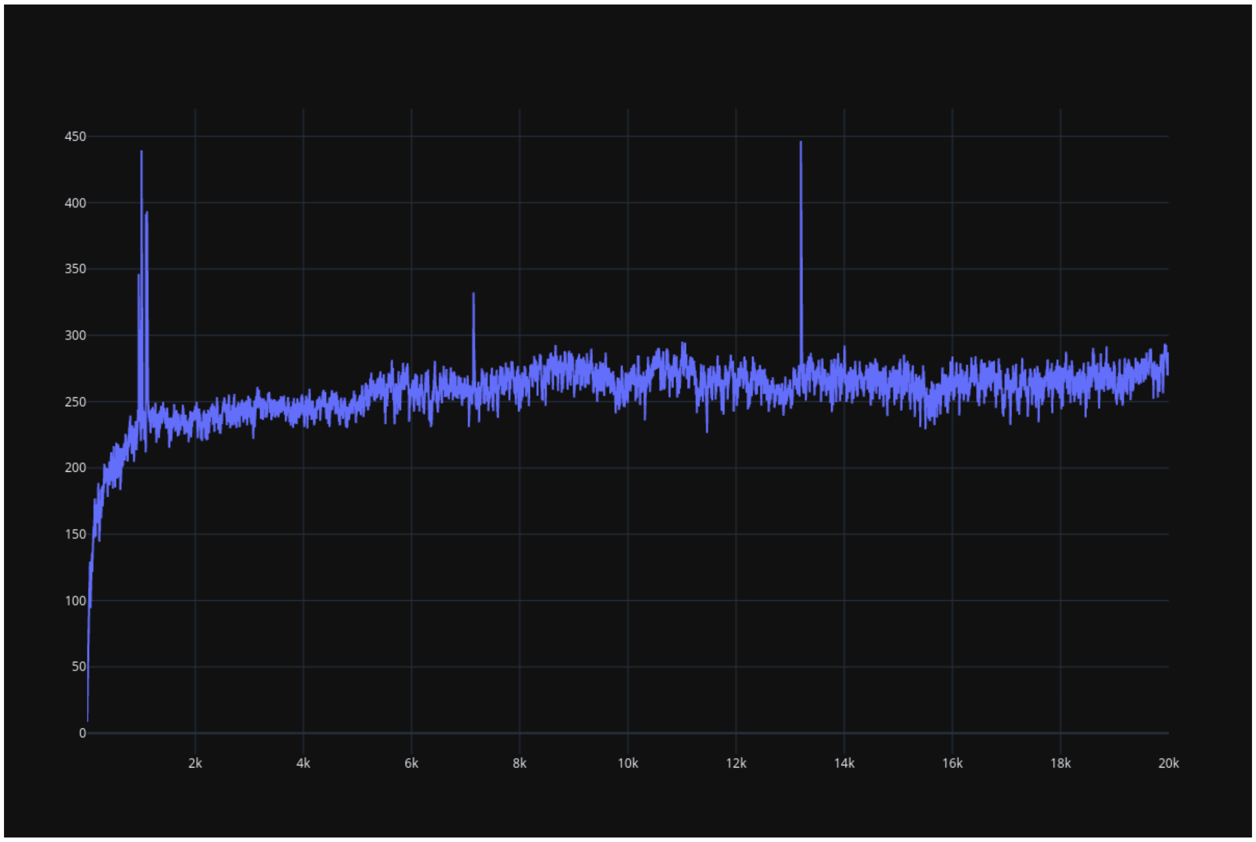


Fig. 3.14: PPO result (history_depth=1)

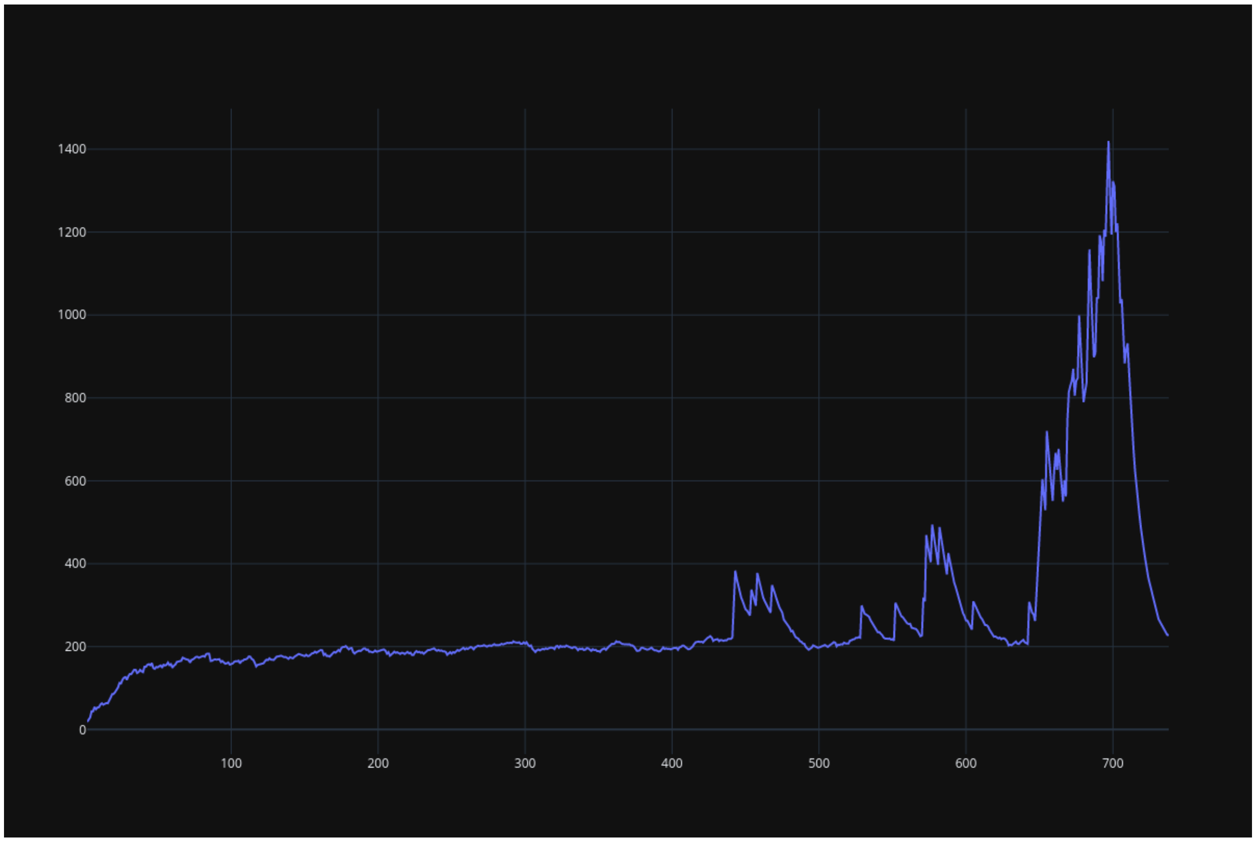


Fig. 3.15: PPO result (history_depth=4)

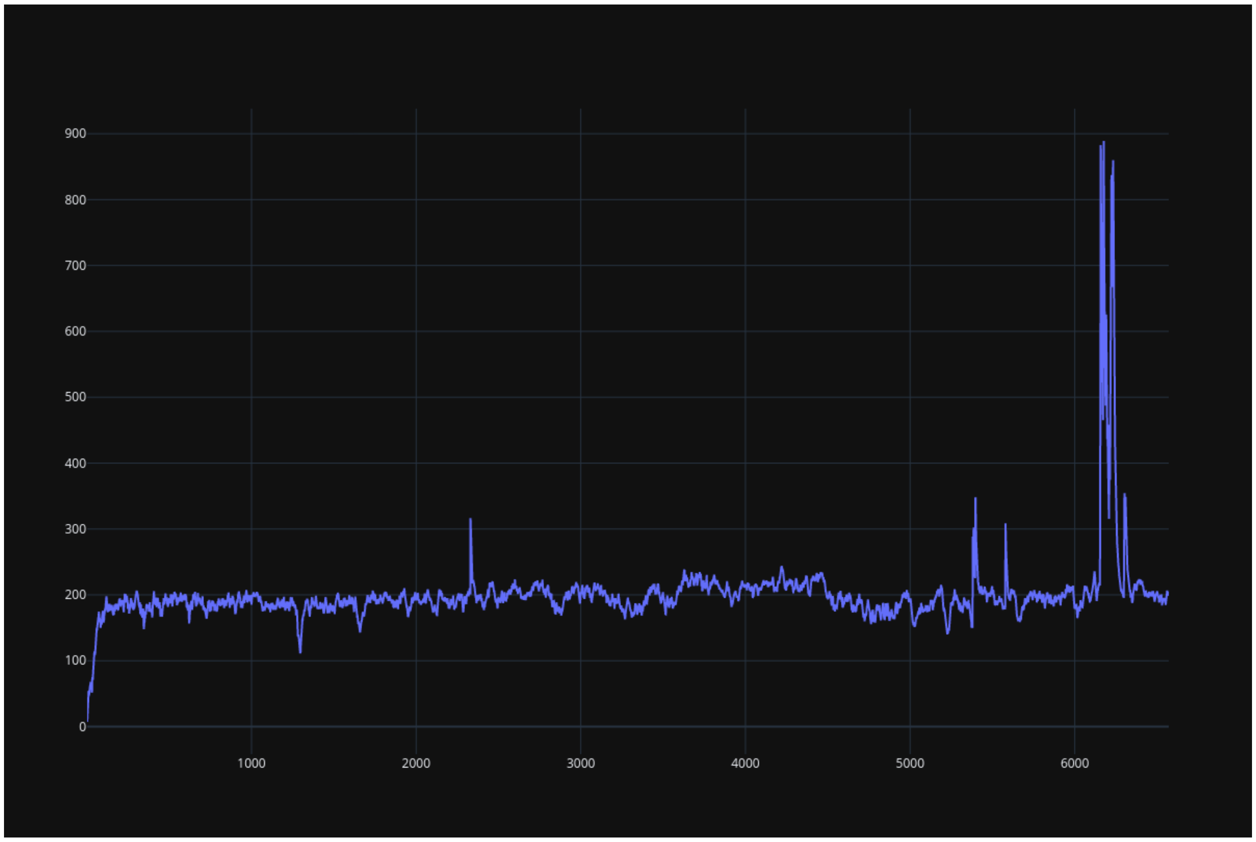


Fig. 3.16: RPPO result

3.2 Advance

3.2.1 Architecture overview

In this section, we will take a brief look at the internal organization of the Machin library, to better understand the functionality of every module.

Env

Currently `machin.env` has two sub modules: `machin.env.utils` and `machin.env.wrappers`.

The submodule `machin.env.utils` of the environment module provides some convenient utility functions you might will need in your own application, such as disabling the rendering window while keeping the rendered result in OpenAI gym.

The submodule `machin.env.wrappers` provides process-level parallel environment wrappers for different environments.

Framework

`machin.frame` is the most important core part of the Machin library, the framework module constitutes of:

1. `machin.frame.algorithms` : RL algorithm implementations.
2. `machin.frame.buffers` : Replay buffer implementations.
3. `machin.frame.helpers` : Utility to help you initialize a framework.
4. `machin.frame.noise` : Action space & parameter space noise generators.

Model

`machin.model` is a collection of popular network models you might will use in your own program, for example, ResNet.

Model module also contains the basis of all network modules: `NeuralNetworkModule`, this wrapper is built upon regular *torch.nn.Module*, and allows users to specify input/output sub module.

Parallel

`machin.parallel` is the second core part of the Machin library, the parallel module is a collection of refined implementations including:

1. `machin.parallel.thread` : Thread (With exception catching).
2. `machin.parallel.process` : Process (With remote exception catching).
3. `machin.parallel.queues` : Queues. (Used in pools).
4. *`machin.parallel.pool`* : Process pools (allow local functions, customize serialization policy), thread pools, pools with contexts, etc.
5. *`machin.parallel.assigner`* : Heuristic based model-device assignment.
6. *`machin.parallel.server`* : Implementations of different servers used in distributed algorithms such as *`A3C`*, *`DQNApex`*, `DDPGApex` and *`IMPALA`*.

7. `machin.parallel.distributed` : A naive implementation of a part of RFC #41546

Utils

`machin.utils` is a **messy hotchpotch** of various tools, it is very hard to categorize them, but they could be helpful sometimes, so we left them here:

1. `machin.utils.checker` : A checker implementation, using forward & backward hooks provided by pytorch to check the input/ouput, input gradient of models. Supports user defined checkers and tensorboard.
2. `machin.utils.conf` : Functions designed to load/save a json configuration file, as well as loading parametrs from commandline.
3. `machin.utils.helper_classes` : Various helper classes, such as `Timer`, `Counter`, etc.
4. `machin.utils.learning_rate` : Functions used in learning rate schedulers. Useful if you would like to have finer control over the learning rate.
5. `machin.utils.loading` : Logging utility module.
6. `machin.utils.media` : Media writing utility, mainly images and videos, useful if you would like to log rendered environments.
7. `machin.utils.prepare` : Functions used to create directories, loading models (take care of devices automatically), for preparing a training session.
8. `machin.utils.save_env` : A standard reinforcement training environment creator, will create unique directories by time for you.
9. `machin.utils.visualize` : Visualize your model, currently only contains some simple functions for gradient flow checking.
10. `machin.utils.tensorboard`: A simple tensorboard wrapper.

3.2.2 Algorithm APIs

Author: Muhan Li

Currently, Machin supports three major types of model-free RL algorithms:

1. Value based algorithms
2. Deterministic policy based algorithms
3. Stochastic policy based algorithms

Algorithms could be grouped into respective categories with the following graph:

We will use some basic symbols to simplify the description:

1. `...` means one or more dimensions, with non-zero sizes.
2. <> means optional results / arguments. `<...>` means any number of optional results / arguments.

Note: When an algorithm API returns one result, the result will not be wrapped in a tuple, when it returns multiple results, results will be wrapped in a tuple. This design is made to support:

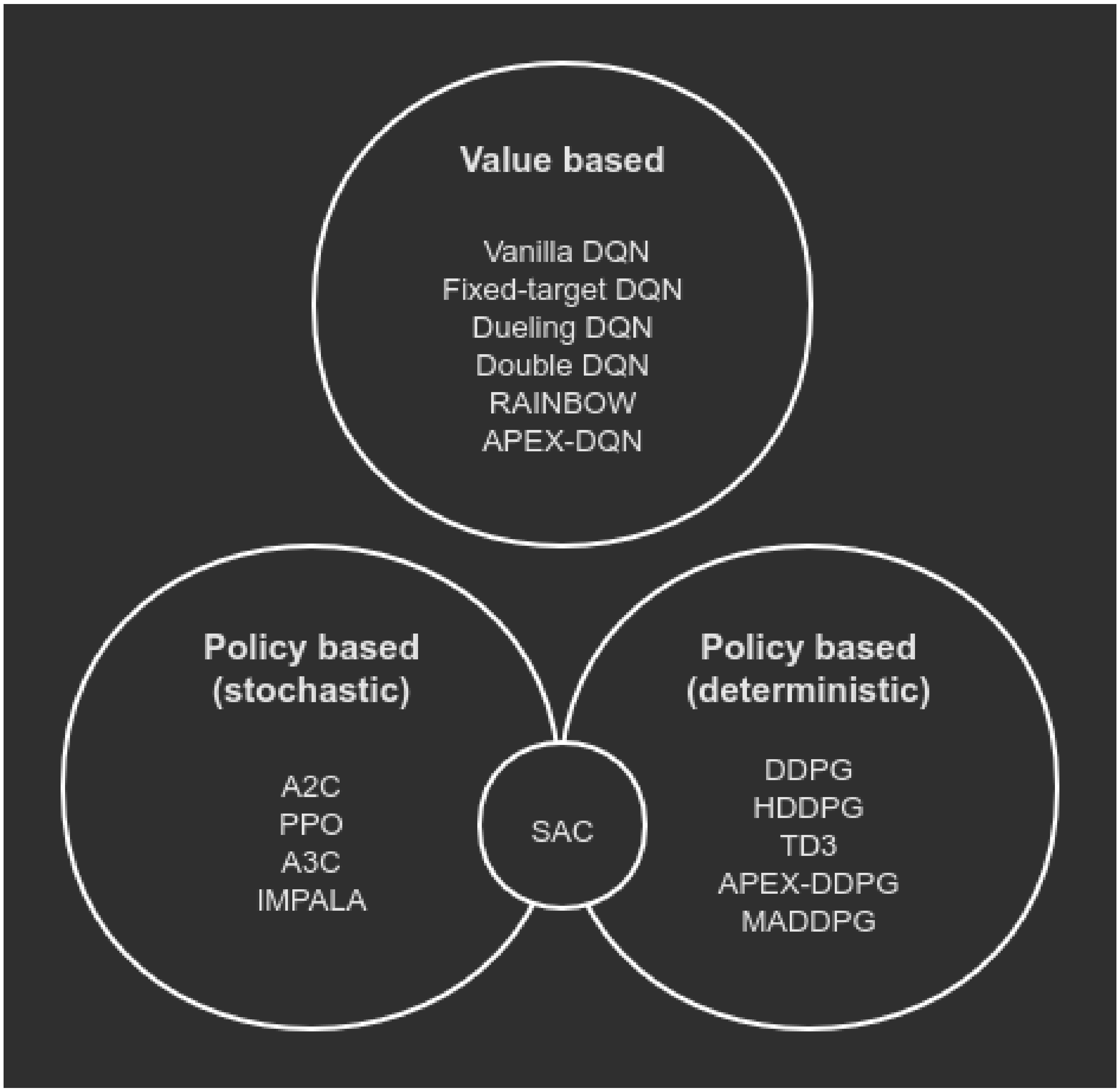


Fig. 3.17: Algorithm categories

```
# your Q network model only returns a Q value tensor
act = dqn.act({"state": some_state})

# your Q network model returns Q value tensor with some additional hidden states
act, h = dqn.act({"state": some_state})
```

Core APIs

All algorithms provide three core APIs:

1. Acting API, beginning with "act".
2. Storing API, beginning with "store".
3. Training API, with name "update"

Acting API

Users will invoke the "act*" api provided by the framework during sampling, to let their models produce an action with respect to their state input, "*" indicates additional extensions such as "_with_noise", "_discreet", etc. depending on the implementation and type of the RL framework.

Below is a list of supported acting APIs of different frameworks:

Algorithm class	Acting API	Input & output	Discreet/Contiguous	Note
DQN DQNPer DQNApex RAINBOW	act_discreet	Dict[str, State[batch_size, . . .]] -> Action[batch_size, 1], <. . . >	D	
	act_discreet_with_noise	Dict[str, State[batch_size, . . .]] -> Action[batch_size, 1], <. . . >	D	
DDPG DDPGPer HDDPG TD3	act	Dict[str, State[batch_size, . . .]] -> Action[batch_size, action_dim], <. . . >	C	
	act_with_noise	Dict[str, State[batch_size, . . .]] -> Action[batch_size, action_dim], <. . . >	C	
	act_discreet	Dict[str, State[batch_size, . . .]] -> Action[batch_size, 1], Prob[batch_size, action_num], <. . . >	D	
	act_discreet_with_noise	Dict[str, State[batch_size, . . .]] -> Action[batch_size, 1], Prob[batch_size, action_num], <. . . >	D	

Storing API

Algorithms generally encapsulate a replay buffer inside, the replay buffer is not necessarily a "real" replay buffer. For online algorithms such as A2C and PPO with no replaying mechanisms, the replay buffer is used as a place to put all of the samples, and is cleared after every training/update step:

```
# sample a batch
batch_size, (state, action, reward, next_state,
             terminal, target_value, advantage) = \
    self.replay_buffer.sample_batch(-1,
                                    sample_method="all",
                                    ...)

...
self.replay_buffer.clear()
```

Most frameworks supports storing a single transition step of a MDP process, or storing the whole MDP process at once:

```
some_framework.store_transition(transition: Union[Transition, Dict])
some_framework.store_episode(episode: List[Union[Transition, Dict]])
```

However, some frameworks may only support the latter one of these two APIs (Eg: IMPALA), due to the special sampling requirements of the algorithm.

Below is a list of supported storing APIs of different frameworks:

Algorithm class	Storing API	Note
DQN DQNPer DQNApex DDPG DDPGPer DDPGApex HDDPG TD3 SAC	store_transition/store_episode	
MADDPG	store_transition/store_episode	Requires you to store transitions/episodes of all agents at the same time.
RAINBOW	store_transition/store_episode	`store_transition` requires you to calculate the n-step value manually.
A2C PPO A3C	store_transition/store_episode	`store_transition` requires you to calculate the n-step value, and the generalized advantage estimation (GAE) manually.
IMPALA	store_episode	

Training API

All frameworks supports the `update` function, but the keyword arguments of the `update` function might be a little bit different. For example, DDPG allows you to choose update actor/critic/their targets, individually, while DQN only supports choose to update Q network/its target individually.

Moreover, the update function of offline algorithms such as DDPG and online algorithms such as A2C and PPO are different. Because A2C and PPO will not update on outdated samples, their `update` function contains an internal update loop, therefore you should not call them many times:

```
# DDPG update:
if episode > 100:
for i in range(step.get()):
```

(continues on next page)

(continued from previous page)

```
    ddpg.update()

# PPO update:
# update() already contains a loop
ppo.store_episode(tmp_observations)
ppo.update()
```

and their `update` will also clear the internal replay buffer every time. So you are recommended to **read the implementation** of your selected algorithm before using it somewhere.

Non-core APIs

All algorithms provide these non-core APIs:

1. Saving/Loading API, with name "save" and "load".
2. Learning Rate Scheduler API, with name "update_lr_scheduler".

Saving/Loading API

All frameworks provide this pair of APIs, for saving and loading models passed to the algorithm. Internally, the models passed to the algorithm framework will become a member of the framework instance, for example:

```
dqn = DQN(q_net, q_net_t, t.optim.Adam, nn.MSELoss(reduction='sum'))

# you may access q_net and q_net_t with:
print(dqn.qnet)
print(dqn.qnet_target)
```

You can print the `_is_restorable` attribute of the algorithm **class** to view models saved/loaded internally, and print the `_is_top` attribute of the algorithm **class** to view top level models, like Q network, actor network, critic network, etc.:

```
print(DQN._is_restorable)
# ["qnet_target"]
print(DQN._is_top)
# ["qnet", "qnet_target"]
```

Saving/Loading API requires you to provide a directory to save/load the models, an *optional* model name map to specify the mapping relation between "model <-> saved model name", and an *optional* version number indicating the version of save:

```
# Model dqn.qnet_target will be saved **as a whole** in "./qnt_1000.pt"
# **saved as whole** means saving like: torch.save(dqn.qnet_target, ...)
dqn.save("./", network_map={"qnet_target": "qnt"}, version=1000)

# If no name mapping is specified, the default "qnet_target" will be used
# as the saving name
dqn.save("./", version=1000)

# If no version is specified, the default saving version number is 0
dqn.save("./", network_map={"qnet_target": "qnt"})

# If no version number is specified, then the model with the largest version
```

(continues on next page)

(continued from previous page)

```
# number will be loaded
dqn.load("./", network_map={"qnet_target": "qnt"})

# Or specify a specific version to load
dqn.load("./", network_map={"qnet_target": "qnt"}, version=1000)

# An invalid version will cause the framework to find the latest available version
dqn.load("./", network_map={"qnet_target": "qnt"}, version=10000)

# If you have a file named "qnt.pt", which has no valid version number, it
# will be ignored.
```

You may move the saved model files to **a different machine with different devices**, there is no need to worry about different device mapping, the parameters of saved models will be loaded into your model(s) passed to the algorithm framework.

Some frameworks may need to save multiple models, for example, `DDPG` needs to save a target critic network and a target actor network, in this case, each model will **be saved to a separate file**, the loading function will try to find the maximum available version in the **valid version intersection** of all models:

```
# suppose there are these models in the target directory:
# actor_target_0.pt, actor_target_100.pt, actor_target_1000.pt
# critic_target_0.pt, critic_target_100.pt
# then version 100 will be loaded
ddpg.load("./")
```

Learning Rate Scheduler API

All frameworks have this API, for adjusting the learning rate scheduler passed to the framework:

```
q_net = QNet(c.observe_dim, c.action_num)
q_net_t = QNet(c.observe_dim, c.action_num)
lr_func = gen_learning_rate_func([(0, 1e-3), (200000, 3e-4)],
                                 logger=logger)
dqn = DQN(q_net, q_net_t,
          t.optim.Adam,
          nn.MSELoss(reduction='sum'),
          replay_device=c.device,
          replay_size=c.replay_size,
          lr_scheduler=LambdaLR,
          lr_scheduler_args=((lr_func,),))
```

You may invoke it like below, after the first update call:

```
dqn.update_lr_scheduler()
```

Algorithm specific APIs

Since algorithms are drastically different, it is hard to conform some of their features to the same style and design, therefore, they are exposed as-is if you would like to interface with these APIs, for using the critic network, evaluating an action, etc. Below is a list of these APIs supported by different frameworks:

Algorithm class	Algorithm specific APIs	Input & output	Note
DQNApex DDPGApex A3C IMPALA	set_sync	bool -> None	disable/enable auto local model syncing with remote server(s). **note**: syncing is performed in every act/criticize/... etc.
DQNApex DDPGApex A3C IMPALA	manual_sync	bool -> None	Manually update local models by pulling the newest parameters from remote server(s).

3.2.3 Algorithm model requirements

Author: Muhan Li

Machin relies on the correct model implementation to function correctly, different RL algorithms may need drastically dissimilar models. Therefore, in this section, we are going to outline the detailed requirements on models of different frameworks.

We will use some basic symbols to simplify the model signature:

1. `abc_0[*]` means a tensor with meaning "abc", and has index 0 in all argument tensors with the same meaning, "*" is a wildcard which accepts one or more non-zero dimensions, valid examples are:

   state_0[batch_size, 1]
   state_1[1, 2, 3, 4, 5]
   state_2[...]

2. `...` means one or more arguments (tensors/not tensors), or one or more dimensions, with non-zero sizes.
3. <> means optional results / arguments. `<...>` means any number of optional results / arguments.

Note: When an algorithm API returns one result, the result will not be wrapped in a tuple, when it returns multiple results, results will be wrapped in a tuple. This design is made to support:

```
# your Q network model only returns a Q value tensor
act = dqn.act({"state": some_state})
```

(continues on next page)

(continued from previous page)

```
# your Q network model returns Q value tensor with some additional hidden states
act, h = dqn.act({"state": some_state})
```

Note: the `forward` method signature **must conform to the following definitions exactly**, with no more or less arguments/keyword arguments.

Note: the requirements in this document does not apply to the conditions where: (1) you have made a custom implementation (2) you have inherited frameworks and customized their result adaptors like *DDPG.action_transform_function()*, etc.

DQN, DQNPer, DQNApex

For *DQN*, *DQNPer*, *DQNApex*, Machin expects a Q network:

```
QNet(state_0[batch_size, ...],
     ...,
     state_n[batch_size, ...])
-> q_value[batch_size, action_num], <...>
```

where `action_num` is the number of available discreet actions.

Example:

```
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, state, state2):
        state = t.cat([state, state2], dim=1)
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        return self.fc3(a)
```

Dueling DQN

An example of the dueling DQN:

```
class QNet(nn.Module):
    def __init__(self, state_dim, action_num):
        super(QNet, self).__init__()
        self.action_num = action_num
        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc_adv = nn.Linear(16, action_num)
        self.fc_val = nn.Linear(16, 1)

    def forward(self, some_state):
        a = t.relu(self.fc1(some_state))
        a = t.relu(self.fc2(a))
```

(continues on next page)

(continued from previous page)

```
        batch_size = a.shape[0]
        adv = self.fc_adv(a)
        val = self.fc_val(a).expand(batch_size, self.action_num)
        return val + adv - adv.mean(1, keepdim=True)
```

RAINBOW

For *RAINBOW*, Machin expects a distributional Q network:

```
DistQNet(state_0[batch_size, ...],
         ...,
         state_n[batch_size, ...])
-> q_value_dist[batch_size, action_num, atom_num], <...>
```

where:

1. `action_num` is the number of available discreet actions
2. `atom_num` is the number of q value distribution bins
3. `sum(q_value_dist[i, j, :]) == 1`

Example:

```
class QNet(nn.Module):
    def __init__(self, state_dim, action_num, atom_num=10):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num * atom_num)
        self.action_num = action_num
        self.atom_num = atom_num

    def forward(self, state, state2):
        state = t.cat([state, state2], dim=1)
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        return t.softmax(self.fc3(a)
                         .view(-1, self.action_num, self.atom_num),
                         dim=-1)
```

DDPG, DDPGPer, DDPGApex, HDDPG, TD3

For *DDPG*, *DDPGPer*, *DDPGApex*, *HDDPG*, *TD3*, Machin expects multiple actor and critic networks like:

```
Actor(state_0[batch_size, ...],
      ...,
      state_n[batch_size, ...])
-> action[batch_size, ...], <...>          # if contiguous
-> action[batch_size, action_num], <...>   # if discreet

Critic(state_0[batch_size, ...],
       ...,
       state_n[batch_size, ...],
```

(continues on next page)

(continued from previous page)

```
       action[batch_size, .../action_num])
-> q_value[batch_size, 1], <...>
```

where:

1. `action_num` is the number of available discreet actions
2. `sum(action[i, :]) == 1` if discreet.

Example:

```
class Actor(nn.Module):
def __init__(self, state_dim, action_dim, action_range):
    super(Actor, self).__init__()

    self.fc1 = nn.Linear(state_dim, 16)
    self.fc2 = nn.Linear(16, 16)
    self.fc3 = nn.Linear(16, action_dim)
    self.action_range = action_range

def forward(self, state):
    a = t.relu(self.fc1(state))
    a = t.relu(self.fc2(a))
    a = t.tanh(self.fc3(a)) * self.action_range
    return a


class ActorDiscrete(nn.Module):
    def __init__(self, state_dim, action_dim):
        # action_dim means action_num here
        super(ActorDiscrete, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_dim)

    def forward(self, state):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        a = t.softmax(self.fc3(a), dim=1)
        return a


class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(state_dim + action_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state, action):
        state_action = t.cat([state, action], 1)
        q = t.relu(self.fc1(state_action))
        q = t.relu(self.fc2(q))
        q = self.fc3(q)
        return q
```

A2C, PPO, A3C, IMPALA

For *A2C*, *PPO*, *A3C*, *IMPALA*, Machin expects multiple actor and critic networks like:

```
Actor(state_0[batch_size, ...],
      ...,
      state_n[batch_size, ...],
      action[batch_size, ...]=None)
-> action[batch_size, ...], <...>
   action_log_prob[batch_size, 1]
   distribution_entropy[batch_size, 1]

Critic(state_0[batch_size, ...],
       ...,
       state_n[batch_size, ...])
-> value[batch_size, 1], <...>
```

where:

1. `action` can be sampled from pytorch distributions using non-differentiable `sample()`.
2. `action_log_prob` is the log likelihood of the sampled action, must be differentiable.
3. `distribution_entropy` is the entropy value of reparameterized distribution, must be differentiable.
4. `Actor` must calculate the log probability of the input `action` if it is not `None`, and return the input action **as-is**.

Example:

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_num):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        probs = t.softmax(self.fc3(a), dim=1)
        dist = Categorical(probs=probs)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act.flatten())
        return act, act_log_prob, act_entropy

class ActorContiguous(nn.Module):
    def __init__(self, state_dim, action_dim, action_range):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.mu_head = nn.Linear(16, action_dim)
        self.sigma_head = nn.Linear(16, action_dim)
        self.action_range = action_range
```

(continues on next page)

(continued from previous page)

```
    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        mu = self.mu_head(a)
        sigma = softplus(self.sigma_head(a))
        dist = Normal(mu, sigma)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()

        # If your distribution is different from "Normal" then you may either:
        # 1. deduce the remapping function for your distribution and clamping
        #    function such as tanh
        # 2. clamp you action, but please take care:
        #    1. do not clamp actions before calculating their log probability,
        #       because the log probability of clamped actions might will be
        #       extremely small, and will cause nan
        #    2. do not clamp actions after sampling and before storing them in
        #       the replay buffer, because during update, log probability will
        #       be re-evaluated they might also be extremely small, and network
        #       will "nan". (might happen in PPO, not in SAC because there is
        #       no re-evaluation)
        # Only clamp actions sent to the environment, this is equivalent to
        # change the action reward distribution, will not cause "nan", but
        # this makes your training environment further differ from you real
        # environment.

        # the suggested way to confine your actions within a valid range
        # is not clamping, but remapping the distribution
        # from the SAC essay:   https://arxiv.org/abs/1801.01290
        act_log_prob = dist.log_prob(act)
        act_tanh = t.tanh(act)
        act = act_tanh * self.action_range

        # the distribution remapping process used in the original essay.
        act_log_prob -= t.log(self.action_range *
                              (1 - act_tanh.pow(2)) +
                              1e-6)
        act_log_prob = act_log_prob.sum(1, keepdim=True)

        return act, act_log_prob, act_entropy

class Critic(nn.Module):
    def __init__(self, state_dim):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state):
        v = t.relu(self.fc1(state))
        v = t.relu(self.fc2(v))
        v = self.fc3(v)
        return v
```

SAC

For *SAC*, Machin expects an actor similar to the actors in stochastic policy gradient methods such as *A2C*, and multiple critics similar to critics used in *DDPG*:

```
Actor(state_0[batch_size, ...],
      ...,
      state_n[batch_size, ...],
      action[batch_size, ...]=None)
-> action[batch_size, ...]
   action_log_prob[batch_size, 1]
   distribution_entropy[batch_size, 1],
   <...>

Critic(state_0[batch_size, ...],
       ...,
       state_n[batch_size, ...],
       action[batch_size, .../action_num])
-> q_value[batch_size, 1], <...>
```

where:

1. `action` can only be sampled from pytorch distributions using **differentiable** `rsample()`.
2. `action_log_prob` is the log likelihood of the sampled action, must be differentiable.
3. `distribution_entropy` is the entropy value of reparameterized distribution, must be differentiable.
4. `Actor` must calculate the log probability of the input `action` if it is not `None`, and return the input action **as-is**.

Example:

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_num):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, action_num)

    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        probs = t.softmax(self.fc3(a), dim=1)
        dist = Categorical(probs=probs)
        act = (action
               if action is not None
               else dist.sample())
        act_entropy = dist.entropy()
        act_log_prob = dist.log_prob(act.flatten())
        return act, act_log_prob, act_entropy

class ActorContiguous(nn.Module):
    def __init__(self, state_dim, action_dim, action_range):
        super(Actor, self).__init__()

        self.fc1 = nn.Linear(state_dim, 16)
        self.fc2 = nn.Linear(16, 16)
```

(continues on next page)

(continued from previous page)

```
        self.mu_head = nn.Linear(16, action_dim)
        self.sigma_head = nn.Linear(16, action_dim)
        self.action_range = action_range

    def forward(self, state, action=None):
        a = t.relu(self.fc1(state))
        a = t.relu(self.fc2(a))
        mu = self.mu_head(a)
        sigma = softplus(self.sigma_head(a))
        dist = Normal(mu, sigma)
        act = (action
               if action is not None
               else dist.rsample())
        act_entropy = dist.entropy()

        # the suggested way to confine your actions within a valid range
        # is not clamping, but remapping the distribution
        act_log_prob = dist.log_prob(act)
        act_tanh = t.tanh(act)
        act = act_tanh * self.action_range

        # the distribution remapping process used in the original essay.
        act_log_prob -= t.log(self.action_range *
                              (1 - act_tanh.pow(2)) +
                              1e-6)
        act_log_prob = act_log_prob.sum(1, keepdim=True)

        return act, act_log_prob, act_entropy

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        self.fc1 = nn.Linear(state_dim + action_dim, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, state, action):
        state_action = t.cat([state, action], 1)
        q = t.relu(self.fc1(state_action))
        q = t.relu(self.fc2(q))
        q = self.fc3(q)
        return q
```

CHAPTER

FOUR

API

4.1 API

4.1.1 machin.env

utils

`machin.env.utils` provides utilities to deal with various environments.

`machin.env.utils.openai_gym.`**`disable_view_window`**`()`

wrappers

`machin.env.wrappers` provides parallel execution wrappers for various environments.

`class` `machin.env.wrappers.base.`**`ParallelWrapperBase`**`(*_, **__)`

Bases: `abc.ABC`

Note: Parallel wrapper is designed to wrap the same kind of environments, they may have different parameters, but must have the same action and observation space.

`abstract reset``(`*idx=None*`)`

Reset all environments if id is `None`, otherwise reset the specific environment(s) with given index(es).

Parameters **`idx`** (*`Union[int, List[int], None]`*) – Environment index(es) to be reset.

Returns Initial observation of all environments. Format is unspecified.

Return type Any

`abstract step``(`*action*, *idx=None*`)`

Let specified environment(s) run one time step. specified environments must be active and have not reached terminal states before.

Parameters

- **`action`** – actions to take.
- **`idx`** (*`Union[int, List[int], None]`*) – Environment index(es) to be run.

Returns New states of environments.

Return type Any

abstract seed(*seed=None*)

Set seed(s) for all environment(s).

Parameters **seed** (*Union[int, List[int], None]*) – A single integer seed for all environments, or a list of integers for each environment, or None for default seed.

Returns New seed of each environment.

Return type List[int]

abstract render(**args*, ***kwargs*)

Render all environments.

Return type Any

abstract close()

Close all environments.

Return type Any

abstract active()

Returns Indexes of active environments.

Return type List[int]

abstract size()

Returns Number of environments.

Return type int

abstract property action_space

Returns: Action space descriptor.

abstract property observation_space

Returns: Observation space descriptor.

exception machin.env.wrappers.openai_gym.**GymTerminationError**

Bases: Exception

class machin.env.wrappers.openai_gym.**ParallelWrapperDummy**(*env_creators*)

Bases: *machin.env.wrappers.base.ParallelWrapperBase*

Dummy parallel wrapper for gym environments, implemented using for-loop.

For debug purpose only.

Parameters **env_creators** (*List[Callable[[int], gym.core.Env]]*) – List of gym environment creators, used to create environments, accepts a index as your environment id.

reset(*idx=None*)

Returns A list of gym states.

Parameters **List[int]] idx** (*Union[int,*) –

Return type List[object]

step(*action*, *idx=None*)

Let specified environment(s) run one time step. Specified environments must be active and have not reached terminal states before.

Parameters

- **`action`** (*`Union[numpy.ndarray, List[Any]]`*) – Actions sent to each specified environment, the size of the first dimension must match the number of selected environments.
- **`idx`** (*`Union[int, List[int]]`*) – Indexes of selected environments, default is all.

Returns Observation, reward, terminal, and diagnostic info.

Return type Tuple[List[object], List[float], List[bool], List[dict]]

`seed`(*seed=None*)

Set seeds for all environments.

Parameters **`seed`** (*`Union[int, List[int]]`*) – If seed is `int`, the same seed will be used for all environments. If seed is `List[int]`, it must have the same size as the number of all environments. If seed is `None`, all environments will use the default seed.

Returns Actual used seed returned by all environments.

Return type List[int]

`render`(*idx=None*, **_*, ***__*)

Render all/specified environments.

Parameters **`idx`** (*`Union[int, List[int]]`*) – Indexes of selected environments, default is all.

Returns A list or rendered frames, of type `np.ndarray` and size (H, W, 3).

Return type List[numpy.ndarray]

`close`()

Close all environments.

Return type None

`active`()

Returns: Indexes of current active environments.

Return type List[int]

`size`()

Returns: Number of environments.

Return type int

`property action_space`

Returns: Action space descriptor.

`property observation_space`

Returns: Observation space descriptor.

`class` `machin.env.wrappers.openai_gym.`**`ParallelWrapperSubProc`**(*env_creators*)

Bases: *`machin.env.wrappers.base.ParallelWrapperBase`*

Parallel wrapper based on sub processes.

Parameters **`env_creators`** (*`List[Callable[[int], gym.core.Env]]`*) – List of gym environment creators, used to create environments on sub process workers, accepts a index as your environment id.

Return type None

`reset`(*idx=None*)

Returns A list of gym states.

Parameters **List[int]] idx** (*Union[int,*) –

Return type List[object]

step (*action*, *idx=None*)

Let specified environment(s) run one time step. Specified environments must be active and have not reached terminal states before.

Parameters

- **action** (*Union[numpy.ndarray, List[Any]]*) – Actions sent to each specified environment, the size of the first dimension must match the number of selected environments.
- **idx** (*Union[int, List[int]]*) – Indexes of selected environments, default is all.

Returns Observation, reward, terminal, and diagnostic info.

Return type Tuple[List[object], List[float], List[bool], List[dict]]

seed (*seed=None*)

Set seeds for all environments.

Parameters **seed** (*Union[int, List[int]]*) – If seed is `int`, the same seed will be used for all environments. If seed is `List[int]`, it must have the same size as the number of all environments. If seed is `None`, all environments will use the default seed.

Returns Actual used seed returned by all environments.

Return type List[int]

render (*idx=None*, **args*, ***kwargs*)

Render all/specified environments.

Parameters **idx** (*Union[int, List[int]]*) – Indexes of selected environments, default is all.

Returns A list or rendered frames, of type `np.ndarray` and size (H, W, 3).

Return type List[numpy.ndarray]

close ()

Close all environments, including the wrapper.

Return type None

active ()

Returns: Indexes of current active environments.

Return type List[int]

size ()

Returns: Number of environments.

Return type int

property action_space

Returns: Action space descriptor.

property observation_space

Returns: Observation space descriptor.

4.1.2 machin.frame

algorithms

Base

class machin.frame.algorithms.base.**TorchFramework**

Bases: `object`

Base framework for all algorithms

enable_multiprocessing()

Enable multiprocessing for all modules.

classmethod get_restorable()

Get restorable modules.

load(*model_dir*, *network_map=None*, *version=- 1*)

Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** (*str*) – Save directory.
- **network_map** (*Dict[str, str]*) – Key is module name, value is saved name.
- **version** (*int*) – Version number of the save to be loaded.

save(*model_dir*, *network_map=None*, *version=0*)

Save models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** (*str*) – Save directory.
- **network_map** (*Dict[str, str]*) – Key is module name, value is saved name.
- **version** (*int*) – Version number of the new save.

visualize_model(*final_tensor*, *name*, *directory*)

Parameters

- **final_tensor** (*torch.Tensor*) –
- **name** (*str*) –
- **directory** (*str*) –

DDPG

class machin.frame.algorithms.ddpg.**DDPG**(*actor*, *actor_target*, *critic*, *critic_target*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.001*, *actor_learning_rate=0.0005*, *critic_learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: *machin.frame.algorithms.base.TorchFramework*

DDPG framework.

Note: Your optimizer will be called as:

```
optimizer(network.parameters(), learning_rate)
```

Your lr_scheduler will be called as:

```
lr_scheduler(
    optimizer,
    *lr_scheduler_args[0],
    **lr_scheduler_kwargs[0],
)
```

Your criterion will be called as:

```
criterion(
    target_value.view(batch_size, 1),
    predicted_value.view(batch_size, 1)
)
```

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **actor_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **critic_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.

- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **gradient_max** (*float*) –

act (*state*, *use_target=False*, ***__*)
Use actor network to produce an action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether use the target network.

Returns Any thing returned by your actor network.

act_discrete (*state*, *use_target=False*, ***__*)
Use actor network to produce a discrete action for the current state.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Action of shape `[batch_size, 1]`. Action probability tensor of shape `[batch_size, action_num]`, produced by your actor. Any other things returned by your Q network. if they exist.

act_discrete_with_noise (*state*, *use_target=False*, ***__*)
Use actor network to produce a noisy discrete action for the current state.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **`state`** (*`Dict[str, Any]`*) – Current state.
- **`use_target`** (*`bool`*) – Whether to use the target network.

Returns Noisy action of shape `[batch_size, 1]`. Action probability tensor of shape `[batch_size, action_num]`. Any other things returned by your Q network. if they exist.

`act_with_noise`(*state*, *noise_param=0.0, 1.0*, *ratio=1.0*, *mode='uniform'*, *use_target=False*, ***__*)
Use actor network to produce a noisy action for the current state.

See also:

`machin.frame.noise.action_space_noise`

Parameters

- **`state`** (*`Dict[str, Any]`*) – Current state.
- **`noise_param`** (*`Any`*) – Noise params.
- **`ratio`** (*`float`*) – Noise ratio.
- **`mode`** (*`str`*) – Noise mode. Supported are: `"uniform"`, `"normal"`, `"clipped_normal"`, `"ou"`
- **`use_target`** (*`bool`*) – Whether use the target network.

Returns Noisy action of shape `[batch_size, action_dim]`. Any other things returned by your actor network. if they exist.

`static action_transform_function`(*raw_output_action*, **_*)
The action transform function is used to transform the output of actor to the input of critic. Action transform function must accept:

1. Raw action from the actor model.
2. Concatenated *`Transition.next_state`*.
3. Any other concatenated lists of custom keys from *`Transition`*.

and returns:

1. A dictionary with the same form as *`Transition.action`*

Parameters **`raw_output_action`** (*`Any`*) – Raw action from the actor model.

`load`(*model_dir*, *network_map=None*, *version=- 1*)
Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **`model_dir`** (*`str`*) – Save directory.
- **`network_map`** (*`Dict[str, str]`*) – Key is module name, value is saved name.
- **`version`** (*`int`*) – Version number of the save to be loaded.

`static reward_function`(*reward*, *discount*, *next_value*, *terminal*, *_*)

`store_episode`(*episode*)

Add a full episode of transition samples to the replay buffer.

Parameters Dict]] episode (*`List[Union[`*`machin.frame.transition.Transition`*`,`*) –

`store_transition`(*transition*)

Add a transition sample to the replay buffer.

Parameters Dict] transition (*`Union[`*`machin.frame.transition.Transition`*`,`*) –

`update`(*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **`update_value`** – Whether to update the Q network.
- **`update_policy`** – Whether to update the actor network.
- **`update_target`** – Whether to update targets.
- **`concatenate_samples`** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

`update_lr_scheduler`()

Update learning rate schedulers.

Hysterical DDPG

class `machin.frame.algorithms.hddpg.`**`HDDPG`**(*actor*, *actor_target*, *critic*, *critic_target*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.005*, *actor_learning_rate=0.0005*, *critic_learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *q_increase_rate=1.0*, *q_decrease_rate=1.0*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: *`machin.frame.algorithms.ddpg.DDPG`*

HDDPG framework.

See also:

`DDPG`

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **actor_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **critic_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **gradient_max** (*float*) –
- **q_increase_rate** (*float*) –
- **q_decrease_rate** (*float*) –

update(*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.

- **`update_policy`** – Whether to update the actor network.
- **`update_target`** – Whether to update targets.
- **`concatenate_samples`** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

DDPG with prioritized replay

class `machin.frame.algorithms.ddpg_per.`**`DDPGPer`**(*actor, actor_target, critic, critic_target, optimizer, criterion, *_, lr_scheduler=None, lr_scheduler_args=None, lr_scheduler_kwargs=None, batch_size=100, update_rate=0.005, actor_learning_rate=0.0005, critic_learning_rate=0.001, discount=0.99, gradient_max=inf, replay_size=500000, replay_device='cpu', replay_buffer=None, visualize=False, visualize_dir='', **__*)

Bases: *`machin.frame.algorithms.ddpg.DDPG`*

DDPG with prioritized experience replay.

Warning: Your criterion must return a tensor of shape `[batch_size,1]` when given two tensors of shape `[batch_size,1]`, since we need to multiply the loss with importance sampling weight element-wise.

If you are using loss modules given by pytorch. It is always safe to use them without any modification.

Note: Your optimizer will be called as:

```
optimizer(network.parameters(), learning_rate)
```

Your lr_scheduler will be called as:

```
lr_scheduler(
    optimizer,
    *lr_scheduler_args[0],
    **lr_scheduler_kwargs[0],
)
```

Your criterion will be called as:

```
criterion(
    target_value.view(batch_size, 1),
    predicted_value.view(batch_size, 1)
)
```

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **actor_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **critic_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** – Criterion used to evaluate the value loss.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **gradient_max** (*float*) –

update(*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.
- **update_target** – Whether to update targets.

- **concatenate_samples** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

TD3

class machin.frame.algorithms.td3.**TD3**(*actor*, *actor_target*, *critic*, *critic_target*, *critic2*, *critic2_target*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.005*, *actor_learning_rate=0.0005*, *critic_learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: `machin.frame.algorithms.ddpg.DDPG`

TD3 framework. Which adds a additional pair of critic and target critic network to DDPG.

See also:

`DDPG`

Parameters

- **actor** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – Actor network module.
- **actor_target** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – Target actor network module.
- **critic** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – Critic network module.
- **critic_target** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – Target critic network module.
- **critic2** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – The second critic network module.
- **critic2_target** (`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`) – The second target critic network module.
- **optimizer** (`Callable`) – Optimizer used to optimize `actor`, `critic`,
- **criterion** (`Callable`) – Criterion used to evaluate the value loss.
- **lr_scheduler** (`Callable`) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (`Tuple[Tuple, Tuple, Tuple]`) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (`Tuple[Dict, Dict, Dict]`) – Keyword arguments of the learning rate scheduler.
- **batch_size** (`int`) – Batch size used during training.

- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **gradient_max** (*float*) –

load (*model_dir*, *network_map=None*, *version=- 1*)

Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** (*str*) – Save directory.
- **network_map** (*Dict[str, str]*) – Key is module name, value is saved name.
- **version** (*int*) – Version number of the save to be loaded.

static policy_noise_function (*actions*, **_*)

update (*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.
- **update_target** – Whether to update targets.
- **concatenate_samples** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

update_lr_scheduler ()

Update learning rate schedulers.

DQN, Fixed-Target DQN, Dueling DQN, Double DQN

class machin.frame.algorithms.dqn.**DQN**(*qnet, qnet_target, optimizer, criterion, *_, lr_scheduler=None, lr_scheduler_args=None, lr_scheduler_kwargs=None, batch_size=100, update_rate=0.005, learning_rate=0.001, discount=0.99, gradient_max=inf, replay_size=500000, replay_device='cpu', replay_buffer=None, mode='double', visualize=False, visualize_dir='', **__*)

Bases: `machin.frame.algorithms.base.TorchFramework`

DQN framework.

Note: DQN is only available for discrete environments.

Note: Dueling DQN is a network structure rather than a framework, so it could be applied to all three modes.

If `mode = "vanilla"`, implements the simplest online DQN, with replay buffer.

If `mode = "fixed_target"`, implements DQN with a target network, and replay buffer. Described in this essay.

If `mode = "double"`, implements Double DQN described in this essay.

Note: Vanilla DQN only needs one network, so internally, `qnet` is assigned to `qnet_target`.

Note: In order to implement dueling DQN, you should create two dense output layers.

In your q network:

```
self.fc_adv = nn.Linear(in_features=...,
                        out_features=num_actions)
self.fc_val = nn.Linear(in_features=...,
                        out_features=1)
```

Then in your `forward()` method, you should implement output as:

```
adv = self.fc_adv(some_input)
val = self.fc_val(some_input).expand(self.batch_sze,
                                     self.num_actions)
return val + adv - adv.mean(1, keepdim=True)
```

Note: Your optimizer will be called as:

```
optimizer(network.parameters(), learning_rate)
```

Your lr_scheduler will be called as:

```
lr_scheduler(
    optimizer,
    *lr_scheduler_args[0],
    **lr_scheduler_kwargs[0],
)
```

Your criterion will be called as:

```
criterion(
    target_value.view(batch_size, 1),
    predicted_value.view(batch_size, 1)
)
```

Parameters

- **qnet** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Q network module.
- **qnet_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target Q network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `qnet`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **learning_rate** (*float*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **mode** (*str*) – one of `"vanilla", "fixed_target", "double"`.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **gradient_max** (*float*) –
- **visualize_dir** (*str*) –

act_discrete(*state*, *use_target=False*, ***__*)
Use Q network to produce a discrete action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

act_discrete_with_noise(*state*, *use_target=False*, ***__*)
Use Q network to produce a noisy discrete action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Noisy action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

static action_get_function(*sampled_actions*)
This function is used to get action numbers (int tensor indicating which discrete actions are used) from the sampled action dictionary.

load(*model_dir*, *network_map=None*, *version=- 1*)
Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** – Save directory.
- **network_map** – Key is module name, value is saved name.
- **version** – Version number of the save to be loaded.

static reward_function(*reward*, *discount*, *next_value*, *terminal*, *_*)

store_episode(*episode*)
Add a full episode of transition samples to the replay buffer.

Parameters Dict]] episode (*List[Union[*machin.frame.transition.Transition*,)* –

store_transition(*transition*)
Add a transition sample to the replay buffer.

Parameters Dict] transition (*Union[*machin.frame.transition.Transition*,)* –

update(*update_value=True*, *update_target=True*, *concatenate_samples=True*, ***__*)
Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether update the Q network.

- **update_target** – Whether update targets.
- **concatenate_samples** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

update_lr_scheduler()
Update learning rate schedulers.

DQN with prioritized replay

class machin.frame.algorithms.dqn_per.**DQNPer**(*qnet*, *qnet_target*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.005*, *learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: *machin.frame.algorithms.dqn.DQN*

DQN with prioritized replay. It is based on Double DQN.

Warning: Your criterion must return a tensor of shape `[batch_size,1]` when given two tensors of shape `[batch_size,1]`, since we need to multiply the loss with importance sampling weight element-wise.

If you are using loss modules given by pytorch. It is always safe to use them without any modification.

Note: DQN is only available for discrete environments.

Note: Dueling DQN is a network structure rather than a framework, so it could be applied to all three modes.

If `mode = "vanilla"`, implements the simplest online DQN, with replay buffer.

If `mode = "fixed_target"`, implements DQN with a target network, and replay buffer. Described in this essay.

If `mode = "double"`, implements Double DQN described in this essay.

Note: Vanilla DQN only needs one network, so internally, `qnet` is assigned to `qnet_target`.

Note: In order to implement dueling DQN, you should create two dense output layers.

In your q network:

```
self.fc_adv = nn.Linear(in_features=...,
                        out_features=num_actions)
self.fc_val = nn.Linear(in_features=...,
                        out_features=1)
```

Then in your `forward()` method, you should implement output as:

```
adv = self.fc_adv(some_input)
val = self.fc_val(some_input).expand(self.batch_sze,
                                     self.num_actions)
return val + adv - adv.mean(1, keepdim=True)
```

Note: Your optimizer will be called as:

```
optimizer(network.parameters(), learning_rate)
```

Your lr_scheduler will be called as:

```
lr_scheduler(
    optimizer,
    *lr_scheduler_args[0],
    **lr_scheduler_kwargs[0],
)
```

Your criterion will be called as:

```
criterion(
    target_value.view(batch_size, 1),
    predicted_value.view(batch_size, 1)
)
```

Parameters

- **`qnet`** (*`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`*) – Q network module.
- **`qnet_target`** (*`Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]`*) – Target Q network module.
- **`optimizer`** (*`Callable`*) – Optimizer used to optimize `qnet`.
- **`criterion`** (*`Callable`*) – Criterion used to evaluate the value loss.
- **`learning_rate`** (*`float`*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **`lr_scheduler`** (*`Callable`*) – Learning rate scheduler of `optimizer`.
- **`lr_scheduler_args`** (*`Tuple[Tuple]`*) – Arguments of the learning rate scheduler.
- **`lr_scheduler_kwargs`** (*`Tuple[Dict]`*) – Keyword arguments of the learning rate scheduler.
- **`batch_size`** (*`int`*) – Batch size used during training.
- **`update_rate`** (*`float`*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **`discount`** (*`float`*) – γ used in the bellman function.
- **`replay_size`** (*`int`*) – Replay buffer size. Not compatible with `replay_buffer`.

- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **mode** – one of `"vanilla", "fixed_target", "double"`.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **gradient_max** (*float*) –
- **visualize_dir** (*str*) –

update (*update_value=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether update the Q network.
- **update_target** – Whether update targets.
- **concatenate_samples** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

RAINBOW

class machin.frame.algorithms.rainbow.**RAINBOW**(*qnet*, *qnet_target*, *optimizer*, *value_min*, *value_max*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.001*, *learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *reward_future_steps=3*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: *machin.frame.algorithms.dqn.DQN*

RAINBOW DQN framework.

RAINBOW framework is described in this essay.

Note: In the RAINBOW framework, the output shape of your q network must be `[batch_size, action_num, atom_num]` when given a state of shape `[batch_size, action_dim]`. And the last dimension **must be soft-maxed**. Atom number is the number of segments of your q value domain.

See also:

DQN

Parameters

- **qnet** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Q network module.

- **qnet_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target Q network module.
- **optimizer** – Optimizer used to optimize `actor` and `critic`.
- **value_min** – Minimum of value domain.
- **value_max** – Maximum of value domain.
- **learning_rate** (*float*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **discount** (*float*) – γ used in the bellman function.
- **reward_future_steps** (*int*) – Number of future steps to be considered when the framework calculates value from reward.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **mode** – one of `"vanilla", "fixed_target", "double"`.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **gradient_max** (*float*) –
- **visualize_dir** (*str*) –

act_discrete (*state*, *use_target=False*, ***__*)

Use Q network to produce a discrete action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

act_discrete_with_noise (*state*, *use_target=False*, ***__*)

Use Q network to produce a noisy discrete action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.

- **`use_target`** (*`bool`*) – Whether to use the target network.

Returns Noisy action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

`store_episode`(*episode*)
Add a full episode of transition samples to the replay buffer.

"value" is automatically calculated.

Parameters **`Dict]] episode`** (*`List[Union[`*`machin.frame.transition.Transition`*`,`*) –

`store_transition`(*transition*)
Add a transition sample to the replay buffer.

Not suggested, since you will have to calculate "value" by yourself.

Parameters **`Dict] transition`** (*`Union[`*`machin.frame.transition.Transition`*`,`*) –

`update`(*update_value=True*, *update_target=True*, *concatenate_samples=True*, ***__*)
Update network weights by sampling from replay buffer.

Parameters

- **`update_value`** – Whether update the Q network.
- **`update_target`** – Whether update targets.
- **`concatenate_samples`** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

A2C

`class` `machin.frame.algorithms.a2c.`**`A2C`**(*actor*, *critic*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *actor_update_times=5*, *critic_update_times=10*, *actor_learning_rate=0.001*, *critic_learning_rate=0.001*, *entropy_weight=None*, *value_weight=0.5*, *gradient_max=inf*, *gae_lambda=1.0*, *discount=0.99*, *normalize_advantage=True*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)
Bases: *`machin.frame.algorithms.base.TorchFramework`*

A2C framework.

Important: when given a state, and an optional, action actor must at least return two values:

1. Action

For **contiguous environments**, action must be of shape `[batch_size, action_dim]` and *clamped by action space*. For **discrete environments**, action could be of shape `[batch_size, action_dim]` if it is a one hot vector, or `[batch_size, 1]` if it is a categorically encoded integer.

2. Log likelihood of action (action probability)

For either type of environment, log likelihood is of shape [batch_size, 1].

Action probability must be differentiable, Gradient of actor is calculated from the gradient of action probability.

The third entropy value is optional:

3. Entropy of action distribution

Entropy is usually calculated using dist.entropy(), its shape is [batch_size, 1]. You must specify entropy_weight to make it effective.

Hint: For contiguous environments, action's are not directly output by your actor, otherwise it would be rather inconvenient to calculate the log probability of action. Instead, your actor network should output parameters for a certain distribution (eg: Normal) and then draw action from it.

For discrete environments, Categorical is sufficient, since differentiable rsample() is not needed.

This trick is also known as **reparameterization**.

Hint: Actions are from samples during training in the actor critic family (A2C, A3C, PPO, TRPO, IMPALA).

When your actor model is given a batch of actions and states, it must evaluate the states, and return the log likelihood of the given actions instead of re-sampling actions.

An example of your actor in contiguous environments:

```python
class ActorNet(nn.Module):
    def __init__(self):
        super(ActorNet, self).__init__()
        self.fc = nn.Linear(3, 100)
        self.mu_head = nn.Linear(100, 1)
        self.sigma_head = nn.Linear(100, 1)

    def forward(self, state, action=None):
        x = t.relu(self.fc(state))
        mu = 2.0 * t.tanh(self.mu_head(x))
        sigma = F.softplus(self.sigma_head(x))
        dist = Normal(mu, sigma)
        action = (action
                  if action is not None
                  else dist.sample())
        action_entropy = dist.entropy()
        action = action.clamp(-2.0, 2.0)
        action_log_prob = dist.log_prob(action)
        return action, action_log_prob, action_entropy
```

Hint: Entropy weight is usually negative, to increase exploration.

Value weight is usually 0.5. So critic network converges less slowly than the actor network and learns more conditions.

Update equation is equivalent to:

$Loss = w_e * Entropy + w_v * Loss_v + w_a * Loss_a$ $Loss_a = -log\_likelihood * advantage$ $Loss_v = criterion(target\_bellman\_value - V(s))$

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **actor_update_times** (*int*) – Times to update actor in `update()`.
- **critic_update_times** (*int*) – Times to update critic in `update()`.
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **entropy_weight** (*float*) – Weight of entropy in your loss function, a positive entropy weight will minimize entropy, while a negative one will maximize entropy.
- **value_weight** (*float*) – Weight of critic value loss.
- **gradient_max** (*float*) – Maximum gradient.
- **gae_lambda** (*float*) – λ used in generalized advantage estimation.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **normalize_advantage** (*bool*) –

act (*state*, **_*, ***__*)
Use actor network to give a policy to the current state.

Returns Anything produced by actor.

Parameters **Any] state** (*Dict[str,*) –

store_episode (*episode*)
Add a full episode of transition samples to the replay buffer.

"value" and "gae" are automatically calculated.

Parameters **Dict]] episode** (*List[Union[*machin.frame.transition.Transition*,*) –

store_transition (*transition*)
Add a transition sample to the replay buffer.

Not suggested, since you will have to calculate "value" and "gae" by yourself.

Parameters **Dict] transition** (*Union[*machin.frame.transition.Transition*,*) –

update (*update_value=True*, *update_policy=True*, *concatenate_samples=True*, ***__*)
Update network weights by sampling from buffer. Buffer will be cleared after update is finished.

Parameters

- **update_value** – Whether update the Q network.
- **update_policy** – Whether update the actor network.
- **concatenate_samples** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

update_lr_scheduler ()
Update learning rate schedulers.

A3C

class machin.frame.algorithms.a3c.**A3C**(*actor*, *critic*, *criterion*, *grad_server*, **_*, *entropy_weight=None*, *value_weight=0.5*, *gradient_max=inf*, *gae_lambda=1.0*, *discount=0.99*, *update_times=50*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, ***__*)

Bases: *machin.frame.algorithms.a2c.A2C*

A3C framework.

See also:

A2C

Note: A3C algorithm relies on parameter servers to synchronize parameters of actor and critic models across samplers (interact with environment) and trainers (using samples to train.

The parameter server type *PushPullGradServer* used here utilizes gradients calculated by trainers:

1. perform a "sum" reduction process on the collected gradients, then apply this reduced gradient to the model managed by its primary reducer

2. push the parameters of this updated managed model to a ordered key-value server so that all processes, including samplers and trainers, can access the updated parameters.

The `grad_servers` argument is a pair of accessors to two *PushPullGradServerImpl* class.

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **optimizer** – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **grad_server** (*Tuple[machin.parallel.server.param_server.PushPullGradServer, machin.parallel.server.param_server.PushPullGradServer]*) – Custom gradient sync server accessors, the first server accessor is for actor, and the second one is for critic.
- **entropy_weight** (*float*) – Weight of entropy in your loss function, a positive entropy weight will minimize entropy, while a negative one will maximize entropy.
- **value_weight** (*float*) – Weight of critic value loss.
- **gradient_max** (*float*) – Maximum gradient.
- **gae_lambda** (*float*) – λ used in generalized advantage estimation.
- **discount** (*float*) – γ used in the bellman function.
- **update_times** (*int*) – Number of update iterations per sample period. Buffer will be cleared after `update()`
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.

act (*state*, ***__*)

Use actor network to give a policy to the current state.

Returns Anything produced by actor.

Parameters **Any] state** (*Dict[str,*) –

manual_sync ()

set_sync (*is_syncing*)

update (*update_value=True*, *update_policy=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from buffer. Buffer will be cleared after update is finished.

Parameters

- **update_value** – Whether update the Q network.
- **update_policy** – Whether update the actor network.
- **concatenate_samples** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

PPO

class machin.frame.algorithms.ppo.**PPO**(*actor*, *critic*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=()*, *lr_scheduler_kwargs=()*, *batch_size=100*, *actor_update_times=5*, *critic_update_times=10*, *actor_learning_rate=0.001*, *critic_learning_rate=0.001*, *entropy_weight=None*, *value_weight=0.5*, *surrogate_loss_clip=0.2*, *gradient_max=inf*, *gae_lambda=1.0*, *discount=0.99*, *normalize_advantage=True*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: `machin.frame.algorithms.a2c.A2C`

PPO framework.

See also:

`A2C`

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **actor_update_times** (*int*) – Times to update actor in `update()`.
- **critic_update_times** (*int*) – Times to update critic in `update()`.
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **entropy_weight** (*float*) – Weight of entropy in your loss function, a positive entropy weight will minimize entropy, while a negative one will maximize entropy.
- **value_weight** (*float*) – Weight of critic value loss.
- **surrogate_loss_clip** (*float*) – Surrogate loss clipping parameter in PPO.
- **gradient_max** (*float*) – Maximum gradient.
- **gae_lambda** (*float*) – λ used in generalized advantage estimation.

- **`discount`** (*`float`*) – γ used in the bellman function.
- **`replay_size`** (*`int`*) – Replay buffer size. Not compatible with `replay_buffer`.
- **`replay_device`** (*`Union[str, torch.device]`*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **`replay_buffer`** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer.
- **`visualize`** (*`bool`*) – Whether visualize the network flow in the first pass.
- **`visualize_dir`** (*`str`*) – Visualized graph save directory.
- **`normalize_advantage`** (*`bool`*) –

`update` (*update_value=True*, *update_policy=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from buffer. Buffer will be cleared after update is finished.

Parameters

- **`update_value`** – Whether update the Q network.
- **`update_policy`** – Whether update the actor network.
- **`concatenate_samples`** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

SAC

`class` `machin.frame.algorithms.sac.`**`SAC`**(*actor*, *critic*, *critic_target*, *critic2*, *critic2_target*, *optimizer*, *criterion*, **_*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *target_entropy=None*, *initial_entropy_alpha=1.0*, *batch_size=100*, *update_rate=0.005*, *actor_learning_rate=0.0005*, *critic_learning_rate=0.001*, *alpha_learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, ***__*)

Bases: *`machin.frame.algorithms.base.TorchFramework`*

SAC framework.

See also:

`A2C` *`DDPG`*

Important: When given a state, and an optional action, actor must at least return two values, similar to the actor structure described in *`A2C`*. However, when actor is asked to select an action based on the current state, you must make sure that the sampling process is **differentiable**. E.g. use the `rsample` method of torch distributions instead of the `sample` method.

Compared to other actor-critic methods, SAC embeds the entropy term into its reward function directly, rather than adding the entropy term to actor's loss function. Therefore, we do not use the entropy output of your actor network.

The SAC algorithm uses Q network as critics, so please reference *DDPG* for the requirements and the definition of `action_trans_func`.

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **critic_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target critic network module.
- **critic2** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – The second critic network module.
- **critic2_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – The second target critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor`, `critic` and `critic2`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- ***_** –
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **target_entropy** (*float*) – Target entropy weight α used in the SAC soft value function: $V_{soft}(s_t) = \mathbb{E}_{q_t \sim \pi}[Q_{soft}(s_t, a_t) - \alpha log \pi(a_t|s_t)]$
- **initial_entropy_alpha** (*float*) – Initial entropy weight α
- **gradient_max** (*float*) – Maximum gradient.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.

- **replay_buffer** (machin.frame.buffers.buffer.Buffer) – Custom replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **alpha_learning_rate** (*float*) –

act (*state*, ***__*)
Use actor network to produce an action for the current state.

Returns Anything produced by actor.

Parameters Any] state (*Dict[str,*) –

static action_transform_function (*raw_output_action*, **_*)

load (*model_dir*, *network_map=None*, *version=- 1*)
Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** – Save directory.
- **network_map** – Key is module name, value is saved name.
- **version** – Version number of the save to be loaded.

static reward_function (*reward*, *discount*, *next_value*, *terminal*, *_*)

store_episode (*episode*)
Add a full episode of transition samples to the replay buffer.

Parameters Dict]] episode (*List[Union[*machin.frame.transition.Transition*,*) –

store_transition (*transition*)
Add a transition sample to the replay buffer.

Parameters Dict] transition (*Union[*machin.frame.transition.Transition*,*) –

update (*update_value=True*, *update_policy=True*, *update_target=True*, *update_entropy_alpha=True*, *concatenate_samples=True*, ***__*)
Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.
- **update_target** – Whether to update targets.
- **update_entropy_alpha** – Whether to update *alpha* of entropy.
- **concatenate_samples** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

update_lr_scheduler()
Update learning rate schedulers.

APEX

class machin.frame.algorithms.apex.**DDPGApex**(*actor*, *actor_target*, *critic*, *critic_target*, *optimizer*, *criterion*, *apex_group*, *model_server*, **_*, *lr_scheduler=None*, *lr_scheduler_args=()*, *lr_scheduler_kwargs=()*, *batch_size=100*, *update_rate=0.005*, *learning_rate=0.001*, *discount=0.99*, *replay_size=500000*, ***__*)

Bases: `machin.frame.algorithms.ddpg_per.DDPGPer`

Massively parallel version of a DDPG with prioritized replay.

The pull function is invoked before using `act`, `act_with_noise`, `act_discrete`, `act_discrete_with_noise` and `criticize`.

The push function is invoked after `update`.

See also:

`DDPGPer`

Hint: Your push and pull function will be called like:

```
function(actor_model, "actor")
```

The default implementation of pull and push functions is provided by `PushPullModelServer`.

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **actor_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **critic_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `qnet`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **apex_group** (*machin.parallel.distributed.world.RpcGroup*) – Group of all processes using the apex-DDPG framework, including all samplers and trainers.
- **model_server** (*Tuple[machin.parallel.server.param_server.PushPullModelServer]*) – Custom model sync server accessor for `actor`.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.

- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **learning_rate** (*float*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Local replay buffer size of a single worker.

act (*state*, *use_target=False*, ***__*)

Use actor network to produce an action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether use the target network.

Returns Any thing returned by your actor network.

act_discrete (*state*, *use_target=False*, ***__*)

Use actor network to produce a discrete action for the current state.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Action of shape `[batch_size, 1]`. Action probability tensor of shape `[batch_size, action_num]`, produced by your actor. Any other things returned by your Q network. if they exist.

act_discrete_with_noise (*state*, *use_target=False*, ***__*)

Use actor network to produce a noisy discrete action for the current state.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Noisy action of shape `[batch_size, 1]`. Action probability tensor of shape `[batch_size, action_num]`. Any other things returned by your Q network. if they exist.

act_with_noise(*state*, *noise_param=0.0, 1.0*, *ratio=1.0*, *mode='uniform'*, *use_target=False*, ***__*)
Use actor network to produce a noisy action for the current state.

See also:

machin.frame.noise.action_space_noise

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **noise_param** (*Tuple*) – Noise params.
- **ratio** (*float*) – Noise ratio.
- **mode** (*str*) – Noise mode. Supported are: `"uniform"`, `"normal"`, `"clipped_normal"`, `"ou"`
- **use_target** (*bool*) – Whether use the target network.

Returns Noisy action of shape `[batch_size, action_dim]`. Any other things returned by your actor network. if they exist.

manual_sync()

set_sync(*is_syncing*)

update(*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*, ***__*)
Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.
- **update_target** – Whether to update targets.
- **concatenate_samples** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

class machin.frame.algorithms.apex.**DQNApex**(*qnet*, *qnet_target*, *optimizer*, *criterion*, *apex_group*, *model_server*, **_*, *lr_scheduler=None*, *lr_scheduler_args=()*, *lr_scheduler_kwargs=()*, *batch_size=100*, *update_rate=0.005*, *learning_rate=0.001*, *discount=0.99*, *replay_size=500000*, ***__*)
Bases: *machin.frame.algorithms.dqn_per.DQNPer*

Massively parallel version of a Double DQN with prioritized replay.

The pull function is invoked before using `act_discrete`, `act_discrete_with_noise` and `criticize`.

The push function is invoked after `update`.

See also:

DQNPer

Note: Apex framework supports multiple workers(samplers), and only one trainer, you may use `DistributedDataParallel` in trainer. If you use `DistributedDataParallel`, you must call `update()` in all member processes of `DistributedDataParallel`.

Parameters

- **qnet** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Q network module.
- **qnet_target** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Target Q network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `qnet`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **apex_group** (*machin.parallel.distributed.world.RpcGroup*) – Group of all processes using the apex-DQN framework, including all samplers and trainers.
- **model_server** (*Tuple[machin.parallel.server.param_server.PushPullModelServer]*) – Custom model sync server accessor for `qnet`.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as:

  $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **learning_rate** (*float*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Local replay buffer size of a single worker.

act_discrete (*state*, *use_target=False*, ***__*)

Use Q network to produce a discrete action for the current state.

Parameters

- **state** (*Dict[str, Any]*) – Current state.
- **use_target** (*bool*) – Whether to use the target network.

Returns Action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

`act_discrete_with_noise`(*state*, *use_target=False*, ***__*)

Use Q network to produce a noisy discrete action for the current state.

Parameters

- **`state`** (*`Dict[str, Any]`*) – Current state.
- **`use_target`** (*`bool`*) – Whether to use the target network.

Returns Noisy action of shape `[batch_size, 1]`. Any other things returned by your Q network. if they exist.

`manual_sync`()

`set_sync`(*is_syncing*)

`update`(*update_value=True*, *update_target=True*, *concatenate_samples=True*, ***__*)

Update network weights by sampling from replay buffer.

Parameters

- **`update_value`** – Whether update the Q network.
- **`update_target`** – Whether update targets.
- **`concatenate_samples`** – Whether concatenate the samples.

Returns mean value of estimated policy value, value loss

IMPALA

class `machin.frame.algorithms.impala.`**`EpisodeDistributedBuffer`**(*buffer_name*, *group*, *buffer_size*, **_*, ***__*)

Bases: *`machin.frame.buffers.buffer_d.DistributedBuffer`*

A distributed buffer which stores each episode as a transition object inside the buffer.

Create a distributed replay buffer instance.

To avoid issues caused by tensor device difference, all transition objects are stored in device "cpu".

Distributed replay buffer constitutes of many local buffers held per process, transmissions between processes only happen during sampling.

During sampling, the tensors in "state", "action" and "next_state" dictionaries, along with "reward", will be concatenated in dimension 0. any other custom keys specified in `**kwargs` will not be concatenated.

See also:

`Buffer`

Note: Since `append()` operates on the local buffer, in order to append to the distributed buffer correctly, please make sure that your actor is also the local buffer holder, i.e. a member of the `group`

Parameters

- **`buffer_size`** (*`int`*) – Maximum local buffer size.

- **group** (*machin.parallel.distributed.world.RpcGroup*) – Process group which holds this buffer.
- **buffer_name** (*str*) – A unique name of your buffer.

append(*transition*, *required_attrs='state', 'action', 'next_state', 'reward', 'terminal', 'action_log_prob'*)
Store a transition object to buffer.

Parameters

- **transition** (*Dict*) – A transition object.
- **required_attrs** – Required attributes. Could be an empty tuple if no attribute is required.

Raises

- **ValueError if transition object doesn't have required** –
- **attributes in required_attrs or has different attributes** –
- **compared to other transition objects stored in buffer.** –

class machin.frame.algorithms.impala.**EpisodeTransition**(*state*, *action*, *next_state*, *reward*, *terminal*, ***kwargs*)
Bases: *machin.frame.transition.Transition*

A transition class which allows storing the whole episode as a single transition object, the batch dimension will be used to stack all transition steps.

Parameters

- **state** (*Dict[str, torch.Tensor]*) – Previous observed state.
- **action** (*Dict[str, torch.Tensor]*) – Action of agent.
- **next_state** (*Dict[str, torch.Tensor]*) – Next observed state.
- **reward** (*Union[float, torch.Tensor]*) – Reward of agent.
- **terminal** (*bool*) – Whether environment has reached terminal state.
- ****kwargs** – Custom attributes. They are ordered in the alphabetic order (provided by `sort()`) when you call `keys()`.

Note: You should not store any tensor inside `**kwargs` as they will not be moved to the sample output device.

class machin.frame.algorithms.impala.**IMPALA**(*actor*, *critic*, *optimizer*, *criterion*, *impala_group*, *model_server*, **_*, *lr_scheduler=None*, *lr_scheduler_args=()*, *lr_scheduler_kwargs=()*, *batch_size=5*, *learning_rate=0.001*, *isw_clip_c=1.0*, *isw_clip_rho=1.0*, *entropy_weight=None*, *value_weight=0.5*, *gradient_max=inf*, *discount=0.99*, *replay_size=500*, *visualize=False*, ***__*)
Bases: *machin.frame.algorithms.base.TorchFramework*

Massively parallel IMPALA framework.

Note: Please make sure isw_clip_rho >= isw_clip_c

Parameters

- **actor** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Actor network module.
- **critic** (*Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]*) – Critic network module.
- **optimizer** (*Callable*) – Optimizer used to optimize `actor` and `critic`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **impala_group** (*machin.parallel.distributed.world.RpcGroup*) – Group of all processes using the IMPALA framework, including all samplers and trainers.
- **model_server** (*Tuple[machin.parallel.server.param_server.PushPullModelServer]*) – Custom model sync server accessor for `actor`.
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.
- **batch_size** (*int*) – Batch size used during training.
- **learning_rate** (*float*) – Learning rate of the optimizer, not compatible with `lr_scheduler`.
- **isw_clip_c** (*float*) – c used in importance weight clipping.
- **isw_clip_rho** (*float*) –
- **entropy_weight** (*float*) – Weight of entropy in your loss function, a positive entropy weight will minimize entropy, while a negative one will maximize entropy.
- **value_weight** (*float*) – Weight of critic value loss.
- **gradient_max** (*float*) – Maximum gradient.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Size of the local replay buffer.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.

act (*state*, **_*, ***__*)

Use actor network to give a policy to the current state.

Returns Anything produced by actor.

Parameters Any] state (*Dict[str,*) –

manual_sync ()

set_sync (*is_syncing*)

store_episode (*episode*)

Add a full episode of transition samples to the replay buffer.

Parameters Dict]] episode (*List[Union[*`machin.frame.transition.Transition`*,*) –

store_transition(*transition*)

> **Warning:** Not supported in IMPALA due to v-trace requirements.

Parameters **Dict] transition** (*Union*[machin.frame.transition.Transition,) –

update(*update_value=True*, *update_policy=True*, ***__*)
Update network weights by sampling from replay buffer.

> **Note:** Will always concatenate samples.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.

Returns mean value of estimated policy value, value loss

update_lr_scheduler()
Update learning rate schedulers.

MADDPG

class machin.frame.algorithms.maddpg.**MADDPG**(*actors*, *actor_targets*, *critics*, *critic_targets*, *critic_visible_actors*, *optimizer*, *criterion*, **_*, *sub_policy_num=0*, *lr_scheduler=None*, *lr_scheduler_args=None*, *lr_scheduler_kwargs=None*, *batch_size=100*, *update_rate=0.001*, *actor_learning_rate=0.0005*, *critic_learning_rate=0.001*, *discount=0.99*, *gradient_max=inf*, *replay_size=500000*, *replay_device='cpu'*, *replay_buffer=None*, *visualize=False*, *visualize_dir=''*, *use_jit=True*, *pool_type='thread'*, *pool_size=None*)

Bases: *machin.frame.algorithms.base.TorchFramework*

MADDPG is a centralized multi-agent training framework, it alleviates the unstable reward problem caused by the disturbance of other agents by gathering all agents observations and train a global critic. This global critic observes all actions and all states from all agents.

See also:

DDPG

> **Note:** In order to parallelize agent inference, a process pool is used internally. However, in order to minimize memory copy / CUDA memory copy, the location of all of your models must be either "cpu", or "cuda" (Using multiple CUDA devices is supported).

Note: MADDPG framework **does not require** all of your actors are homogeneous. Each pair of your actors and critcs could be heterogeneous.

Note: Suppose you have three pair of actors and critics, with index 0, 1, 2. If critic 0 can observe the action of actor 0 and 1, critic 1 can observe the action of actor 1 and 2, critic 2 can observe the action of actor 2 and 0, the `critic_visible_actors` should be:

```
[[0, 1], [1, 2], [2, 0]]
```

Note:

This implementation contains:

- Ensemble Training

This implementation does not contain:

- Inferring other agents' policies
- Mixed continuous/discrete action spaces

Parameters

- **actors** (*List[Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]]*) – Actor network modules.
- **actor_targets** (*List[Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]]*) – Target actor network modules.
- **critics** (*List[Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]]*) – Critic network modules.
- **critic_targets** (*List[Union[machin.model.nets.base.NeuralNetworkModule, torch.nn.modules.module.Module]]*) – Target critic network modules.
- **critic_visible_actors** (*List[List[int]]*) – Indexes of visible actors for each critic.
- **optimizer** (*Callable*) – Optimizer used to optimize `actors` and `critics`.
- **criterion** (*Callable*) – Criterion used to evaluate the value loss.
- **sub_policy_num** (*int*) – Times to replicate each actor. Equals to *ensemble_policy_num - 1*
- **lr_scheduler** (*Callable*) – Learning rate scheduler of `optimizer`.
- **lr_scheduler_args** (*Tuple[Tuple, Tuple]*) – Arguments of the learning rate scheduler.
- **lr_scheduler_kwargs** (*Tuple[Dict, Dict]*) – Keyword arguments of the learning rate scheduler.

- **batch_size** (*int*) – Batch size used during training.
- **update_rate** (*float*) – τ used to update target networks. Target parameters are updated as: $\theta_t = \theta * \tau + \theta_t * (1 - \tau)$
- **actor_learning_rate** (*float*) – Learning rate of the actor optimizer, not compatible with `lr_scheduler`.
- **critic_learning_rate** (*float*) – Learning rate of the critic optimizer, not compatible with `lr_scheduler`.
- **discount** (*float*) – γ used in the bellman function.
- **replay_size** (*int*) – Replay buffer size for each actor. Not compatible with `replay_buffer`.
- **replay_device** (*Union[str, torch.device]*) – Device where the replay buffer locates on, Not compatible with `replay_buffer`.
- **replay_buffer** (`machin.frame.buffers.buffer.Buffer`) – Custom replay buffer. Will be replicated for actor.
- **visualize** (*bool*) – Whether visualize the network flow in the first pass.
- **visualize_dir** (*str*) – Visualized graph save directory.
- **use_jit** (*bool*) – Whether use torch jit to perform the forward pass in parallel instead of using the internal pool. Provides significant speed and efficiency advantage, but requires actors and critics convertible to TorchScript.
- **pool_type** (*str*) – Type of the internal execution pool, either "process" or "thread".
- **pool_size** (*int*) – Size of the internal execution pool.
- **gradient_max** (*float*) –

act (*states*, *use_target=False*, ***__*)
Use all actor networks to produce actions for the current state. A random sub-policy from the policy ensemble of each actor will be chosen.

Parameters

- **states** (*List[Dict[str, Any]]*) – A list of current states of each actor.
- **use_target** (*bool*) – Whether use the target network.

Returns A list of anything returned by your actor. If your actor returns multiple values, they will be wrapped in a tuple.

act_discrete (*states*, *use_target=False*)
Use all actor networks to produce discrete actions for the current state. A random sub-policy from the policy ensemble of each actor will be chosen.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **`states`** (*`List[Dict[str, Any]]`*) – A list of current states of each actor.
- **`use_target`** (*`bool`*) – Whether use the target network.

Returns

1. Integer discrete actions of shape `[batch_size, 1]`.
2. Action probability tensors of shape `[batch_size, action_num]`.
3. Any other things returned by your actor.

Return type A list of tuples containing

`act_discrete_with_noise`(*states*, *use_target=False*)

Use all actor networks to produce discrete actions for the current state. A random sub-policy from the policy ensemble of each actor will be chosen.

Notes

actor network must output a probability tensor, of shape (batch_size, action_dims), and has a sum of 1 for each row in dimension 1.

Parameters

- **`states`** (*`List[Dict[str, Any]]`*) – A list of current states of each actor.
- **`use_target`** (*`bool`*) – Whether use the target network.

Returns

1. Integer noisy discrete actions.
2. Action probability tensors of shape `[batch_size, action_num]`.
3. Any other things returned by your actor.

Return type A list of tuples containing

`act_with_noise`(*states*, *noise_param=0.0, 1.0*, *ratio=1.0*, *mode='uniform'*, *use_target=False*, ***__*)

Use all actor networks to produce noisy actions for the current state. A random sub-policy from the policy ensemble of each actor will be chosen.

See also:

`machin.frame.noise.action_space_noise`

Parameters

- **`states`** (*`List[Dict[str, Any]]`*) – A list of current states of each actor.
- **`noise_param`** (*`Any`*) – Noise params.
- **`ratio`** (*`float`*) – Noise ratio.
- **`mode`** (*`str`*) – Noise mode. Supported are: `"uniform"`, `"normal"`, `"clipped_normal"`, `"ou"`
- **`use_target`** (*`bool`*) – Whether use the target network.

Returns A list of noisy actions of shape `[batch_size, action_dim]`.

static action_concat_function(*actions*, **_*)

Parameters **actions** (*List[Dict]*) –

static action_transform_function(*raw_output_action*, **_*)

Parameters **raw_output_action** (*Any*) –

load(*model_dir*, *network_map=None*, *version=- 1*)

Load models.

An example of network map:

```
{"restorable_model_1": "file_name_1",
 "restorable_model_2": "file_name_2"}
```

Get keys by calling `<Class name>.get_restorable()`

Parameters

- **model_dir** – Save directory.
- **network_map** – Key is module name, value is saved name.
- **version** – Version number of the save to be loaded.

static reward_function(*reward*, *discount*, *next_value*, *terminal*, **_*)

static state_concat_function(*states*, **_*)

Parameters **states** (*List[Dict]*) –

store_episodes(*episodes*)

Add a List of full episodes, from all actors, to the replay buffers. Each episode is a list of transition samples.

Parameters **Dict]]] episodes** (*List[List[Union[machin.frame.transition.Transition,*) –

store_transitions(*transitions*)

Add a list of transition samples, from all actors at the same time step, to the replay buffers.

Parameters **transitions** (*List[Union[machin.frame.transition.Transition, Dict]]*) – List of transition objects.

update(*update_value=True*, *update_policy=True*, *update_target=True*, *concatenate_samples=True*)

Update network weights by sampling from replay buffer.

Parameters

- **update_value** – Whether to update the Q network.
- **update_policy** – Whether to update the actor network.
- **update_target** – Whether to update targets.
- **concatenate_samples** – Whether to concatenate the samples.

Returns mean value of estimated policy value, value loss

update_lr_scheduler()

Update learning rate schedulers.

class `machin.frame.algorithms.maddpg.`**`SHMBuffer`**(*buffer_size*, *buffer_device='cpu'*, **_*, ***__*)

Bases: *`machin.frame.buffers.buffer.Buffer`*

Create a buffer instance.

Buffer stores a series of transition objects and functions as a ring buffer. **It is not thread-safe**.

See also:

`Transition`

During sampling, the tensors in "state", "action" and "next_state" dictionaries, along with "reward", will be concatenated in dimension 0. any other custom keys specified in `**kwargs` will not be concatenated.

Parameters

- **`buffer_size`** – Maximum buffer size.
- **`buffer_device`** – Device where buffer is stored.

static make_tensor_from_batch(*batch*, *device*, *concatenate*)

Make a tensor from a batch of data. Will concatenate input tensors in dimension 0. Or create a tensor of size (batch_size, 1) for scalars.

Parameters

- **`batch`** – Batch data.
- **`device`** – Device to move data to
- **`concatenate`** – Whether performing concatenation.

Returns Original batch if batch is empty, or tensor depends on your data (if concatenate), or original batch (if not concatenate).

buffers

Buffer

class `machin.frame.buffers.buffer.`**`Buffer`**(*buffer_size*, *buffer_device='cpu'*, **_*, ***__*)

Bases: `object`

Create a buffer instance.

Buffer stores a series of transition objects and functions as a ring buffer. **It is not thread-safe**.

See also:

`Transition`

During sampling, the tensors in "state", "action" and "next_state" dictionaries, along with "reward", will be concatenated in dimension 0. any other custom keys specified in `**kwargs` will not be concatenated.

Parameters

- **`buffer_size`** – Maximum buffer size.
- **`buffer_device`** – Device where buffer is stored.

append(*transition*, *required_attrs='state', 'action', 'next_state', 'reward', 'terminal'*)

Store a transition object to buffer.

Parameters

- **`transition`** (*`Union[machin.frame.transition.Transition, Dict]`*) – A transition object.
- **`required_attrs`** – Required attributes. Could be an empty tuple if no attribute is required.

Raises

- **`ValueError if transition object doesn't have required`** –
- **`attributes in required_attrs or has different attributes`** –
- **`compared to other transition objects stored in buffer.`** –

`clear``()`
Remove all entries from the buffer

`static make_tensor_from_batch``(`*batch*, *device*, *concatenate*`)`
Make a tensor from a batch of data. Will concatenate input tensors in dimension 0. Or create a tensor of size (batch_size, 1) for scalars.

Parameters

- **`batch`** (*`List[Union[NewType.<locals>.new_type, torch.Tensor]]`*) – Batch data.
- **`device`** (*`Union[str, torch.device]`*) – Device to move data to
- **`concatenate`** (*`bool`*) – Whether performing concatenation.

Returns Original batch if batch is empty, or tensor depends on your data (if concatenate), or original batch (if not concatenate).

`classmethod post_process_batch``(`*batch*, *device*, *concatenate*, *sample_attrs*, *additional_concat_attrs*`)`
Post-process (concatenate) sampled batch.

Parameters

- **`batch`** (*`List[machin.frame.transition.Transition]`*) –
- **`torch.device] device`** (*`Union[str,`*) –
- **`concatenate`** (*`bool`*) –
- **`sample_attrs`** (*`List[str]`*) –
- **`additional_concat_attrs`** (*`List[str]`*) –

`sample_batch``(`*batch_size*, *concatenate=True*, *device=None*, *sample_method='random_unique'*, *sample_attrs=None*, *additional_concat_attrs=None*, **_*, ***__*`)`
Sample a random batch from buffer.

See also:

Default sample methods are defined as static class methods.

`Buffer.sample_method_random_unique()`

`Buffer.sample_method_random()`

`Buffer.sample_method_all()`

Note: "Concatenation" means `torch.cat([...], dim=0)` for tensors, and `torch.tensor([...]).view(batch_size, 1)` for scalars.

Warning: Custom attributes must not contain tensors. And only scalar custom attributes can be concatenated, such as `int`, `float`, `bool`.

Parameters

- **`batch_size`** (*`int`*) – A hint size of the result sample. actual sample size depends on your sample method.
- **`sample_method`** (*`Union[Callable, str]`*) – Sample method, could be one of: `"random"`, `"random_unique"`, `"all"`, or a function: `func(list, batch_size)->(list, result_size)`
- **`concatenate`** (*`bool`*) – Whether concatenate state, action and next_state in dimension 0. If `True`, for each value in dictionaries of major attributes. and each value of sub attributes, returns a concatenated tensor. Custom Attributes specified in `additional_concat_attrs` will also be concatenated. If `False`, return a list of tensors.
- **`device`** (*`Union[str, torch.device]`*) – Device to copy to.
- **`sample_attrs`** (*`List[str]`*) – If sample_keys is specified, then only specified keys of the transition object will be sampled. You may use `"*"` as a wildcard to collect remaining **custom keys** as a `dict`, you cannot collect major and sub attributes using this. Invalid sample attributes will be ignored.
- **`additional_concat_attrs`** (*`List[str]`*) – additional **custom keys** needed to be concatenated, will only work if `concatenate` is `True`.

Returns

1. Batch size, Sampled attribute values in the same order as `sample_keys`.
2. Sampled attribute values is a tuple. Or `None` if sampled batch size is zero (E.g.: if buffer is empty or your sample size is 0 and you are not sampling using the “all” method).
   - For major attributes, result are dictionaries of tensors with the same keys in your transition objects.
   - For sub attributes, result are tensors.
   - For custom attributes, if they are not in `additional_concat_attrs`, then lists, otherwise tensors.

Return type Any

`static sample_method_all`(*buffer*, *_*)

Sample all samples from buffer. Always return the whole buffer, will ignore the `batch_size` parameter.

Parameters **`buffer`** (*`List[machin.frame.transition.Transition]`*) –

Return type Tuple[int, List[*machin.frame.transition.Transition*]]

`static sample_method_random`(*buffer*, *batch_size*)

Sample random samples from buffer.

Note: Sampled size could be any value from 0 to `batch_size`.

Parameters

- **buffer** (*List[machin.frame.transition.Transition]*) –
- **batch_size** (*int*) –

Return type Tuple[int, List[*machin.frame.transition.Transition*]]

static sample_method_random_unique(*buffer*, *batch_size*)
Sample unique random samples from buffer.

Note: Sampled size could be any value from 0 to `batch_size`.

Parameters

- **buffer** (*List[machin.frame.transition.Transition]*) –
- **batch_size** (*int*) –

Return type Tuple[int, List[*machin.frame.transition.Transition*]]

size()

Returns Length of current buffer.

Distributed buffer

class machin.frame.buffers.buffer_d.**DistributedBuffer**(*buffer_name*, *group*, *buffer_size*, **_*, ***__*)
Bases: *machin.frame.buffers.buffer.Buffer*

Create a distributed replay buffer instance.

To avoid issues caused by tensor device difference, all transition objects are stored in device "cpu".

Distributed replay buffer constitutes of many local buffers held per process, transmissions between processes only happen during sampling.

During sampling, the tensors in "state", "action" and "next_state" dictionaries, along with "reward", will be concatenated in dimension 0. any other custom keys specified in `**kwargs` will not be concatenated.

See also:

Buffer

Note: Since `append()` operates on the local buffer, in order to append to the distributed buffer correctly, please make sure that your actor is also the local buffer holder, i.e. a member of the `group`

Parameters

- **buffer_size** (*int*) – Maximum local buffer size.
- **group** (*machin.parallel.distributed.world.RpcGroup*) – Process group which holds this buffer.
- **buffer_name** (*str*) – A unique name of your buffer.

all_clear()
Remove all entries from all local buffers.

all_size()

Returns Total length of all buffers.

append(*transition*, *required_attrs='state', 'action', 'next_state', 'reward', 'terminal'*)
Store a transition object to buffer.

Parameters

- **transition** (*Union[*machin.frame.transition.Transition*, Dict]*) – A transition object.
- **required_attrs** – Required attributes. Could be an empty tuple if no attribute is required.

Raises

- **ValueError if transition object doesn't have required** –
- **attributes in required_attrs or has different attributes** –
- **compared to other transition objects stored in buffer.** –

clear()
Clear current local buffer.

sample_batch(*batch_size*, *concatenate=True*, *device=None*, *sample_method='random_unique'*, *sample_attrs=None*, *additional_concat_attrs=None*, **_*, ***__*)
Sample a random batch from buffer.

See also:

Default sample methods are defined as static class methods.

Buffer.sample_method_random_unique()

Buffer.sample_method_random()

Buffer.sample_method_all()

Note: "Concatenation" means `torch.cat([...], dim=0)` for tensors, and `torch.tensor([...]).view(batch_size, 1)` for scalars.

Warning: Custom attributes must not contain tensors. And only scalar custom attributes can be concatenated, such as `int`, `float`, `bool`.

Parameters

- **batch_size** (*int*) – A hint size of the result sample. actual sample size depends on your sample method.
- **sample_method** (*Union[Callable, str]*) – Sample method, could be one of: `"random"`, `"random_unique"`, `"all"`, or a function: `func(list, batch_size)->(list, result_size)`
- **concatenate** (*bool*) – Whether concatenate state, action and next_state in dimension 0. If `True`, for each value in dictionaries of major attributes. and each value of sub attributes, returns a concatenated tensor. Custom Attributes specified in `additional_concat_attrs` will also be concatenated. If `False`, return a list of tensors.

- **device** (*Union[str, torch.device]*) – Device to copy to.
- **sample_attrs** (*List[str]*) – If sample_keys is specified, then only specified keys of the transition object will be sampled. You may use `"*"` as a wildcard to collect remaining **custom keys** as a `dict`, you cannot collect major and sub attributes using this. Invalid sample attributes will be ignored.
- **additional_concat_attrs** (*List[str]*) – additional **custom keys** needed to be concatenated, will only work if `concatenate` is `True`.

Returns

1. Batch size, Sampled attribute values in the same order as `sample_keys`.
2. Sampled attribute values is a tuple. Or `None` if sampled batch size is zero (E.g.: if buffer is empty or your sample size is 0 and you are not sampling using the "all" method).
   - For major attributes, result are dictionaries of tensors with the same keys in your transition objects.
   - For sub attributes, result are tensors.
   - For custom attributes, if they are not in `additional_concat_attrs`, then lists, otherwise tensors.

Return type Any

size()

Returns Length of current local buffer.

Prioritized buffer

class machin.frame.buffers.prioritized_buffer.**PrioritizedBuffer**(*buffer_size*, *buffer_device='cpu'*, *epsilon=0.01*, *alpha=0.6*, *beta=0.4*, *beta_increment_per_sampling=0.001*, **_*, ***__*)

Bases: *machin.frame.buffers.buffer.Buffer*

Parameters

- **buffer_size** – Maximum buffer size.
- **buffer_device** – Device where buffer is stored.
- **epsilon** – A small positive constant used to prevent edge-case zero weight transitions from never being visited.
- **alpha** – Prioritization weight. Used during transition sampling: $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$. When `alpha = 0`, all samples have the same probability to be sampled. When `alpha = 1`, all samples are drawn uniformly according to their weight.
- **beta** – Bias correcting weight. When `beta = 1`, bias introduced by prioritized replay will be corrected. Used during importance weight calculation: $w_j = (N \cdot P(j))^{-\beta} / max_i w_i$
- **beta_increment_per_sampling** – Beta increase step size, will gradually increase `beta` to 1.

append(*transition*, *priority=None*, *required_attrs='state', 'action', 'next_state', 'reward', 'terminal'*)
Store a transition object to buffer.

Parameters

- **transition** (`Union[machin.frame.transition.Transition, Dict]`) – A transition object.
- **priority** (`Optional[float]`) – Priority of transition.
- **required_attrs** – Required attributes.

clear()
Clear and resets the buffer to its initial state.

sample_batch(*batch_size*, *concatenate=True*, *device=None*, *sample_attrs=None*, *additional_concat_attrs=None*, **_*, ***__*)
Sample the most important batch from the prioritized buffer.

See also:

`Buffer.sample_batch()`

Parameters

- **batch_size** (`int`) – A hint size of the result sample.
- **concatenate** (`bool`) – Whether concatenate state, action and next_state in dimension 0. If `True`, for each value in dictionaries of major attributes. and each value of sub attributes, returns a concatenated tensor. Custom Attributes specified in `additional_concat_attrs` will also be concatenated. If `False`, return a list of tensors.
- **device** (`Union[str, torch.device]`) – Device to copy to.
- **sample_attrs** (`List[str]`) – If sample_keys is specified, then only specified keys of the transition object will be sampled. You may use `"*"` as a wildcard to collect remaining keys.
- **additional_concat_attrs** (`List[str]`) – additional custom keys needed to be concatenated,

Returns

1. Batch size.
2. Sampled attribute values in the same order as `sample_keys`.

   Sampled attribute values is a tuple. Or `None` if sampled batch size is zero (E.g.: if buffer is empty or your sample size is 0).
3. Indexes of samples in the weight tree, `np.ndarray`. Or `None` if sampled batch size is zero
4. Importance sampling weight of samples, `np.ndarray`. Or `None` if sampled batch size is zero

Return type Any

size()

Returns Length of current buffer.

update_priority(*priorities*, *indexes*)
Update priorities of samples.

Parameters

- **`priorities`** (*`numpy.ndarray`*) – New priorities.
- **`indexes`** (*`numpy.ndarray`*) – Indexes of samples, returned by *`sample_batch()`*

Weight tree

class `machin.frame.buffers.prioritized_buffer.`**`WeightTree`**(*size*)

Bases: `object`

Sum weight tree data structure.

Initialize a weight tree.

Note: Weights must be positive.

Note: Weight tree is stored as a flattened, full binary tree in a `np.ndarray`. The lowest level of leaves comes first, the highest root node is stored at last.

Example:

Tree with weights: `[[1, 2, 3, 4], [3, 7], [11]]`

will be stored as: `[1, 2, 3, 4, 3, 7, 11]`

Note: Performance On i7-6700HQ (M: Million):

90ms for building a tree with 10M elements.

230ms for looking up 10M elements in a tree with 10M elements.

20ms for 1M element batched update in a tree with 10M elements.

240ms for 1M element single update in a tree with 10M elements.

Parameters **`size`** – Number of weight tree leaves.

`find_leaf_index`(*weight*)

Find leaf indexes given weight. Weight must be in range $[0, weight\_sum]$

Parameters **`weight`** (*`Union[float, List[float], numpy.ndarray]`*) – Weight(s) used to query leaf index(es).

Returns Leaf index(es), if weight is scalar, returns `int`, if not, returns `np.ndarray`.

`get_leaf_all_weights`()

Returns Current weights of all leaves, `np.ndarray` of shape `(size)`.

Return type numpy.ndarray

`get_leaf_max`()

Returns Current maximum leaf weight.

Return type float

get_leaf_weight(*index*)
Get weights of selected leaves.

Parameters **index** (*Union[int, List[int], numpy.ndarray]*) – Leaf indexes in range `[0, size - 1]`, used to query weights.

Returns Current weight(s) of selected leaves. If index is scalar, returns `float`, if not, returns `np.ndarray`.

Return type Any

get_weight_sum()

Returns Total weight sum.

Return type float

print_weights(*precision=2*)
Pretty print the tree, for debug purpose.

Parameters **precision** – Number of digits of weights to print.

update_all_leaves(*weights*)
Reset all leaf weights, rebuild weight tree from ground up.

Parameters **weights** (*Union[List[float], numpy.ndarray]*) – All leaf weights. List or array length should be in range `[0, size]`.

update_leaf(*weight*, *index*)
Update a single weight tree leaf.

Parameters

- **weight** (*float*) – New weight of the leaf.
- **index** (*int*) – Leaf index to update, must be in range `[0, size - 1]`.

update_leaf_batch(*weights*, *indexes*)
Update weight tree leaves in batch.

Parameters

- **weights** (*Union[List[float], numpy.ndarray]*) – New weights of leaves.
- **indexes** (*Union[List[int], numpy.ndarray]*) – Leaf indexes to update, must be in range `[0, size - 1]`.

Distributed prioritized buffer

class machin.frame.buffers.prioritized_buffer_d.**DistributedPrioritizedBuffer**(*buffer_name*, *group*, *buffer_size*, **_*, ***__*)

Bases: *machin.frame.buffers.prioritized_buffer.PrioritizedBuffer*

Create a distributed prioritized replay buffer instance.

To avoid issues caused by tensor device difference, all transition objects are stored in device "cpu".

Distributed prioritized replay buffer constitutes of many local buffers held per process, since it is very inefficient to maintain a weight tree across processes, each process holds a weight tree of records in its local buffer and a local buffer (same as `DistributedBuffer`).

The sampling process(es) will first use rpc to acquire the wr_lock, signalling "stop" to appending performed by actor processes, then perform a sum of all local weight trees, and finally perform sampling, after sampling and updating the importance weight, the lock will be released.

During sampling, the tensors in "state", "action" and "next_state" dictionaries, along with "reward", will be concatenated in dimension 0. any other custom keys specified in `**kwargs` will not be concatenated.

See also:

`PrioritizedBuffer`

Note: *`DistributedPrioritizedBuffer`* is not split into an accessor and an implementation, because we would like to operate on the buffer directly, when calling "size()" or "append()", to increase efficiency (since rpc layer is bypassed).

Parameters

- **`buffer_size`** (*`int`*) – Maximum local buffer size.
- **`group`** (*`machin.parallel.distributed.world.RpcGroup`*) – Process group which holds this buffer.
- **`buffer_name`** (*`str`*) –

`all_clear`()
Remove all entries from all local buffers.

`all_size`()

Returns Total length of all buffers.

`append`(*transition*, *priority=None*, *required_attrs='state', 'action', 'next_state', 'reward', 'terminal'*)
Store a transition object to buffer.

Parameters

- **`transition`** (*`Union[machin.frame.transition.Transition, Dict]`*) – A transition object.
- **`priority`** (*`Optional[float]`*) – Priority of transition.
- **`required_attrs`** – Required attributes.

`clear`()
Remove all entries from current local buffer.

`sample_batch`(*batch_size*, *concatenate=True*, *device=None*, *sample_attrs=None*, *additional_concat_attrs=None*, **_*, ***__*)
Sample the most important batch from the prioritized buffer.

See also:

`Buffer.sample_batch()`

Parameters

- **`batch_size`** (*`int`*) – A hint size of the result sample.
- **`concatenate`** (*`bool`*) – Whether concatenate state, action and next_state in dimension 0. If `True`, for each value in dictionaries of major attributes. and each value of sub attributes, returns a concatenated tensor. Custom Attributes specified in

`additional_concat_attrs` will also be concatenated. If `False`, return a list of tensors.

- **`device`** (*`Union[str, torch.device]`*) – Device to copy to.
- **`sample_attrs`** (*`List[str]`*) – If sample_keys is specified, then only specified keys of the transition object will be sampled. You may use `"*"` as a wildcard to collect remaining keys.
- **`additional_concat_attrs`** (*`List[str]`*) – additional custom keys needed to be concatenated,

Returns

1. Batch size.
2. Sampled attribute values in the same order as `sample_keys`.

   Sampled attribute values is a tuple. Or `None` if sampled batch size is zero (E.g.: if buffer is empty or your sample size is 0).
3. Indexes of samples in the weight tree, `np.ndarray`. Or `None` if sampled batch size is zero
4. Importance sampling weight of samples, `np.ndarray`. Or `None` if sampled batch size is zero

Return type Any

`size`()

Returns Length of current local buffer.

`update_priority`(*priorities*, *indexes*)

Update priorities of samples.

Parameters

- **`priorities`** (*`numpy.ndarray`*) – New priorities.
- **`indexes`** (*`collections.OrderedDict`*) – Indexes of samples, returned by `sample_batch()`

noise

action_space_noise

`machin.frame.noise.action_space_noise.`**`add_clipped_normal_noise_to_action`**(*action*, *noise_param=0.0, 1.0, -1.0, 1.0, ratio=1.0*)

Add clipped normal noise to action tensor.

Hint: The innermost tuple contains: `(normal_mean, normal_sigma, clip_min, clip_max)`

If `noise_param` is `Tuple[float, float, float, float]`, then the same clipped normal noise will be added to `action[*, :]`.

If `noise_param` is `Iterable[Tuple[float, float, float, float]]`, then for each `action[*, i]` slice i, clipped normal noise with `noise_param[i]` will be applied respectively.

Parameters

- **action** (*torch.Tensor*) – Raw action
- **noise_param** (*Union[Iterable[Tuple], Tuple]*) – Param of the normal noise.
- **ratio** – Sampled noise is multiplied with this ratio.

Returns Action with uniform noise.

`machin.frame.noise.action_space_noise.`**`add_normal_noise_to_action`**(*action*, *noise_param=0.0, 1.0*, *ratio=1.0*)

Add normal noise to action tensor.

Hint: The innermost tuple contains: `(normal_mean, normal_sigma)`

If `noise_param` is `Tuple[float, float]`, then the same normal noise will be added to `action[*, :]`.

If `noise_param` is `Iterable[Tuple[float, float]]`, then for each `action[*, i]` slice i, clipped normal noise with `noise_param[i]` will be applied respectively.

Parameters

- **action** (*torch.Tensor*) – Raw action
- **noise_param** – Param of the normal noise.
- **ratio** – Sampled noise is multiplied with this ratio.

Returns Action with normal noise.

`machin.frame.noise.action_space_noise.`**`add_ou_noise_to_action`**(*action*, *noise_param=None*, *ratio=1.0*, *reset=False*)

Add Ornstein-Uhlenbeck noise to action tensor.

Warning: Ornstein-Uhlenbeck noise generator is shared. And you cannot specify OU noise of different distributions for each of the last dimension of your action.

Parameters

- **action** (*torch.Tensor*) – Raw action
- **noise_param** (*Dict[str, Any]*) – `OrnsteinUhlenbeckGen` params. Used as keyword arguments of the generator. Will only be effective if `reset` is `True`.
- **ratio** – Sampled noise is multiplied with this ratio.
- **reset** – Whether to reset the default Ornstein-Uhlenbeck noise generator.

Returns Action with Ornstein-Uhlenbeck noise.

machin.frame.noise.action_space_noise.**add_uniform_noise_to_action**(*action*, *noise_param=0.0, 1.0*, *ratio=1.0*)

Add uniform noise to action tensor.

Hint: The innermost tuple contains: `(uniform_min, uniform_max)`

If `noise_param` is `Tuple[float, float]`, then the same uniform noise will be added to `action[*, :]`.

If `noise_param` is `Iterable[Tuple[float, float]]`, then for each `action[*, i]` slice i, uniform noise with `noise_param[i]` will be added respectively.

Parameters

- **action** (*torch.Tensor*) – Raw action.
- **noise_param** (*Union[Iterable[Tuple], Tuple]*) – Param of the uniform noise.
- **ratio** (*float*) – Sampled noise is multiplied with this ratio.

Returns Action with uniform noise.

generator

class machin.frame.noise.generator.**ClippedNormalNoiseGen**(*shape*, *mu=0.0*, *sigma=1.0*, *nmin=-1.0*, *nmax=1.0*)

Bases: *machin.frame.noise.generator.NoiseGen*

Normal noise generator.

Example

```
>>> gen = NormalNoiseGen([2, 3], 0, 1)
>>> gen("cuda:0")
tensor([[-0.5957,  0.2360,  1.0999],
        [ 1.6259,  1.2052, -0.0667]], device="cuda:0")
```

Parameters

- **shape** (*Any*) – Output shape.
- **mu** (*float*) – Average mean of normal noise.
- **sigma** (*float*) – Standard deviation of normal noise.
- **nmin** (*float*) –
- **nmax** (*float*) –

class machin.frame.noise.generator.**NoiseGen**

Bases: `abc.ABC`

Base class for noise generators.

reset()

Reset internal states of the noise generator, if it has any.

class machin.frame.noise.generator.**NormalNoiseGen**(*shape*, *mu=0.0*, *sigma=1.0*)

Bases: *machin.frame.noise.generator.NoiseGen*

Normal noise generator.

Example

```
>>> gen = NormalNoiseGen([2, 3], 0, 1)
>>> gen("cuda:0")
tensor([[-0.5957,  0.2360,  1.0999],
        [ 1.6259,  1.2052, -0.0667]], device="cuda:0")
```

Parameters

- **shape** (*Any*) – Output shape.
- **mu** (*float*) – Average mean of normal noise.
- **sigma** (*float*) – Standard deviation of normal noise.

class machin.frame.noise.generator.**OrnsteinUhlenbeckNoiseGen**(*shape*, *mu=0.0*, *sigma=1.0*, *theta=0.15*, *dt=0.01*, *x0=None*)

Bases: *machin.frame.noise.generator.NoiseGen*

Ornstein-Uhlenbeck noise generator. Based on definition:

$X_{n+1} = X_n + \theta(\mu - X_n)\Delta t + \sigma \Delta W_n$

Example

```
>>> gen = OrnsteinUhlenbeckNoiseGen([2, 3], 0, 1)
>>> gen("cuda:0")
tensor([[ 0.1829,  0.1589, -0.1932],
        [-0.1568,  0.0579,  0.2107]], device="cuda:0")
>>> gen.reset()
```

Parameters

- **shape** (*Any*) – Output shape.
- **mu** (*float*) – Average mean of noise.
- **sigma** (*float*) – Weight of the random wiener process.
- **theta** (*float*) – Weight of difference correction.
- **dt** (*float*) – Time step size.
- **x0** (*torch.Tensor*) – Initial x value. Must have the same shape as `shape`.

reset()

Reset the generator to its initial state.

class machin.frame.noise.generator.**UniformNoiseGen**(*shape*, *umin=0.0*, *umax=1.0*)

Bases: *machin.frame.noise.generator.NoiseGen*

Normal noise generator.

Example

```
>>> gen = UniformNoiseGen([2, 3], 0, 1)
>>> gen("cuda:0")
tensor([[0.0745, 0.6581, 0.9572],
        [0.4450, 0.8157, 0.6421]], device="cuda:0")
```

Parameters

- **shape** (*Any*) – Output shape.
- **umin** (*float*) – Minimum value of uniform noise.
- **umax** (*float*) – Maximum value of uniform noise.

param_space_noise

class machin.frame.noise.param_space_noise.**AdaptiveParamNoise**(*initial_stddev=0.1*, *desired_action_stddev=0.1*, *adoption_coefficient=1.01*)

Bases: object

Implements the adaptive parameter space method in <<Parameter space noise for exploration>>.

Hint: Let θ be the standard deviation of noise, and α be the adpotion coefficient, then:

$$\theta_{n+1} = \left\{ \begin{array}{ll} \alpha \theta_k & if\ d(\pi,\tilde{\pi}) \leq \delta, \\ \frac{1}{\alpha} \theta_k & otherwise, \end{array} \right.$$

Noise is directly applied to network parameters.

Parameters

- **initial_stddev** (*float*) – Initial noise standard deviation.
- **desired_action_stddev** (*float*) – Desired standard deviation for
- **adoption_coefficient** (*float*) – Adoption coefficient.

adapt(*distance*)

Update noise standard deviation according to distance.

Parameters **distance** (*float*) – Current distance between the noisy action and clean action.

get_dev()

Returns Current noise standard deviation.

Return type float

`machin.frame.noise.param_space_noise.`**`perturb_model`**(*model*, *perturb_switch*, *reset_switch*, *distance_func=<function <lambda>>*, *desired_action_stddev=0.5*, *noise_generator=<class 'machin.frame.noise.generator.NormalNoiseGen'>*, *noise_generator_args=()*, *noise_generator_kwargs=None*, *noise_generate_function=None*, *debug_backward=False*)

Give model's parameters a little perturbation. Implements <<Parameter space noise for exploration>>.

Note: Only parameters of type `t.Tensor` and gettable from `model.named_parameters()` will be perturbed.

Original parameters will be automatically swapped in during the backward pass, and you can safely call optimizers afterwards.

Hint: 1. `noise_generator` must accept (shape, *args) in its `__init__` function, where shape is the required shape. it also needs to have `__call__(device=None)` which produce a noise tensor on the specified device when invoked.

2. `noise_generate_function` must accept (shape, device, std:float) and return a noise tensor on the specified device.

Example

In order to use this function to perturb your model, you need to:

```python
from machin.utils.helper_classes import Switch
from machin.frame.noise.param_space_noise import perturb_model
from machin.utils.visualize import visualize_graph
import torch as t

dims = 5

t.manual_seed(0)
model = t.nn.Linear(dims, dims)
optim = t.optim.Adam(model.parameters(), 1e-3)
p_switch, r_switch = Switch(), Switch()
cancel = perturb_model(model, p_switch, r_switch)

# you should keep this switch on if you do one training step after
# every sampling step. otherwise you may turn it off in one episode
# and turn it on in the next to speed up training.
r_switch.on()

# turn off/on the perturbation switch to see the difference
p_switch.on()

# do some sampling
action = model(t.ones([dims]))
```

(continues on next page)

(continued from previous page)

```
# in order to let parameter noise adapt to generate noisy actions
# within ``desired_action_stddev``, you must periodically
# use the original model to generate some actions:
p_switch.off()
action = model(t.ones([dims]))

# visualize will not show any leaf noise tensors
# because they are created in t.no_grad() context
# and added in-place.
visualize_graph(action, exit_after_vis=False)

# do some training
loss = (action - t.ones([dims])).sum()
loss.backward()
optim.step()
print(model.weight)

# clear hooks
cancel()
```

Parameters

- **`model`** (*`torch.nn.modules.module.Module`*) – Neural network model.
- **`perturb_switch`** (`machin.utils.helper_classes.Switch`) – The switch used to enable perturbation. If switch is set to `False` (off), then during the forward process, original parameters are used.
- **`reset_switch`** (`machin.utils.helper_classes.Switch`) – The switch used to reset perturbation noise. If switch is set to `True` (on), and `perturb_switch` is also on, then during every forward process, a new set of noise is applied to each param. If only `perturb_switch` is on, then the same set of noisy parameters is used in the forward process and they **will not be updated**.
- **`distance_func`** (*`Callable`*) – Distance function, accepts two tensors produced by `model` (one is noisy), return the distance as float. Used to compare the distance between actions generated by noisy parameters and original parameters.
- **`desired_action_stddev`** (*`float`*) – Desired action standard deviation.
- **`noise_generator`** (*`Any`*) – Noise generator class.
- **`noise_generator_args`** (*`Tuple`*) – Additional args other than shape of the noise generator.
- **`noise_generator_kwargs`** (*`Dict`*) – Additional kwargs other than shape of the noise generator.
- **`noise_generate_function`** (*`Callable`*) – Noise generation function, mutually exclusive with `noise_generator` and `noise_generator_args`.
- **`debug_backward`** – Print a message if the backward hook is correctly executed.

Returns

1. A reset function with no arguments, will swap in original paramters.
2. **A deregister function with no arguments, will deregister all hooks** applied on your model.

transition

class machin.frame.transition.**Transition**(*state*, *action*, *next_state*, *reward*, *terminal*, ***kwargs*)

Bases: *machin.frame.transition.TransitionBase*

The default Transition class.

Have three main attributes: `state`, `action` and `next_state`.

Have two sub attributes: `reward` and `terminal`.

Store one transition step of one agent.

Parameters

- **state** (*Dict[str, torch.Tensor]*) – Previous observed state.
- **action** (*Dict[str, torch.Tensor]*) – Action of agent.
- **next_state** (*Dict[str, torch.Tensor]*) – Next observed state.
- **reward** (*Union[float, torch.Tensor]*) – Reward of agent.
- **terminal** (*bool*) – Whether environment has reached terminal state.
- ****kwargs** – Custom attributes. They are ordered in the alphabetic order (provided by `sort()`) when you call `keys()`.

Note: You should not store any tensor inside `**kwargs` as they will not be moved to the sample output device.

action = None

next_state = None

reward = None

state = None

terminal = None

class machin.frame.transition.**TransitionBase**(*major_attr*, *sub_attr*, *custom_attr*, *major_data*, *sub_data*, *custom_data*)

Bases: `object`

Base class for all transitions

Note: Major attributes store things like state, action, next_states, etc. They are usually **concatenated by their dictionary keys** during sampling, and passed as keyword arguments to actors, critics, etc.

Sub attributes store things like terminal states, reward, etc. They are usually **concatenated directly** during sampling, and used in different algorithms.

Custom attributes store not concatenatable values, usually user specified states, used in models or as special arguments in different algorithms. They will be collected together as a list during sampling, **no further concatenation is performed**.

Parameters

- **major_attr** (*Iterable[str]*) – A list of major attribute names.
- **sub_attr** (*Iterable[str]*) – A list of sub attribute names.

- **`custom_attr`** (*`Iterable[str]`*) – A list of custom attribute names.
- **`major_data`** (*`Iterable[Dict[str, torch.Tensor]]`*) – Data of major attributes.
- **`sub_data`** (*`Iterable[Union[NewType.<locals>.new_type, torch.Tensor]]`*) – Data of sub attributes.
- **`custom_data`** (*`Iterable[Any]`*) – Data of custom attributes.

`has_keys`(*keys*)

Parameters **`keys`** (*`Iterable[str]`*) – A list of keys

Returns A bool indicating whether current transition object contains all specified keys.

`items`()

Returns All attribute values in current transition object.

`keys`()

Returns All attribute names in current transition object. Ordered in: "major_attrs, sub_attrs, custom_attrs"

`to`(*device*)

Move current transition object to another device. will be a no-op if it already locates on that device.

Parameters **`device`** (*`Union[str, torch.device]`*) – A valid pytorch device.

Returns Self.

`property custom_attr`

`property major_attr`

`property sub_attr`

`class` `machin.frame.transition.`**`TransitionStorageBasic`**(*max_size*)

Bases: `list`

TransitionStorageBasic is a linear, size-capped chunk of memory for transitions, it makes sure that every stored transition is copied, and isolated from the passed in transition object.

Parameters **`max_size`** – Maximum size of the transition storage.

`clear`()

Remove all items from list.

`store`(*transition*)

Parameters **`transition`** (`machin.frame.transition.TransitionBase`) – Transition object to be stored

Returns The position where transition is inserted.

Return type int

`class` `machin.frame.transition.`**`TransitionStorageSmart`**(*max_size*)

Bases: *`machin.frame.transition.TransitionStorageBasic`*

TransitionStorageSmart is a smarter, but (potentially) slower storage class for transitions, but in many cases it is as fast as the basic storage and halves memory usage because it only deep copies half of the states.

TransitionStorageSmart will compare the major attributes of the current stored transition object with that of the last stored transition object. And set them to refer to the same tensor.

Sub attributes and custom attributes will be direcly copied.

Args: max_size: Maximum size of the transition storage.

clear()
Remove all items from list.

store(*transition*)

Parameters **transition** (`machin.frame.transition.TransitionBase`) – Transition object to be stored

Returns The position where transition is inserted.

Return type int

4.1.3 machin.model

nets

`machin.model.nets` provides implementations for various popular network architectures.

class `machin.model.nets.`**NeuralNetworkModule**
Bases: `torch.nn.modules.module.Module`, `abc.ABC`

Note: input device and output device are determined by module parameters, your input module / output submodule should not store parameters on more than one device, and you also should not move your output to other devices other than your parameter storage device in forward().

Initializes internal Module state, shared by both nn.Module and ScriptModule.

set_input_module(*input_module*)
Set the input submodule of current module.

Parameters **input_module** (*torch.nn.modules.module.Module*) –

set_output_module(*output_module*)
Set the output submodule of current module.

Parameters **output_module** (*torch.nn.modules.module.Module*) –

property input_device

property output_device

static find_child(*seq*, *is_first=True*)
Find the first / last leaf child module.

forward(**_*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training = None

`machin.model.nets.`**`dynamic_module_wrapper`**(*wrapped_module*)

Wrapped module must locate on one single device, but could be moved around.

Input device and output device are automatically detected.

Parameters **`wrapped_module`** (*`torch.nn.modules.module.Module`*) –

`machin.model.nets.`**`static_module_wrapper`**(*wrapped_module*, *input_device*, *output_device*)

Wrapped module could locate on multiple devices, but must not be moved.

Input device and output device are statically specified by user.

Parameters

- **`wrapped_module`** (*`torch.nn.modules.module.Module`*) –
- **`torch.device] input_device`** (*`Union[str,`*) –
- **`torch.device] output_device`** (*`Union[str,`*) –

`class` `machin.model.nets.`**`ResNet`**(*in_planes*, *depth*, *out_planes*, *out_pool_size=1, 1*, *norm='none'*)

Bases: `machin.model.nets.base.NeuralNetworkModule`

Create a resnet of specified depth.

Parameters

- **`in_planes`** (*`int`*) – Number of input planes.
- **`depth`** (*`int`*) – Depth of resnet. Could be one of `18, 34, 50, 101, 152`.
- **`out_planes`** (*`int`*) – Number of output planes.
- **`out_pool_size`** – Size of pooling output
- **`norm`** – Normalization method, could be one of "none", "batch" or "weight".

`training = None`

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

4.1.4 machin.utils

checker

`exception` `machin.utils.checker.`**`CheckError`**

Bases: `Exception`

`machin.utils.checker.`**`check_model`**(*writer*, *model*, *input_check_hooks=(<function i_chk_nan>, <function i_chk_range>)*, *output_check_hooks=(<function o_chk_nan>, <function o_chk_range>)*, *param_check_hooks=(<function p_chk_nan>, <function p_chk_range>)*, *input_check_interval=1*, *output_check_interval=1*, *param_check_interval=100*, *name=''*)

Check model input, output and parameters using hooks. All hooks (Input, output and parameter) check hooks are executed in the forward pass.

An example:

```
model = nn.Linear([100, 100])
check_model(model)

# Continue to do whatever you like.
model(t.zeros([100]))
```

Note: Only leaf modules will be checked (such as `nn.Linear` and not some complex neural network modules made of several sub-modules). But you can manually control granularity.

Warning: Do not output `tuple` in your `forward()` function if you have output check hooks, otherwise you must specify names for each output.

Hint: You may manually control the check granularity by using *`mark_as_atom_module()`*.

You may specify a list of names for your module outputs so names given to your output check hooks will not be numbers, by using *`mark_module_output()`*

Hint: For all three kinds of hooks, your hook need to have the following signature:

`hook(counter, writer, model, module, name, value)`

where:

- `counter` is the *`Counter`*, you can use *`Counter.get()`* to get the current pass number.
- `writer` is `SummaryWriter` from `tensorboardx`.
- `model` is your model.
- `module` is the module currently being checked.
- `name` is input/output/parameter name string. For input, their detail names will be extracted from module `forward` signature. Output detail names will be numbers or names you have specified.
- `value` is input/output/parameter value.

Parameters

- **`writer`** (*`tensorboardX.writer.SummaryWriter`*) – Tensorboard `SummaryWriter` used to log.
- **`model`** (*`torch.nn.modules.module.Module`*) – Model to be checked.
- **`input_check_hooks`** – A series of input check hooks.
- **`output_check_hooks`** – A series of output check hooks.
- **`param_check_hooks`** – A series of parameter check hooks.
- **`input_check_interval`** – Interval (number of forward passes) of input checking.

- **`output_check_interval`** – Interval (number of forward passes) of output checking.
- **`param_check_interval`** – Interval (number of backward passes) of parameter checking.
- **`name`** – Your model name.

Returns A function `f()`, calling `f()` will deregister all check hooks.

`machin.utils.checker.`**`check_nan`**(*tensor*, *name=''*)

Check whether tensor has `nan` element.

Parameters

- **`tensor`** (*`torch.Tensor`*) – Tensor to check
- **`name`** – Name of tensor, will be printed in the error message.

Raises **`RuntimeError if tensor has any nan element.`** –

`machin.utils.checker.`**`check_shape`**(*tensor*, *required_shape*, *name=''*)

Check whether tensor has the specified shape.

Parameters

- **`tensor`** (*`torch.Tensor`*) – Tensor to check.
- **`required_shape`** (*`List[int]`*) – A list of `int` specifying shape of each dimension.
- **`name`** – Name of tensor, will be printed in the error message.

Raises **`RuntimeError if shape of the tensor doesn't match.`** –

`machin.utils.checker.`**`i_chk_nan`**(*_counter*, *_writer*, *_model*, *_module*, *input_name*, *input_val*)

Check whether there is any nan element in the input, if input is a tensor.

`machin.utils.checker.`**`i_chk_range`**(*counter*, *writer*, *_model*, *_module*, *input_name*, *input_val*)

Compute min, max and mean value of the input, if input is a tensor.

`machin.utils.checker.`**`mark_as_atom_module`**(*module*)

Mark module as a atom leaf module, so it can be checked.

`machin.utils.checker.`**`mark_module_output`**(*module*, *output_names*)

Mark names of the module output. It will also tell checker about the number of outputs.

Parameters

- **`module`** – Module to be marked.
- **`output_names`** (*`List[str]`*) – Name of each output value.

`machin.utils.checker.`**`o_chk_nan`**(*_counter*, *_writer*, *_model*, *_module*, *output_name*, *output_val*)

Check whether there is any nan element in the output, if input is a tensor.

`machin.utils.checker.`**`o_chk_range`**(*counter*, *writer*, *_model*, *_module*, *output_name*, *output_val*)

Compute min, max and mean value of the output, if output is a tensor.

`machin.utils.checker.`**`p_chk_nan`**(*counter*, *_writer*, *_model*, *_module*, *param_name*, *param_val*)

Check whether there is any nan element in the parameter.

`machin.utils.checker.`**`p_chk_range`**(*counter*, *writer*, *_model*, *_module*, *param_name*, *param_val*)

Compute min, max and mean value of the parameter.

conf

class machin.utils.conf.**Config**(***configs*)

Bases: *machin.utils.helper_classes.Object*

machin.utils.conf.**load_config_cmd**(*merge_conf=None*)

Get configs from the commandline by using "–conf".

`--conf a=b` will set `<Returned Config>.a = b`

Example:

```
python3 test.py --conf device="cuda:1"
                --conf some_dict={"some_key":1}
```

Example:

```
from machin.utils.conf import Config
from machin.utils.save_env import SaveEnv

# set some config attributes
c = Config(
    model_save_int = 100,
    root_dir = "some_directory",
    restart_from_trial = "2020_05_09_15_00_31"
)

load_config_cmd(c)

# restart_from_trial specifies the trial name in your root
# directory.
# If it is set, then SaveEnv constructor will
# load arguments from that trial record, will overwrite.
# If not, then SaveEnv constructor will save configurations
# as: ``<c.some_root_dir>/<trial_start_time>/config/config.json``

save_env = SaveEnv(c)
```

Parameters **merge_conf** (machin.utils.conf.Config) – Config to merge.

Return type *machin.utils.conf.Config*

machin.utils.conf.**load_config_file**(*json_file*, *merge_conf=None*)

Get configs from a json file.

Parameters

- **json_file** (*str*) – Path to the json config file.
- **merge_conf** (machin.utils.conf.Config) – Config to merge.

Returns configuration

Return type *machin.utils.conf.Config*

machin.utils.conf.**merge_config**(*conf*, *merge*)

Merge config object with a dictionary, or a Config object, same keys in the `conf` will be overwritten by keys in `merge`.

Parameters

- **conf** (machin.utils.conf.Config) –

- **machin.utils.conf.Config] merge** (*Union[dict,*) –

Return type *machin.utils.conf.Config*

machin.utils.conf.**save_config**(*conf*, *json_file*)
Dump config object to a json file.

Parameters

- **conf** (machin.utils.conf.Config) –
- **json_file** (*str*) –

helper_classes

class machin.utils.helper_classes.**Counter**(*start=0*, *step=1*)
Bases: object

count()
Move counter forward by step

get()
Get the internal number of counter.

reset()
Reset the counter.

class machin.utils.helper_classes.**Object**(*data=None*, *const_attrs=None*)
Bases: object

An generic object class, which stores a dictionary internally, and you can access and set its keys by accessing and seting attributes of the object.

data
Internal dictionary.

attr(*item*, *value=None*, *change=False*)

call(**args*, ***kwargs*)
the implementation of Object.__call__, override it to customize call behavior.

class machin.utils.helper_classes.**Switch**(*state=False*)
Bases: object

Parameters state (*bool*) – Internal state, True for on, False for off.

flip()
Inverse the internal state.

get()

Returns state of switch.

Return type bool

off()
Set to off.

on()
Set to on.

class machin.utils.helper_classes.**Timer**
Bases: object

begin()
Begin timing.

end()

Returns Curent time difference since last `begin()`

class `machin.utils.helper_classes.`**Trigger**(*state=False*)
Bases: *machin.utils.helper_classes.Switch*

Parameters **state** (*bool*) – Internal state, `True` for on, `False` for off.

get()
Get the state of trigger, will also set trigger to off.

Returns state of trigger.

learning_rate

This module is the place for all learning rate functions, currently, only manual learning rate changing according to global steps is implemented,.

`machin.utils.learning_rate.`**gen_learning_rate_func**(*lr_map*, *logger=None*)
Example:

```
from torch.optim.lr_scheduler import LambdaLR

# 0 <= step < 200000, lr=1e-3, 200000 <= step, lr=3e-4
lr_func = gen_learning_rate_func([(0, 1e-3), (200000, 3e-4)],)
lr_sch = LambdaLR(optimizer, lr_func)
```

Parameters

- **lr_map** (*List[Tuple[int, float]]*) – A 2d learning rate map, the first element of each row is step. the second is learning rate.
- **logger** (*logging.Logger*) – A logger to log current learning rate

Returns A learning rate generation function with signature *lr_gen(step)->lr*, accepts int and returns float. use it in your pytorch lr scheduler.

logging

`machin.utils.logging.`**default_logger**
The default global logger.

TODO: maybe add logging utilities for distributed scenario?

class `machin.utils.logging.`**FakeLogger**
Bases: `object`

critical(*msg*, **args*, ***kwargs*)

debug(*msg*, **args*, ***kwargs*)

error(*msg*, **args*, ***kwargs*)

exception(*msg*, **args*, *exc_info=True*, ***kwargs*)

info(*msg*, **args*, ***kwargs*)

log(*level*, *msg*, **args*, ***kwargs*)

setLevel(*level*)

warn(*msg*, **args*, ***kwargs*)

warning(*msg*, **args*, ***kwargs*)

media

`machin.utils.media.`**create_image**(*image*, *path*, *filename*, *extension='.png'*)

Parameters

- **image** (`numpy.array`) – A numpy array of shape (H, W, C) or (H, W), and with `dtype` = any float or any int. When a frame is float type, its value range should be [0, 1]. When a frame is integer type, its value range should be [0, 255].
- **path** (`str`) – Directory to save the image.
- **filename** (`str`) – File name.
- **extension** (`str`) – File extension.

`machin.utils.media.`**create_image_subproc**(*image*, *path*, *filename*, *extension='.png'*, *daemon=True*)

Create image with a subprocess.

See also:

`create_image()`

Note: if `daemon` is true, then this function cannot be used in a daemonic subprocess.

Parameters

- **image** (`numpy.array`) – A numpy array of shape (H, W, C) or (H, W), and with `dtype` = any float or any int. When a frame is float type, its value range should be [0, 1]. When a frame is integer type, its value range should be [0, 255].
- **path** (`str`) – Directory to save the image.
- **filename** (`str`) – File name.
- **extension** (`str`) – File extension.
- **daemon** (`bool`) – Whether launching the saving process as a daemonic process.

Returns A wait function, once called, block until creation has finished.

`machin.utils.media.`**create_video**(*frames*, *path*, *filename*, *extension='.gif'*, *fps=15*)

Parameters

- **frames** (`List[numpy.array]`) – A list of numpy arrays of shape (H, W, C) or (H, W), and with `dtype` = any float or any int. When a frame is float type, its value range should be [0, 1]. When a frame is integer type, its value range should be [0, 255].
- **path** (`str`) – Directory to save the video.
- **filename** (`str`) – File name.
- **extension** (`str`) – File extension.
- **fps** (`int`) – frames per second.

machin.utils.media.**create_video_subproc**(*frames*, *path*, *filename*, *extension='.gif'*, *fps=15*, *daemon=True*)

Create video with a subprocess, since it takes a lot of time for `moviepy` to encode the video file.

See also:

`create_video()`

Note: if `daemon` is true, then this function cannot be used in a daemonic subprocess.

Parameters

- **frames** (*List[numpy.array]*) – A list of numpy arrays of shape (H, W, C) or (H, W), and with `dtype` = any float or any int. When a frame is float type, its value range should be [0, 1]. When a frame is integer type, its value range should be [0, 255].
- **path** (*str*) – Directory to save the video.
- **filename** (*str*) – File name.
- **extension** (*str*) – File extension.
- **fps** (*int*) – frames per second.
- **daemon** (*bool*) – Whether launching the saving process as a daemonic process.

Returns A wait function, once called, block until creation has finished.

machin.utils.media.**show_image**(*image*, *show_normalized=True*, *pause_time=0.01*, *title=''*)

Use matplotlib to show a single image. You may repeatedly call this method with the same `title` argument to show a video or a dynamically changing image.

Parameters

- **image** (*numpy.array*) – A numpy array of shape (H, W, C) or (H, W), and with `dtype` = any float or any int. When a frame is float type, its value range should be [0, 1]. When a frame is integer type, its value range should be [0, 255].
- **show_normalized** (*bool*) – Show normalized image alongside the original one.
- **pause_time** (*float*) – Pause time between displaying current image and the next one.
- **title** (*str*) – Title of the display window.

prepare

machin.utils.prepare.**prep_clear_dirs**(*dirs*)

Parameters dirs (*Iterable[str]*) – a list of directories to clear

machin.utils.prepare.**prep_create_dirs**(*dirs*)

Note: will recursively create directories.

Parameters dirs (*Iterable[str]*) – a list of directories to create if these directories are not found.

machin.utils.prepare.**prep_load_model**(*model_dir*, *model_map*, *version=None*, *quiet=False*, *logger=None*)

Automatically find and load models.

Parameters

- **model_dir** (*str*) – Directory to save models.
- **model_map** (*Dict[str, torch.nn.modules.module.Module]*) – Model saving map.
- **version** (*int*) – Version to load, if specified, otherwise automatically find the latest version.
- **quiet** (*bool*) – Raise no error if no valid version could be found.
- **logger** (*Any*) – Logger to use.

machin.utils.prepare.**prep_load_state_dict**(*model*, *state_dict*)

Automatically load a **loaded state dictionary**

Note: This function handles tensor device remapping.

Parameters

- **model** (*torch.nn.modules.module.Module*) –
- **state_dict** (*Any*) –

save_env

class machin.utils.save_env.**SaveEnv**(*env_root*, *restart_from_trial=None*, *time_format='%Y_%m_%d_%H_%M_%S'*)

Bases: object

Create the default environment for saving. creates something like:

```
<your environment root>
├── config
├── log
│   ├── images
│   └── train_log
└── model
```

Parameters

- **env_root** (*str*) – root directory for all trials of the environment.
- **restart_from_trial** (*Optional[str]*) – instead of creating a new save environment for a new trial, use a existing save environment of an older trial, old trial name should be in format time_format

clear_trial_config_dir()

clear_trial_image_dir()

clear_trial_model_dir()

clear_trial_train_log_dir()

create_dirs(*dirs*)

Create additional directories in root.

Parameters **dirs** (*Iterable[str]*) – Directories.

get_trial_config_dir()

get_trial_image_dir()

get_trial_model_dir()

get_trial_root()

get_trial_time()

get_trial_train_log_dir()

remove_trials_older_than(*diff_day=0*, *diff_hour=1*, *diff_minute=0*, *diff_second=0*)

By default this function removes all trials started one hour earlier than current time.

Parameters

- **diff_day** (`int`) – Difference in days.
- **diff_hour** (`int`) – Difference in hours.
- **diff_minute** (`int`) – Difference in minutes.
- **diff_second** (`int`) – Difference in seconds.

tensor_board

`machin.utils.tensor_board.`**default_board**

The default global board.

class `machin.utils.tensor_board.`**TensorBoard**

Bases: `object`

Create a tensor board object.

writer

`SummaryWriter` of package `tensorboardX`.

init(**writer_args*)

is_inited()

Returns: whether the board has been initialized with a writer.

Return type bool

visualize

`machin.utils.visualize.`**visualize_graph**(*final_tensor*, *visualize_dir=''*, *exit_after_vis=True*)

Visualize a pytorch flow graph

Parameters

- **final_tensor** – The last output tensor of the flow graph
- **visualize_dir** – Directory to place the visualized files
- **exit_after_vis** – Whether to exit the whole program after visualization.

4.1.5 machin.parallel

distributed

class `machin.parallel.distributed.`**`CollectiveGroup`**(*group*, *current_relative_rank*)

Bases: `object`

A thin wrapper of collective communication primitives of `torch.distributed`, the only difference is that `irecv` now supports to recv from any src

Do not do it your self, use `create_collective_group()` instead.

`all_gather`(*tensor_list*, *tensor*, *async_op=False*)

Complex tensors are supported.

Parameters

- **`tensor_list`** (`list[Tensor]`) – Output list. It should contain correctly-sized tensors to be used for output of the collective.
- **`tensor`** (`Tensor`) – Tensor to be broadcast from current process.
- **`async_op`** (`bool, optional`) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

Examples

```
>>> # All tensors below are of torch.int64 dtype.
>>> tensor_list = [torch.zero(2, dtype=torch.int64) for _ in range(2)]
>>> tensor_list
[tensor([0, 0]), tensor([0, 0])] # Rank 0 and 1
>>> tensor = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> tensor
tensor([1, 2]) # Rank 0
tensor([3, 4]) # Rank 1
>>> dist.all_gather(tensor_list, tensor)
>>> tensor_list
[tensor([1, 2]), tensor([3, 4])] # Rank 0
[tensor([1, 2]), tensor([3, 4])] # Rank 1
```

```
>>> # All tensors below are of torch.cfloat dtype.
>>> tensor_list = [torch.zero(2, dtype=torch.cfloat) for _ in range(2)]
>>> tensor_list
[tensor([0.+0.j, 0.+0.j]), tensor([0.+0.j, 0.+0.j])] # Rank 0 and 1
>>> tensor = torch.tensor([1+1j, 2+2j], dtype=torch.cfloat) + 2 * rank *␣
↪(1+1j)
>>> tensor
tensor([1.+1.j, 2.+2.j]) # Rank 0
tensor([3.+3.j, 4.+4.j]) # Rank 1
>>> dist.all_gather(tensor_list, tensor)
>>> tensor_list
[tensor([1.+1.j, 2.+2.j]), tensor([3.+3.j, 4.+4.j])] # Rank 0
[tensor([1.+1.j, 2.+2.j]), tensor([3.+3.j, 4.+4.j])] # Rank 1
```

`all_gather_multigpu`(*output_tensor_lists*, *input_tensor_list*, *async_op=False*)

Each tensor in `tensor_list` should reside on a separate GPU

Only nccl backend is currently supported tensors should only be GPU tensors

Complex tensors are supported.

Parameters

- **output_tensor_lists** (*List[List[Tensor]]*) – Output lists. It should contain correctly-sized tensors on each GPU to be used for output of the collective, e.g. `output_tensor_lists[i]` contains the all_gather result that resides on the GPU of `input_tensor_list[i]`.

  Note that each element of `output_tensor_lists` has the size of `world_size * len(input_tensor_list)`, since the function all each element of `output_tensor_lists[i]`, note that `input_tensor_list[j]` of rank k will be appear in `output_tensor_lists[i][k * world_size + j]`

  Also note that `len(output_tensor_lists)`, and the size of each element in `output_tensor_lists` (each element is a list, therefore `len(output_tensor_lists[i])`) need to be the same for all the distributed processes calling this function.

- **input_tensor_list** (*List[Tensor]*) – List of tensors(on different GPUs) to be broadcast from current process. Note that `len(input_tensor_list)` needs to be the same for all the distributed processes calling this function.

- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

all_reduce(*tensor*, *op=<ReduceOp.SUM: 0>*, *async_op=False*)

Reduces the tensor data across all machines in such a way that all get the final result.

After the call `tensor` is going to be bitwise identical in all processes.

Complex tensors are supported.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

Examples

```
>>> # All tensors below are of torch.int64 type.
>>> tensor = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> tensor
tensor([1, 2]) # Rank 0
tensor([3, 4]) # Rank 1
>>> dist.all_reduce(tensor, op=ReduceOp.SUM)
>>> tensor
tensor([4, 6]) # Rank 0
tensor([4, 6]) # Rank 1
```

```
>>> # All tensors below are of torch.cfloat type.
>>> tensor = torch.tensor([1+1j, 2+2j], dtype=torch.cfloat) + 2 * rank *␣
↪(1+1j)
```

(continues on next page)

(continued from previous page)

```
>>> tensor
tensor([1.+1.j, 2.+2.j]) # Rank 0
tensor([3.+3.j, 4.+4.j]) # Rank 1
>>> dist.all_reduce(tensor, op=ReduceOp.SUM)
>>> tensor
tensor([4.+4.j, 6.+6.j]) # Rank 0
tensor([4.+4.j, 6.+6.j]) # Rank 1
```

all_reduce_multigpu(*tensor_list*, *op=<ReduceOp.SUM: 0>*, *async_op=False*)

Reduces the tensor data across all machines in such a way that all get the final result. This function reduces a number of tensors on every node, while each tensor resides on different GPUs. Therefore, the input tensor in the tensor list needs to be GPU tensors. Also, each tensor in the tensor list needs to reside on a different GPU.

After the call, all `tensor` in `tensor_list` is going to be bitwise identical in all processes.

Complex tensors are supported.

Only nccl and gloo backend is currently supported tensors should only be GPU tensors

Parameters

- **list** (*tensor*) – List of input and output tensors of the collective. The function operates in-place and requires that each tensor to be a GPU tensor on different GPUs. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

barrier(*async_op=False*)

Synchronizes all processes.

if async_op is False, or if async work handle is called on wait().

Parameters

- **async_op** (*bool, optional*) – Whether this op should be an async op
- **device_ids** (*[int], optional*) – List of device/GPU ids. Valid only for NCCL backend.

Returns Async work handle, if async_op is set to True.

broadcast(*tensor*, *src*, *async_op=False*)

`tensor` must have the same number of elements in all processes participating in the collective.

Parameters

- **tensor** (*Tensor*) – Data to be sent if `src` is the rank of current process, and tensor to be used to save received data otherwise.
- **src** (*int*) – Source rank.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

broadcast_multigpu(*tensor_list*, *src*, *async_op=False*, *src_tensor=0*)

per node.

`tensor` must have the same number of elements in all the GPUs from all processes participating in the collective. each tensor in the list must be on a different GPU

Only nccl and gloo backend are currently supported tensors should only be GPU tensors

Parameters

- **tensor_list** (*List[Tensor]*) – Tensors that participate in the collective operation. If `src` is the rank, then the specified `src_tensor` element of `tensor_list` (`tensor_list[src_tensor]`) will be broadcast to all other tensors (on different GPUs) in the src process and all tensors in `tensor_list` of other non-src processes. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.
- **src** (*int*) – Source rank.
- **async_op** (*bool, optional*) – Whether this op should be an async op
- **src_tensor** (*int, optional*) – Source tensor rank within `tensor_list`

Returns Async work handle, if async_op is set to True.

destroy()

Destroy this collective communication group.

gather(*tensor*, *gather_list*, *dst=0*, *async_op=False*)

Gathers a list of tensors in a single process.

Parameters

- **tensor** (*Tensor*) – Input tensor.
- **gather_list** (*list[Tensor], optional*) – List of appropriately-sized tensors to use for gathered data (default is None, must be specified on the destination rank)
- **dst** (*int, optional*) – Destination rank (default is 0)
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

irecv(*tensor*, *src=None*, *tag=0*)

Returns An object you can call .wait() on, .wait() will return the source rank.

isend(*tensor*, *dst*, *tag=0*)

Sends a tensor asynchronously.

Parameters

- **tensor** (*Tensor*) – Tensor to send.
- **dst** (*int*) – Destination rank.
- **tag** (*int, optional*) – Tag to match send with remote recv

Returns A distributed request object.

recv(*tensor*, *src=None*, *tag=0*)

Receives a tensor synchronously.

Parameters

- **tensor** (*Tensor*) – Tensor to fill with received data.

- **src** (*int, optional*) – Source rank. Will receive from any process if unspecified.
- **tag** (*int, optional*) – Tag to match recv with remote send

Returns Sender rank

reduce (*tensor*, *dst*, *op=<ReduceOp.SUM: 0>*, *async_op=False*)
Reduces the tensor data across all machines.

Only the process with rank `dst` is going to receive the final result.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

reduce_multigpu (*tensor_list*, *dst*, *op=<ReduceOp.SUM: 0>*, *async_op=False*, *dst_tensor=0*)
Reduces the tensor data on multiple GPUs across all machines. Each tensor in `tensor_list` should reside on a separate GPU

Only the GPU of `tensor_list[dst_tensor]` on the process with rank `dst` is going to receive the final result.

Only nccl backend is currently supported tensors should only be GPU tensors

Parameters

- **tensor_list** (*List[Tensor]*) – Input and output GPU tensors of the collective. The function operates in-place. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **async_op** (*bool, optional*) – Whether this op should be an async op
- **dst_tensor** (*int, optional*) – Destination tensor rank within `tensor_list`

Returns Async work handle, if async_op is set to True. None, otherwise

scatter (*tensor*, *scatter_list=None*, *src=0*, *async_op=False*)
Each process will receive exactly one tensor and store its data in the `tensor` argument.

Parameters

- **tensor** (*Tensor*) – Output tensor.
- **scatter_list** (*list[Tensor]*) – List of tensors to scatter (default is None, must be specified on the source rank)
- **src** (*int*) – Source rank (default is 0)
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True.

send (*tensor*, *dst*, *tag=0*)
Sends a tensor synchronously.

Parameters

- **tensor** (*Tensor*) – Tensor to send.
- **dst** (*int*) – Destination rank.
- **tag** (*int, optional*) – Tag to match send with remote recv

size()

Returns collective group size.

class machin.parallel.distributed.**RpcGroup**(*group_name*, *group_members*, *first_create=True*)

Bases: object

barrier(**args*, ***kwargs*)

deregister(**args*, ***kwargs*)

destroy(**args*, ***kwargs*)

static get_cur_name()

Return type str

get_group_members()

Returns A list of group members.

Return type List[str]

get_group_name()

Returns Name of this group.

Return type str

get_paired(*key*)

Parameters **key** (*Any*) – Key of the paired value, in this group.

Returns A RRef to the paired value.

Raises **KeyError if not found.** –

is_member(*target=None*)

Check whether target name is a group member.

Parameters **target** (*str*) –

Return type bool

is_paired(*key*)

Check whether a key has been paired to the current group.

Parameters **key** (*Any*) – A key which uniquely identifies this value in this group. The name only needs to be unique for this value in this group.

is_registered(*key*)

Check whether a service has been registered in the current group.

Parameters **key** (*Any*) – A key which uniquely identifies this service in this group. The name only needs to be unique for this service in this group.

pair(**args*, ***kwargs*)

register(**args*, ***kwargs*)

registered_async(*key*, *args=()*, *kwargs=None*)

Parameters

- **`key`** (*`Any`*) – Key of the registered service, in this group.
- **`args`** – Service arguments.
- **`kwargs`** – Service keyword arguments.

Returns A future object you can call `wait()``on. ``wait()` will block the thread until execution is completed, and will return the result returned by the service.

Raises `KeyError if service is not found.` –

`registered_remote`(*key*, *args=()*, *kwargs=None*)

Parameters

- **`key`** (*`Any`*) – Key of the registered service, in this group.
- **`args`** – Service arguments.
- **`kwargs`** – Service keyword arguments.

Returns A RRef object pointing to the result returned by the service.

Raises `KeyError if service is not found.` –

`registered_sync`(*key*, *args=()*, *kwargs=None*)

Parameters

- **`key`** (*`Any`*) – Key of the registered service, in this group.
- **`args`** – Service arguments.
- **`kwargs`** – Service keyword arguments.

Returns Result returned by the service.

Raises `KeyError if service is not found.` –

`remote`(*to*, *func*, *timeout=- 1*, *args=()*, *kwargs=None*)

Make a remote call to run `func` on worker `to` and return an `RRef` to the result value immediately. Worker `to` will be the owner of the returned `RRef`, and the worker calling `remote` is a user. The owner manages the global reference count of its `RRef`, and the owner `RRef` is only destructed when globally there are no living references to it.

Parameters

- **`to`** (*`str or WorkerInfo or int`*) – name/rank/`WorkerInfo` of the destination worker.
- **`func`** (*`callable`*) – a callable function, such as Python callables, builtin operators (e.g. `add()`) and annotated TorchScript functions.
- **`args`** (*`tuple`*) – the argument tuple for the `func` invocation.
- **`kwargs`** (*`dict`*) – is a dictionary of keyword arguments for the `func` invocation.
- **`timeout`** (*`float, optional`*) – timeout in seconds for this remote call. If the creation of this `RRef` on worker `to` is not successfully processed on this worker within this timeout, then the next time there is an attempt to use the RRef (such as `to_here()`), a timeout will be raised indicating this failure. A value of 0 indicates an infinite timeout, i.e. a timeout error will never be raised. If not provided, the default value set during initialization or with `_set_rpc_timeout` is used.

Returns A user `RRef` instance to the result value. Use the blocking API `torch.distributed.rpc.RRef.to_here()` to retrieve the result value locally.

> **Warning:** Using GPU tensors as arguments or return values of `func` is not supported since we don't support sending GPU tensors over the wire. You need to explicitly copy GPU tensors to CPU before using them as arguments or return values of `func`.

> **Warning:** The `remote` API does not copy storages of argument tensors until sending them over the wire, which could be done by a different thread depending on the RPC backend type. The caller should make sure that the contents of those tensors stay intact until the returned RRef is confirmed by the owner, which can be checked using the `torch.distributed.rpc.RRef.confirmed_by_owner()` API.

> **Warning:** Errors such as timeouts for the `remote` API are handled on a best-effort basis. This means that when remote calls initiated by `remote` fail, such as with a timeout error, we take a best-effort approach to error handling. This means that errors are handled and set on the resulting RRef on an asynchronous basis. If the RRef has not been used by the application before this handling (such as `to_here` or fork call), then future uses of the `RRef` will appropriately raise errors. However, it is possible that the user application will use the `RRef` before the errors are handled. In this case, errors may not be raised as they have not yet been handled.

Example:: Make sure that `MASTER_ADDR` and `MASTER_PORT` are set properly API for more details. For example,

```
>>> export MASTER_ADDR=localhost
>>> export MASTER_PORT=5678
```

Then run the following code in two different processes:

```
>>> # On worker 0:
>>> import torch
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> rref1 = rpc.remote("worker1", torch.add, args=(torch.ones(2), 3))
>>> rref2 = rpc.remote("worker1", torch.add, args=(torch.ones(2), 1))
>>> x = rref1.to_here() + rref2.to_here()
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

Below is an example of running a TorchScript function using RPC.

```
>>> # On both workers:
>>> @torch.jit.script
>>> def my_script_add(t1, t2):
>>>    return torch.add(t1, t2)
>>> # On worker 0:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> rref = rpc.remote("worker1", my_script_add, args=(torch.ones(2), 3))
>>> rref.to_here()
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

rpc_async (*to*, *func*, *timeout=- 1*, *args=()*, *kwargs=None*)

Make a non-blocking RPC call to run function `func` on worker `to`. RPC messages are sent and received in parallel to execution of Python code. This method is thread-safe. This method will immediately return a `Future` that can be awaited on.

Parameters

- **to** (*str or WorkerInfo or int*) – name/rank/`WorkerInfo` of the destination worker.
- **func** (*callable*) – a callable function, such as Python callables, builtin operators (e.g. `add()`) and annotated TorchScript functions.
- **args** (*tuple*) – the argument tuple for the `func` invocation.
- **kwargs** (*dict*) – is a dictionary of keyword arguments for the `func` invocation.
- **timeout** (*float, optional*) – timeout in seconds to use for this RPC. If the RPC does not complete in this amount of time, an exception indicating it has timed out will be raised. A value of 0 indicates an infinite timeout, i.e. a timeout error will never be raised. If not provided, the default value set during initialization or with `_set_rpc_timeout` is used.

Returns Returns a `Future` object that can be waited on. When completed, the return value of `func` on `args` and `kwargs` can be retrieved from the `Future` object.

Warning: Using GPU tensors as arguments or return values of `func` is not supported since we don't support sending GPU tensors over the wire. You need to explicitly copy GPU tensors to CPU before using them as arguments or return values of `func`.

Warning: The `rpc_async` API does not copy storages of argument tensors until sending them over the wire, which could be done by a different thread depending on the RPC backend type. The caller should make sure that the contents of those tensors stay intact until the returned `Future` completes.

Example:: Make sure that `MASTER_ADDR` and `MASTER_PORT` are set properly API for more details. For example,

```
>>> export MASTER_ADDR=localhost
>>> export MASTER_PORT=5678
```

Then run the following code in two different processes:

```
>>> # On worker 0:
>>> import torch
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> fut1 = rpc.rpc_async("worker1", torch.add, args=(torch.ones(2), 3))
>>> fut2 = rpc.rpc_async("worker1", min, args=(1, 2))
>>> result = fut1.wait() + fut2.wait()
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

Below is an example of running a TorchScript function using RPC.

```
>>> # On both workers:
>>> @torch.jit.script
>>> def my_script_add(t1, t2):
>>>    return torch.add(t1, t2)
```

```
>>> # On worker 0:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> fut = rpc.rpc_async("worker1", my_script_add, args=(torch.ones(2), 3))
>>> ret = fut.wait()
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

rpc_sync (*to*, *func*, *timeout=- 1*, *args=()*, *kwargs=None*)

Make a blocking RPC call to run function `func` on worker `to`. RPC messages are sent and received in parallel to execution of Python code. This method is thread-safe.

Parameters

- **to** (*str or WorkerInfo or int*) – name/rank/`WorkerInfo` of the destination worker.
- **func** (*callable*) – a callable function, such as Python callables, builtin operators (e.g. `add()`) and annotated TorchScript functions.
- **args** (*tuple*) – the argument tuple for the `func` invocation.
- **kwargs** (*dict*) – is a dictionary of keyword arguments for the `func` invocation.
- **timeout** (*float, optional*) – timeout in seconds to use for this RPC. If the RPC does not complete in this amount of time, an exception indicating it has timed out will be raised. A value of 0 indicates an infinite timeout, i.e. a timeout error will never be raised. If not provided, the default value set during initialization or with `_set_rpc_timeout` is used.

Returns Returns the result of running `func` with `args` and `kwargs`.

> **Warning:** Using GPU tensors as arguments or return values of `func` is not supported since we don't support sending GPU tensors over the wire. You need to explicitly copy GPU tensors to CPU before using them as arguments or return values of `func`.

Example:: Make sure that `MASTER_ADDR` and `MASTER_PORT` are set properly API for more details. For example,

```
>>> export MASTER_ADDR=localhost
>>> export MASTER_PORT=5678
```

Then run the following code in two different processes:

```
>>> # On worker 0:
>>> import torch
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> ret = rpc.rpc_sync("worker1", torch.add, args=(torch.ones(2), 3))
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

Below is an example of running a TorchScript function using RPC.

```
>>> # On both workers:
>>> @torch.jit.script
>>> def my_script_add(t1, t2):
>>>     return torch.add(t1, t2)
```

```
>>> # On worker 0:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker0", rank=0, world_size=2)
>>> ret = rpc.rpc_sync("worker1", my_script_add, args=(torch.ones(2), 3))
>>> rpc.shutdown()
```

```
>>> # On worker 1:
>>> import torch.distributed.rpc as rpc
>>> rpc.init_rpc("worker1", rank=1, world_size=2)
>>> rpc.shutdown()
```

`size`()

Get the number of members in group.

`unpair`(**args*, ***kwargs*)

`machin.parallel.distributed.`**`World`**(**args*, ***kwargs*)

`machin.parallel.distributed.`**`get_world`**()

`machin.parallel.distributed.`**`get_cur_rank`**()

Returns Current real process rank.

`machin.parallel.distributed.`**`get_cur_name`**()

Returns Current real process name.

server

class `machin.parallel.server.`**`OrderedServerBase`**

Bases: `abc.ABC`

Descendent classes of OrderedServer does not have to guarantee strong consistency, that is, even if `OrderedServerBase.push_service`()` has returned True, there are possibilities that these acknowledged push are discarded.

abstract pull(*key*, *version=None*)

Pull a value with the specified `version` in `key`.

Parameters

- **key** – Key.
- **version** – Target version, if `None`, then the newest version of value of key will be pulled.

Returns `None` if version is not found, auto-deleted, or key is not found, otherwise returns value with the specified `version` in `key`, and then `version`

abstract push(*key*, *value*, *version*, *prev_version*)

Push a new `version` of `value` in `key` to the ordered server.

Note: If `version = prev_version` then there is no order guarantee. But you may exploit this feature.

Parameters

- **key** – Key.
- **value** – value.
- **version** – New version.
- **prev_version** – Previous version.

Returns `True` if success, and `False` if not.

class `machin.parallel.server.`**`OrderedServerSimple`**(*server_name*, *group*)

Bases: `machin.parallel.server.ordered_server.OrderedServerBase`

Parameters

- **server_name** (*str*) –
- **group** (*machin.parallel.distributed.world.RpcGroup*) –

pull(*key*, *version=None*)

Pull a value with the specified `version` in `key`.

Parameters

- **key** – Key.
- **version** – Target version, if `None`, then the newest version of value of key will be pulled.

Returns `None` if version is not found, auto-deleted, or key is not found, otherwise returns value with the specified `version` in `key`, and then `version`

push (*key*, *value*, *version*, *prev_version*)

Push a new `version` of `value` in `key` to the ordered server.

Note: If `version = prev_version` then there is no order guarantee. But you may exploit this feature.

Parameters

- **key** – Key.
- **value** – value.
- **version** – New version.
- **prev_version** – Previous version.

Returns `True` if success, and `False` if not.

class machin.parallel.server.**OrderedServerSimpleImpl** (*server_name*, *group*, *version_depth=1*, ***__*)

Bases: `object`

A simple key-value server, with strict ordered update

This init function must be only invoked on the runner process, and the runner process must be a member process of `group`.

Parameters

- **server_name** (*str*) – Name of this server, used to registered the server as a paired class of `group`.
- **group** (*machin.parallel.distributed.world.RpcGroup*) – Rpc group where server locates.
- **server_runner** – Name of the process serving the ordered server. By default is the first member of the group.
- **version_depth** (*int*) – Storage depth of old versions of the same key. If `depth = 1`, then only the newest version of the key will be saved.

class machin.parallel.server.**PushPullGradServer** (*server_name*, *group*, *model_name*, *secondary_reducers*, *o_server*)

Bases: `object`

Parameters

- **server_name** (*str*) –
- **group** (*machin.parallel.distributed.world.RpcGroup*) –
- **model_name** (*str*) –
- **secondary_reducers** (*List[str]*) –
- **o_server** (*machin.parallel.server.ordered_server.OrderedServerBase*) –

pull (*model*)

Pull the newest model. Its gradients will be cleared.

Parameters **`model`** (*`torch.nn.modules.module.Module`*) – Model to push.

`push`(*model*)

Push the gradients of your model, then pull the newest parameters. Its gradients will be cleared.

Parameters **`model`** (*`torch.nn.modules.module.Module`*) – Model to push.

class `machin.parallel.server.`**`PushPullGradServerImpl`**(*server_name*, *group*, *model_name='model'*, *primary_reducer=None*, *secondary_reducers=None*, *o_server=None*, *reduce_method='sum'*, *reduce_device='cpu'*, *reduce_batch_size=4*, *max_queue_size=64*)

Bases: `object`

A simple parameter server, which synchronize model parameters by pushing gradients and pulling back new parameters, no strict order is guaranteed.

Warning: `DistributedDataParallel` is not supported. since we cannot load state dictionary after creation.

Note: You should initialize `PushPullGradServer` on all members of `secondary_reducers`, and `primary_reducer`. Both of them should be members of the `group`.

Note: Internally the primary reducer will push updated versions to the ordered server.

Hint: Reduction is performed in a tree fashion:

1. In the first step, clients will push new gradients to a random secondary reducer, and the secondary reducer will perform the first reduction pass, then secondary reducers will push their results to the primary reducer.
2. In the second step, the primary reducer will reduce results from the secondary reducer to get the final reduced gradient dictionary (has the same structure as state_dict), and assign gradients to its **managed model**, and perform the optimization.
3. In the final step, the primary reducer will push the final model to the model server group, then clients can pull the newest model.

Parameters

- **`server_name`** (*`str`*) – Name of this server, used to registered the server as a paired class of `group`.
- **`group`** (*`machin.parallel.distributed.world.RpcGroup`*) – Server group.
- **`model_name`** (*`str`*) – Name of the managed model in the ordered server, only needed if `server` needs such a identifier. The default ordered server does not require this.

- **primary_reducer** (*str*) – Name of the process serving as the primary reducer, which collects reduced gradients from secondary reducers and perform the final reduction.
- **secondary_reducers** (*List[str]*) – Name of the process serving as secondary reducers.
- **o_server** (*machin.parallel.server.ordered_server.OrderedServerBase*) – Custom ordered server accessor. By default, the ordered server is a *OrderedServerSimple* hosted on the primary reducer.
- **reduce_method** (*str*) – "mean" or "sum"
- **reduce_device** (*Union[torch.device, str]*) – Device to perform reduction, by default it is "cpu".
- **reduce_batch_size** (*int*) – Size of a single reduction batch, server will wait until the number of requests in the reduction queue have reached this size.
- **max_queue_size** (*int*) – Maximum reduction request queue size.

manage_model (*model, optimizer*)

Let the main reducer manage your model. Must be called before start.

Warning: Make sure that the managed model is different from the model you use in your algorithms such as A3C!

Parameters

- **model** (*torch.nn.modules.module.Module*) – Model to manage.
- **optimizer** (>>>) – Optimizer of your model. you should initialize it first:
- **optimizer** –

Raises **RuntimeError if current rpc role is not the main reducer.** –

start ()

stop ()

watch ()

REDUCE_MASTER = 0

REDUCE_SLAVE = 1

class machin.parallel.server.**PushPullModelServer** (*model_name, o_server=None*)

Bases: `object`

Create an accessor to the services provided by *PushPullModelServerImpl*

Parameters

- **model_name** (*str*) – Name of the managed model in the ordered server, only needed if `server` needs such a identifier. The default ordered server does not require this.
- **o_server** (*machin.parallel.server.ordered_server.OrderedServerBase*) – Ordered server accessor.

pull (*model*)

Pull the newest state dict of your model and update its parameters and `pp_version`. Gradients will not be cleared.

Parameters **`model`** (*`torch.nn.modules.module.Module`*) – Model to pull.

`push` (*model*, *pull_on_fail=True*)

Try to push a model to the ordered server, if failed, the newest model will be automatically pulled and its parameters will be assigned to `model`. Gradients will not be cleared.

Parameters

- **`model`** (*`torch.nn.modules.module.Module`*) – Model to push.
- **`pull_on_fail`** – Pull the newest parameters if push failed.

class machin.parallel.server.**PushPullModelServerImpl** (*server_name*, *group*, *model_name='model'*, *o_server=None*)

Bases: `object`

A simple parameter server, which synchronize model parameters by pushing and pulling all parameters and maintaining a strict ordered version chain.

Warning: Only one model is supported.

This init function must be only invoked on the runner process, and the runner process must be a member process of `group`.

Parameters

- **`server_name`** (*`str`*) – Name of this server, used to registered the server as a paired class of `group`.
- **`group`** (*`machin.parallel.distributed.world.RpcGroup`*) – RpcGroup of the default server *`OrderedServerSimple`* mutually exclusive with `o_server`
- **`model_name`** (*`str`*) – Name of the managed model in the ordered server, only needed if `server` needs such a identifier. The default ordered server does not require this.
- **`o_server`** (*`machin.parallel.server.ordered_server.OrderedServerBase`*) – Custom ordered server accessor.

assigner

class machin.parallel.assigner.**ModelAssigner** (*models*, *model_connection*, *devices=None*, *model_size_multiplier=2*, *max_mem_ratio=0.5*, *cpu_weight=0*, *connection_weight=2*, *size_match_weight=0.01*, *complexity_match_weight=1*, *entropy_weight=1*, *iterations=500*, *update_rate=0.01*, *gpu_gpu_distance=1*, *cpu_gpu_distance=10*, *move_models=True*)

Bases: `object`

Assigner for pytorch modules.

Assign models to different devices. In the scope of a single process. Assigner assumes all GPUs have the **same processing power**.

Assignment is based on four aspects:

1. Distance and model connections. Connection is usually indicated by the amount of data transmitted between two models.
2. Compute complexity.
3. Model size.
4. Entropy.

Four aspects are controlled by four weights:

1. `connection_weight`, assigner will try to reduce the total `distance * connection` if this weight is larger.
2. `size_match_weight`, this weight controls the total memory space used on a single device, only works if total assigned memory of models exceeds allowed device memory size (internally it uses a relu activation), the larger, the tighter and more restricted the fit.
3. `complexity_match_weight`, this weights balance the model computation cost across devices, assigner will try to even the `computation cost / compute power` ratio for each device if this weight is larger.
4. `entropy_weight`, this weight minimize the uncertainty of model placement probability, so `model i` will have a close to 1 probability of locating on some `device j` if this weight is larger.

Assignment uses gradient descent to compute the probability matrix of each `model i` locating on each available `device j`.

See also:

ModelSizeEstimator

Note: When the sum of your model size is very close to the capacity of your device memory, *ModelAssigner* does not respond very well to the `size_match_weight`, therefore, please consider about increasing `model_size_multiplier` or decreasing `max_mem_ratio`.

Parameters

- **models** (*List[torch.nn.modules.module.Module]*) – Models to assign.
- **model_connection** (*Dict[Tuple[int, int], int]*) – Connection weight between modules. **Must be positive**
- **devices** (*List[Union[torch.device, str]]*) – Available devices.
- **model_size_multiplier** – Size multiplier of models, used to reserve enough space for models,
- **max_mem_ratio** – Maximum percent of memory allowed.
- **cpu_weight** – Weight of cpu. Relative to the computing power of one GPU. By default it is 0 so no computation will be performed on CPU. **Must be positive**
- **connection_weight** – Weight of connection between models.
- **size_match_weight** – Weight of size match.
- **complexity_match_weight** – Weight of complexity match.
- **entropy_weight** – Weight of entropy.
- **iterations** – Number of optimization iterations.
- **update_rate** – Learning rate of the adam optimizer.

- **gpu_gpu_distance** – Estimated distance cost between gpu-gpu. **Must be positive**
- **cpu_gpu_distance** – Estimated distance cost between cpu-gpu. **Must be positive**
- **move_models** – Whether to automatically move the models after assignment.

static optimize_placement(*optimizer*, *placement*, *model_size*, *size_capacity*, *model_complexity*, *complexity_capacity*, *model_connection*, *device_distance*, *connection_weight*, *size_match_weight*, *complexity_match_weight*, *entropy_weight*)

Suppose there are n models to place and m devices available.

Parameters

- **optimizer** – optimizer of placement
- **placement** (*torch.Tensor*) – shape `[n, m]`
- **model_size** (*torch.Tensor*) – shape `[1, n]`
- **size_capacity** (*torch.Tensor*) – shape `[1, m]`
- **model_complexity** (*torch.Tensor*) – shape `[1, n]`
- **complexity_capacity** (*torch.Tensor*) – shape `[1, m]`
- **model_connection** (*torch.Tensor*) – shape `[n, n]`
- **device_distance** (*torch.Tensor*) – shape `[m, m]`
- **connection_weight** (*float*) – Weight of connection between models.
- **size_match_weight** (*float*) – Weight of size match.
- **complexity_match_weight** (*float*) – Weight of complexity match.
- **entropy_weight** (*float*) – weight of entropy.

property assignment

List[t.device]: Assigned devices for each model in your model list.

class machin.parallel.assigner.**ModelSizeEstimator**(*model*, *size_multiplier=2*)

Bases: `object`

Size estimator for pytorch modules.

Estimates the size of PyTorch models in memory.

Note: This estimator can only estimate the total size of parameters and buffers. Therefore we need to multiply the raw estimated size with a correction coefficient to reserve enough space for models.

Parameters

- **model** (*torch.nn.modules.module.Module*) – Model to be estimated.
- **size_multiplier** – Model estimated size will be multiplied with this value, to ensure enough space will be reserved to contain your model and inputs.

estimate_size()

Estimate model size in memory in megabytes.

get_buffer_sizes()

Get sizes of all buffers in `model` in mega bytes.

Return type float

get_parameter_sizes()
Get sizes of all parameters in `model` in mega bytes.

Return type float

pickle

class machin.parallel.pickle.**Pickler**(*file*, *recurse=False*, *copy_tensor=False*)
Bases: `dill._dill.Pickler`

Note: Picklers shares ".dispatch" among instances, and owns "dispatch_table" per instance.

The base Pickler (not dill, from builtin pickle library), will first look up the default dump method in ".dispatch", if no valid method is found, it will try to find a custom dump method in ".dispatch_table".

This takes a binary file for writing a pickle data stream.

The optional *protocol* argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3 and 4. The default protocol is 3; a backward-incompatible protocol designed for Python 3.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

The *file* argument must have a write() method that accepts a single bytes argument. It can thus be a file object opened for binary writing, an io.BytesIO instance, or any other custom object that meets this interface.

If *fix_imports* is True and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

machin.parallel.pickle.**dumps**(*obj*, *recurse=False*, *copy_tensor=True*)
Convert objects to bytes. Works for cpu and gpu tensors.

Warning: Till pytorch 1.5.0, there is a bug for referenced gpu tensors, which would require users to keep shared gpu tensors during the whole process life and not reassigning / deleting them, however, you may refill them with different values.

See here

Parameters

- **obj** – Object to dump.
- **recurse** – Enable recursive dumping, enable this to dump local functions and lambdas.
- **copy_tensor** – Whether to dump tensors, storage as a full copy. If it is set to "False", then dumped tensors must either locate on GPUs or in shared memory.

Returns Bytes.

machin.parallel.pickle.**mark_static_module**(*module*)
Some modules are **static**, which means they are stateless and will remain the same whether you import it in process A or process B.

If your module contains reference to functions, objects or anything inside a CDLL (usually the reference is a pointer), it is not picklable by dill, and will cause nasty errors, however, by marking this module as "Static", dill will recognize this module as a builtin module and not saving the states of this module, dill will only save a reference to it in this situation.

Parameters **module** (*Any*) – Some module which imports CDLLs by hand and not using pybind11.

pool

class machin.parallel.pool.**CtxPool**(*processes*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *worker_contexts=None*, *is_recursive=False*, *is_daemon=True*, *is_copy_tensor=True*, *share_method=None*)

Bases: *machin.parallel.pool.Pool*

Pool with context for each worker. your function must accept a `ctx` object as your first non-keyword argument.

If `worker_contexts` is not specified, then `ctx` will be `None`.

The length of `worker_contexts` must be the same as `processes`

Note: To share "cpu" tensors in shared memory, you must set:

```
is_copy_tensor=False,
share_method="cpu"
```

To share "cuda" tensors, you must set:

```
is_copy_tensor=False,
share_method="cuda"
```

Note: The default context used in pool is "spawn", to avoid any issues brought by "fork". "fork" will only be used if you want to pass cpu tensors in shared memory.

Parameters

- **processes** (*int*) – Number of processes in the pool.
- **initializer** – Initializer function executed by the pool/
- **initargs** – Args passed to the init function.
- **maxtasksperchild** – Maximum number of tasks per worker process.
- **is_recursive** – Set to `True` to support local functions and lambdas.
- **is_daemon** – Whether worker processes in the pool are started as daemon processes.
- **is_copy_tensor** – Whether to copy tensors or pass tensors by reference to worker processes.
- **share_method** – If `is_copy_tensor` is `False`, you must specify this argument. "cpu" means you may use cpu tensors in the shared memory, "cuda" means cuda tensors, you can only specify one share method.

class machin.parallel.pool.**CtxPoolStorage**

Bases: `object`

This storage class is used by all *CtxPool* instances. However, since for each worker process, they have different memory spaces, `storage` is unique for all workers.

`storage` is accessed on the client process side.

storage = None

class machin.parallel.pool.**CtxThreadPool** (*processes*, *initializer=None*, *initargs=()*, *worker_contexts=None*)

Bases: *machin.parallel.pool.ThreadPool*

Parameters **processes** (*int*) –

apply (*func*, *args=()*, *kwds=None*)
Equivalent of *func(*args, **kwds)*. Pool must be running.

apply_async (*func*, *args=()*, *kwds=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *apply()* method.

imap (*func*, *iterable*, *chunksize=1*)
Equivalent of *map()* – can be MUCH slower than *Pool.map()*.

imap_unordered (*func*, *iterable*, *chunksize=1*)
Like *imap()* method but ordering of results is arbitrary.

map (*func*, *iterable*, *chunksize=None*)
Apply *func* to each element in *iterable*, collecting the results in a list that is returned.

map_async (*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *map()* method.

starmap (*func*, *iterable*, *chunksize=None*)
Like *map()* method but the elements of the *iterable* are expected to be iterables as well and will be unpacked as arguments. Hence *func* and (a, b) becomes func(a, b).

starmap_async (*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *starmap()* method.

class machin.parallel.pool.**P2PPool** (*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *is_recursive=False*, *is_daemon=True*, *is_copy_tensor=True*, *share_method=None*)

Bases: *machin.parallel.pool.Pool*

Note: To share "cpu" tensors in shared memory, you must set:

```
is_copy_tensor=False,
share_method="cpu"
```

To share "cuda" tensors, you must set:

```
is_copy_tensor=False,
share_method="cuda"
```

Note: The default context used in pool is "spawn", to avoid any issues brought by "fork". "fork" will only be used if you want to pass cpu tensors in shared memory.

Parameters

- **`processes`** – Number of processes in the pool.
- **`initializer`** – Initializer function executed by the pool/
- **`initargs`** – Args passed to the init function.
- **`maxtasksperchild`** – Maximum number of tasks per worker process.
- **`is_recursive`** – Set to `True` to support local functions and lambdas.
- **`is_daemon`** – Whether worker processes in the pool are started as daemon processes.
- **`is_copy_tensor`** – Whether to copy tensors or pass tensors by reference to worker processes.
- **`share_method`** – If `is_copy_tensor` is `False`, you must specify this argument. "cpu" means you may use cpu tensors in the shared memory, "cuda" means cuda tensors, you can only specify one share method.

`close``()`

`class` `machin.parallel.pool.`**`Pool`**`(`*processes=None, initializer=None, initargs=(), maxtasksperchild=None, is_recursive=False, is_daemon=True, is_copy_tensor=True, share_method=None*`)`

Bases: `multiprocessing.pool.Pool`

Enhanced multiprocessing pool for pytorch, provides:

1. Support for lambdas and local functions.
2. Ability to select the tensor serialize scheme.

Note: To share "cpu" tensors in shared memory, you must set:

```
is_copy_tensor=False,
share_method="cpu"
```

To share "cuda" tensors, you must set:

```
is_copy_tensor=False,
share_method="cuda"
```

Note: The default context used in pool is "spawn", to avoid any issues brought by "fork". "fork" will only be used if you want to pass cpu tensors in shared memory.

Parameters

- **`processes`** – Number of processes in the pool.
- **`initializer`** – Initializer function executed by the pool/
- **`initargs`** – Args passed to the init function.
- **`maxtasksperchild`** – Maximum number of tasks per worker process.
- **`is_recursive`** – Set to `True` to support local functions and lambdas.
- **`is_daemon`** – Whether worker processes in the pool are started as daemon processes.

- **is_copy_tensor** – Whether to copy tensors or pass tensors by reference to worker processes.
- **share_method** – If `is_copy_tensor` is `False`, you must specify this argument. "cpu" means you may use cpu tensors in the shared memory, "cuda" means cuda tensors, you can only specify one share method.

apply (*func*, *args=()*, *kwds=None*)
Equivalent of *func(*args, **kwds)*. Pool must be running.

apply_async (*func*, *args=()*, *kwds=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *apply()* method.

imap (*func*, *iterable*, *chunksize=1*)
Equivalent of *map()* – can be MUCH slower than *Pool.map()*.

imap_unordered (*func*, *iterable*, *chunksize=1*)
Like *imap()* method but ordering of results is arbitrary.

map (*func*, *iterable*, *chunksize=None*)
Apply *func* to each element in *iterable*, collecting the results in a list that is returned.

map_async (*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *map()* method.

size ()

Returns The number of workers in pool.

starmap (*func*, *iterable*, *chunksize=None*)
Like *map()* method but the elements of the *iterable* are expected to be iterables as well and will be unpacked as arguments. Hence *func* and (a, b) becomes func(a, b).

starmap_async (*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *starmap()* method.

class machin.parallel.pool.**ThreadPool** (*processes=None*, *initializer=None*, *initargs=()*)
Bases: `multiprocessing.pool.ThreadPool`

A typical thread pool.

size ()

Returns The number of workers in pool.

machin.parallel.pool.**proxy_caller** (**input_*)
Call a serialized function and return results.

machin.parallel.pool.**proxy_ctx_caller** (**input_*)
Call a serialized function with worker context and return results.

machin.parallel.pool.**proxy_dumper** (*recurse*, *copy_tensor*, *func*, *args_list*)
Serialize a function so it can be called.

Returns List[function string, arguments…]

queue

class `machin.parallel.queue.`**`SimpleQueue`**(*, *ctx=None*, *copy_tensor=False*)

Bases: `object`

A simple single direction queue for inter-process communications. There could be multiple receivers and multiple senders on each side.

Parameters

- **`ctx`** – Multiprocessing context, you can get this using `get_context`
- **`copy_tensor`** – Set the queue to send a fully serialized tensor if `True`, and only a stub of reference if `False`.

See also:

`dump_tensor()`

`close`()

`empty`()

Returns Whether the queue is empty or not.

`get`(*timeout=None*)

Get an object from the queue. This api is required by `multiprocessing.pool` to perform inter-process communication.

Note: This api is used by sub-processes in pool to get tasks and work.

Returns Any object.

`put`(*obj*)

Put an object into the queue. This api is required by `multiprocessing.pool` to perform inter-process communication.

Note: This api is used by sub-processes in pool to put results and respond.

Parameters **`obj`** (*`Any`*) – Any object.

`quick_get`(*timeout=None*)

Get an object from the queue.

Note: this api is used by the result manager (`Pool._result_handler`) thread to get results from the queue, since it is single threaded, there is no need to use locks, and therefore quicker.

Returns Any object.

`quick_put`(*obj*)

Put an object into the queue.

Note: this api is used by the pool manager (`Pool._task_handler`) thread to put tasks into the queue, since it is single threaded, there is no need to use locks, and therefore quicker.

Parameters **`obj`** (*`Any`*) – Any object.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX