天授

发布 *0.4.1*

天授项目贡献者

2021 年 04 月 15 日

教程

天授 是一个基于 PyTorch 的深度强化学习平台，目前实现的算法有：

- DQN `DQNPolicy` Deep Q-Network
- 双网络 DQN `DQNPolicy` Double DQN
- 值分布 DQN `C51Policy` Categorical DQN
- 分位数回归 DQN `QRDQNPolicy` Quantile Regression DQN
- 策略梯度 `PGPolicy` Policy Gradient
- 优势动作评价 `A2CPolicy` Advantage Actor-Critic
- 信任区域策略优化 `TRPOPolicy` Trust Region Policy Optimization
- 近端策略优化 `PPOPolicy` Proximal Policy Optimization
- 深度确定性策略梯度 `DDPGPolicy` Deep Deterministic Policy Gradient
- 双延迟深度确定性策略梯度 `TD3Policy` Twin Delayed DDPG
- 软动作评价 `SACPolicy` Soft Actor-Critic
- 离散软动作评价 `DiscreteSACPolicy` Discrete Soft Actor-Critic
- 模仿学习 `ImitationPolicy` Imitation Learning
- `DiscreteBCQPolicy` Discrete Batch-Constrained deep Q-Learning
- 后验采样强化学习 `PSRLPolicy` Posterior Sampling Reinforcement Learning
- 优先级经验重放 `PrioritizedReplayBuffer` Prioritized Experience Replay
- 广义优势函数估计器 `compute_episodic_return()` Generalized Advantage Estimator

天授还有如下特点：

- 实现优雅，使用 3000 多行代码即完全实现上述功能
- 目前为止实现效果最好的 MuJoCo benchmark
- 支持任意算法的多个环境（同步异步均可的）并行采样，详见环境并行采样
- 支持动作网络和价值网络使用循环神经网络（RNN）来实现，详见*RNN 训练*
- 支持自定义环境，包括任意类型的观测值和动作值（比如一个字典、一个自定义的类），详见自定义环境与状态表示
- 支持自定义训练策略，详见定制化训练策略
- 支持 N-step bootstrap 采样方式 `compute_nstep_return()` 和优先级经验重放 `PrioritizedReplayBuffer` 在任意基于 Q 学习的算法上的应用；感谢 numba jit 的优化让 GAE、nstep 和 PER 运行速度变得巨快无比
- 支持多智能体学习，详见多智能体强化学习
- 拥有全面的 单元测试，包括功能测试、完整训练流程测试、文档测试、代码风格测试和类型测试

与英文文档不同，中文文档提供了一个宏观层面的对天授平台的概览。（其实都是 毕业论文 里面弄出来的）

CHAPTER 1

安装

天授目前发布在 PyPI 和 conda-forge 中，需要 Python 版本 3.6 以上。

通过 PyPI 进行安装：

```
$ pip install tianshou
```

通过 conda 进行安装：

```
$ conda -c conda-forge install tianshou
```

还可以直接从 GitHub 源代码最新版本进行安装：

```
$ pip install git+https://github.com/thu-ml/tianshou.git@master --upgrade
```

在安装完毕后，打开您的 Python 并输入

```
import tianshou
print(tianshou.__version__)
```

如果没有异常出现，那么说明已经成功安装了。

1.1 Slide

[Invited Talk] 2020/10/11 强化学习、对抗学习及博弈，第三届混合增强智能前沿讲习班，PDF

1.2 Deep Q Network

深度强化学习在很多应用场景中表现惊艳，比如 DQN [[MKS+15]] 就是一个很好的例子，在 Atari 游戏中一鸣惊人。在本教程中，我们会逐步展示如何在 Cartpole 任务上使用天授训练一个 DQN 智能体。完整的代码位于 test/discrete/test_dqn.py。

与现有深度强化学习平台（比如 RLlib）不同，它们将超参数、网络结构等弄成一个 config.yaml。天授从代码层面上提供了一个简洁的搭建方式。

1.2.1 创建环境

首先创建与智能体交互的环境。环境接口遵循 OpenAI Gym。运行如下命令：

```
import gym
import tianshou as ts

env = gym.make('CartPole-v0')
```

CartPole-v0 是一个很简单的离散动作空间场景，DQN 也是为了解决这种任务。在使用不同种类的强化学习算法前，您需要了解每个算法是否能够应用在离散动作空间场景 / 连续动作空间场景中，比如像 DDPG [[LHP+16]] 就只能用在连续动作空间任务中，其他基于策略梯度的算法可以用在任意这两个场景中。

1.2.2 并行环境装饰器

此处定义训练环境和测试环境。使用原来的 gym.Env 当然是可以的：

```
train_envs = gym.make('CartPole-v0')
test_envs = gym.make('CartPole-v0')
```

天授提供了向量化环境装饰器，比如 VectorEnv、SubprocVectorEnv 和 RayVectorEnv。可以像下面这样使用：

```
train_envs = ts.env.DummyVectorEnv([lambda: gym.make('CartPole-v0') for _ in␣
↪range(8)])
test_envs = ts.env.DummyVectorEnv([lambda: gym.make('CartPole-v0') for _ in␣
↪range(100)])
```

此处在 train_envs 建立了 8 个环境，在 test_envs 建立了 100 个环境。接下来为了展示需要，使用后面那块代码。

1.2.3 建立网络

天授支持任意的用户定义的网络和优化器，但是需要遵循既定 API，比如像下面这样：

```
import torch, numpy as np
from torch import nn

class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(*[
            nn.Linear(np.prod(state_shape), 128), nn.ReLU(inplace=True),
```

(下页继续)

(续上页)

```
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape))
        ])
    def forward(self, obs, state=None, info={}):
        if not isinstance(obs, torch.Tensor):
            obs = torch.tensor(obs, dtype=torch.float)
        batch = obs.shape[0]
        logits = self.model(obs.view(batch, -1))
        return logits, state

state_shape = env.observation_space.shape or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=1e-3)
```

定义网络的规则如下：

- 输入
  - obs，观测值，为 numpy.ndarray、torch.Tensor、或者自定义的类、或者字典
  - state，隐藏状态表示，为 RNN 使用，可以为字典或者 numpy.ndarray 或者 torch.Tensor
  - info，环境信息，由环境提供，是一个字典
- 输出
  - logits：网络的原始输出，会被用来计算 Policy，比如输出 Q 值，然后在 DQNPolicy 中后续会进一步计算 $\arg\max_a Q(s, a)$；又比如 PPO [[SWD+17]] 算法中，如果使用对角高斯策略，则 logits 为 (mu, sigma)
  - state：下一个隐藏状态，还是为了 RNN

一些已经定义好并已经内置的 MLP 网络可以在 tianshou.utils.net.common、tianshou.utils.net.discrete 和 tianshou.utils.net.continuous 中找到。

1.2.4 初始化策略

我们使用上述代码中定义的 net 和 optim，以及其他超参数，来定义一个策略。此处定义了一个有目标网络（Target Network）的 DQN 策略：

```
policy = ts.policy.DQNPolicy(net, optim, discount_factor=0.9, estimation_step=3,␣
↪target_update_freq=320)
```

1.2.5 定义采集器

采集器（Collector）是天授中的一个关键概念。它定义了策略与不同环境交互的逻辑。在每一回合（step）中，采集器会让策略与环境交互指定数目（至少）的步数或者轮数，并且会将产生的数据存储在重放缓冲区中。

```
train_collector = ts.data.Collector(policy, train_envs, ts.data.
↪VectorReplayBuffer(20000, 10), exploration_noise=True)
test_collector = ts.data.Collector(policy, test_envs, exploration_noise=True)
```

1.2.6 使用训练器训练策略

天授提供了两种类型的训练器，onpolicy_trainer 和 offpolicy_trainer，分别对应同策略学习和异策略学习。训练器会在 stop_fn 达到条件的时候停止训练。由于 DQN 是一种异策略算法，因此使用 offpolicy_trainer 进行训练：

```
result = ts.trainer.offpolicy_trainer(
    policy, train_collector, test_collector,
    max_epoch=10, step_per_epoch=10000, step_per_collect=10,
    update_per_step=0.1, episode_per_test=100, batch_size=64,
    train_fn=lambda epoch, env_step: policy.set_eps(0.1),
    test_fn=lambda epoch, env_step: policy.set_eps(0.05),
    stop_fn=lambda mean_rewards: mean_rewards >= env.spec.reward_threshold)
print(f'Finished training! Use {result["duration"]}')
```

每个参数的具体含义如下：

- max_epoch：最大允许的训练轮数，有可能没训练完这么多轮就会停止（因为满足了 stop_fn 的条件）
- step_per_epoch：每个 epoch 要更新多少次策略网络
- step_per_collect：每次更新前要收集多少帧与环境的交互数据。上面的代码参数意思是，每收集 10 帧进行一次网络更新
- episode_per_test：每次测试的时候花几个 rollout 进行测试
- batch_size：每次策略计算的时候批量处理多少数据
- train_fn：在每个 epoch 训练之前被调用的函数，输入的是当前第几轮 epoch 和当前用于训练的 env 一共 step 了多少次。上面的代码意味着，在每次训练前将 epsilon 设置成 0.1
- test_fn：在每个 epoch 测试之前被调用的函数，输入的是当前第几轮 epoch 和当前用于训练的 env 一共 step 了多少次。上面的代码意味着，在每次测试前将 epsilon 设置成 0.05
- stop_fn：停止条件，输入是当前平均总奖励回报（the average undiscounted returns），返回是否要停止训练
- logger：天授支持 TensorBoard，可以像下面这样初始化：

```
from torch.utils.tensorboard import SummaryWriter
from tianshou.utils import BasicLogger
writer = SummaryWriter('log/dqn')
logger = BasicLogger(writer)
```

把 logger 送进去，训练器会自动把训练日志记录在里面。

训练器返回的结果是个字典，如下所示：

```
{
    'train_step': 9246,
    'train_episode': 504.0,
    'train_time/collector': '0.65s',
    'train_time/model': '1.97s',
    'train_speed': '3518.79 step/s',
    'test_step': 49112,
    'test_episode': 400.0,
    'test_time': '1.38s',
    'test_speed': '35600.52 step/s',
    'best_reward': 199.03,
```

(下页继续)

(续上页)

```
    'duration': '4.01s'
}
```

可以看出大概 4 秒就在 CartPole 任务上训练出来一个 DQN 智能体，在 100 轮测试中平均总奖励回报为 199.03。

1.2.7 存储、导入策略

因为策略继承了 `torch.nn.Module`，所以存储和导入策略和 PyTorch 中的网络并无差别，如下所示：

```
torch.save(policy.state_dict(), 'dqn.pth')
policy.load_state_dict(torch.load('dqn.pth'))
```

1.2.8 可视化智能体的表现

采集器 `Collector` 支持渲染智能体的表现。下面的代码展示了以 35FPS 的帧率查看智能体表现：

```
policy.eval()
policy.set_eps(0.05)
collector = ts.data.Collector(policy, env, exploration_noise=True)
collector.collect(n_episode=1, render=1 / 35)
```

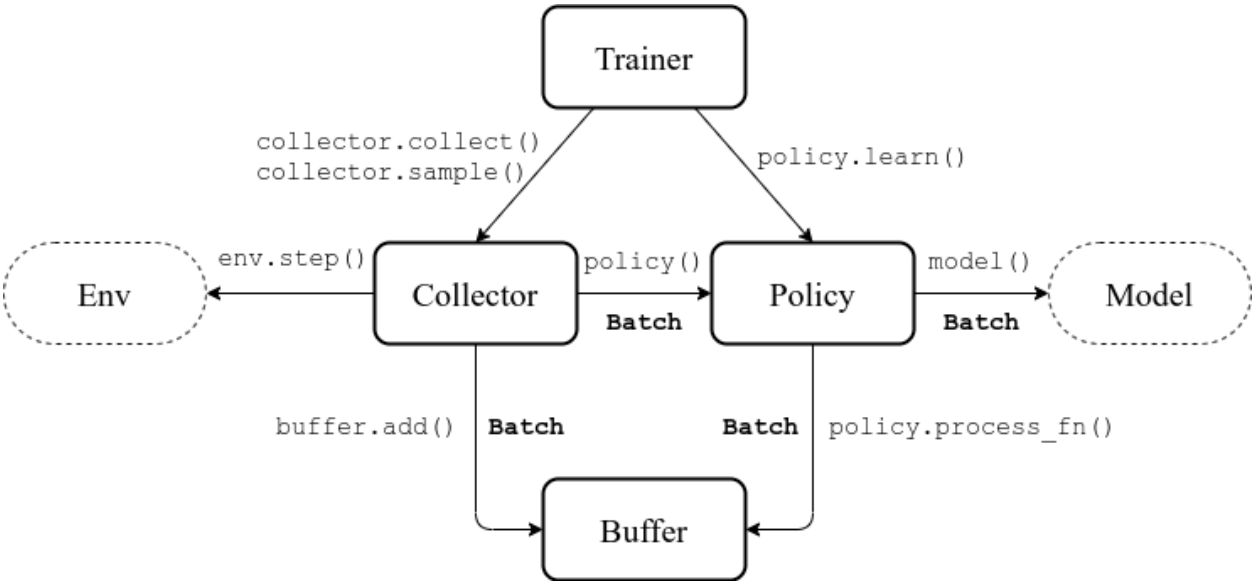
1.2.9 定制化训练器

天授为了能够支持用户的定制化训练器，在 Trainer 做了尽可能少的封装。使用者可以自由地编写自己所需要的训练策略，比如：

```
# 在正式训练前先收集 5000 帧数据
train_collector.collect(n_step=5000, random=True)

policy.set_eps(0.1)
for i in range(int(1e6)):  # 训练总数
    collect_result = train_collector.collect(n_step=10)

    # 如果收集的 episode 平均总奖励回报超过了阈值，或者每隔 1000 步，
    # 就会对 policy 进行测试
    if collect_result['rews'].mean() >= env.spec.reward_threshold or i % 1000 == 0:
        policy.set_eps(0.05)
        result = test_collector.collect(n_episode=100)
        if result['rews'].mean() >= env.spec.reward_threshold:
            print(f'Finished training! Test mean returns: {result["rews"].mean()}')
            break
        else:
            # 重新设置 eps 为 0.1，表示训练策略
            policy.set_eps(0.1)

    # 使用采样出的数据组进行策略训练
    losses = policy.update(64, train_collector.buffer)
```

1.3 基本概念

天授把一个 RL 训练流程划分成了几个子模块：trainer（负责训练逻辑）、collector（负责数据采集）、policy（负责训练策略）和 buffer（负责数据存储），此外还有两个外围的模块，一个是 env，一个是 model（policy 负责 RL 算法实现比如 loss function 的计算，model 就只是个正常的神经网络）。下图描述了这些模块的依赖：



1.3.1 Batch

天授提供了 Batch 作为内部模块传递数据所使用的数据结构，它既像字典又像数组，可以以这两种方式组织数据和访问数据，像下面这样：

```
>>> import torch, numpy as np
>>> from tianshou.data import Batch
>>> data = Batch(a=4, b=[5, 5], c='2312312', d=('a', -2, -3))
>>> # 注意，list 会自动变成 numpy
>>> data.b
array([5, 5])
>>> data.b = np.array([3, 4, 5])
>>> print(data)
Batch(
    a: 4,
    b: array([3, 4, 5]),
    c: '2312312',
    d: array(['a', '-2', '-3'], dtype=object),
)
>>> data = Batch(obs={'index': np.zeros((2, 3))}, act=torch.zeros((2, 2)))
>>> data[:, 1] += 6
>>> print(data[-1])
Batch(
    obs: Batch(
             index: array([0., 6., 0.]),
         ),
    act: tensor([0., 6.]),
)
```

总之就是可以定义任何 key-value 放在 Batch 里面，然后可以做一些常规的操作比如 +-*、cat/stack 之类的。Understand Batch 里面详细描述了 Batch 的各种用法，非常值得一看（虽然它是英文的但只要看代码也还行?）。

1.3.2 Buffer

`ReplayBuffer` 负责存储数据和采样出来数据用于 policy 的训练。目前天授保留了 7 个关键字在 Buffer 里面：

- `obs` t 时刻的观测值；
- `act` t 时刻采取的动作值；
- `rew` t 时刻环境返回的奖励函数值；
- `done` t 时刻是否结束这个 episode；
- `obs_next` $t+1$ 时刻的观测值；
- `info` t 时刻环境给出的额外信息（gym.Env 会返回 4 个东西，最后一个就是它）；
- `policy` t 时刻由 policy 计算出的需要额外存储的数据；

下面的代码片段展示了 Buffer 的一些典型用法：

```
>>> import pickle, numpy as np
>>> from tianshou.data import Batch, ReplayBuffer
>>> buf = ReplayBuffer(size=20)
>>> for i in range(3):
...     buf.add(Batch(obs=i, act=i, rew=i, done=0, obs_next=i + 1, info={}))

>>> buf.obs  # 因为设置了 size = 20, 所以 len(buf.obs) == 20
array([0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> # 但是里面只有 3 个合法的数据, 因此 len(buf) == 3
>>> len(buf)
3
>>> pickle.dump(buf, open('buf.pkl', 'wb'))  # 把 buffer 所有数据保存到 "buf.pkl"
>>> buf.save_hdf5('buf.hdf5')  # 把 buffer 所有数据保存到 "buf.hdf5"

>>> buf2 = ReplayBuffer(size=10)
>>> for i in range(15):
...     done = i % 4 == 0
...     buf2.add(Batch(obs=i, act=i, rew=i, done=done, obs_next=i + 1, info={}))
>>> len(buf2)
10
>>> buf2.obs  # 因为 buf2 的 size = 10, 所以它只会存储最后 10 步的结果
array([10, 11, 12, 13, 14,  5,  6,  7,  8,  9])

>>> buf.update(buf2)  # 把 buf2 的数据挪到 buf 里面, 同时保持相对时间顺序
>>> buf.obs
array([ 0,  1,  2,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,  0,  0,  0,
        0,  0,  0,  0])

>>> indice = buf.sample_index(0)  # 使用 batchsize=0 来获取 buffer 里面的全部数据
>>> indice
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> buf.prev(indice)  # 给定 index, 计算上一个 transition 所对应的 index
array([ 0,  0,  1,  2,  3,  4,  5,  7,  7,  8,  9, 11, 11])
>>> buf.next(indice)  # 给定 index, 计算下一个 transition 所对应的 index
array([ 1,  2,  3,  4,  5,  6,  6,  8,  9, 10, 10, 12, 12])
```

(下页继续)

(续上页)

```
>>> # 从 buffer 里面拿一个随机的数据, batch_data 就是 buf[indice]
>>> batch_data, indice = buf.sample(batch_size=4)
>>> batch_data.obs == buf[indice].obs
array([ True,  True,  True,  True])
>>> len(buf)
13

>>> buf = pickle.load(open('buf.pkl', 'rb'))  # 从"buf.pkl" 文件恢复出 buffer
>>> len(buf)
3
>>> buf = ReplayBuffer.load_hdf5('buf.hdf5')  # 从"buf.hdf5" 导入完整的 buffer
>>> len(buf)
3
```

ReplayBuffer 还支持堆叠采样（为了 RNN，详情查看 Issue 19）、不存储 obs_next（为了省些内存），以及任意类型的数据结构存储（这个是 Batch 支持的）：

```
>>> buf = ReplayBuffer(size=9, stack_num=4, ignore_obs_next=True)
>>> for i in range(16):
...     done = i % 5 == 0
...     ptr, ep_rew, ep_len, ep_idx = buf.add(
...         Batch(obs={'id': i}, act=i, rew=i,
...               done=done, obs_next={'id': i + 1}))
...     print(i, ep_len, ep_rew)
0 [1] [0.]
1 [0] [0.]
2 [0] [0.]
3 [0] [0.]
4 [0] [0.]
5 [5] [15.]
6 [0] [0.]
7 [0] [0.]
8 [0] [0.]
9 [0] [0.]
10 [5] [40.]
11 [0] [0.]
12 [0] [0.]
13 [0] [0.]
14 [0] [0.]
15 [5] [65.]
>>> print(buf)  # 可以发现 obs_next 并不在里面存着
ReplayBuffer(
    obs: Batch(
             id: array([ 9, 10, 11, 12, 13, 14, 15,  7,  8]),
         ),
    act: array([ 9, 10, 11, 12, 13, 14, 15,  7,  8]),
    rew: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
    done: array([False, True, False, False, False, False, True, False,
                 False]),
)
>>> index = np.arange(len(buf))
>>> print(buf.get(index, 'obs').id)
[[ 7  7  8  9]
 [ 7  8  9 10]
 [11 11 11 11]]
```

(下页继续)

(续上页)

```
 [11 11 11 12]
 [11 11 12 13]
 [11 12 13 14]
 [12 13 14 15]
 [ 7  7  7  7]
 [ 7  7  7  8]]
>>> # 也可以这样取出 stacked 过的 obs (注意 stack 只对 obs/obs_next/info/policy 有效)
>>> abs(buf.get(index, 'obs')['id'] - buf[index].obs.id).sum().sum()
0
>>> # 可以通过 __getitem__ 来弄出 obs_next (虽然并没存), 但是 [:] 会按照时间顺序 (而不是实际存储顺
序) 来取数据
>>> # 比如下面这个就相当于 index == [7, 8, 0, 1, 2, 3, 4, 5, 6]
>>> print(buf[:].obs_next.id)
[[ 7  7  7  8]
 [ 7  7  8  9]
 [ 7  8  9 10]
 [ 7  8  9 10]
 [11 11 11 12]
 [11 11 12 13]
 [11 12 13 14]
 [12 13 14 15]
 [12 13 14 15]]
>>> full_index = np.array([7, 8, 0, 1, 2, 3, 4, 5, 6])
>>> np.allclose(buf[:].obs_next.id, buf[full_index].obs_next.id)
True
```

天授还提供了其他类型的 buffer 比如 `PrioritizedReplayBuffer`(基于线段树)、`VectorReplayBuffer`(能够向其中添加不同 episode 的数据的同时维护时间顺序)。可以访问对应的文档来查看。

1.3.3 Policy

天授把一个 RL 算法用一个继承自 `BasePolicy` 的类来实现，主要的部分有如下几个：

- `__init__()`：策略初始化，比如初始化自定义的模型等；
- `forward()`：根据给定的观测值 obs，计算出动作值 action；
- `process_fn()`：在获取训练数据之前和 buffer 进行交互，比如使用 GAE 或者 nstep 算法来估计优势函数；
- `learn()`：使用一个 Batch 的数据进行策略的更新；
- `post_process_fn()`：使用一个 Batch 的数据进行 Buffer 的更新（比如更新 PER）；
- `update()`：最主要的接口。这个 update 函数先是从 buffer 采样出一个 batch，然后调用 process_fn 预处理，然后 learn 更新策略，然后 post_process_fn 完成一次迭代：`process_fn -> learn -> post_process_fn`。

各种状态和阶段

强化学习训练流程可以分为两个部分：训练部分（Training state）和测试部分（Testing State），而训练部分可以细分为采集数据阶段（Collecting state）和更新策略阶段（Updating state），两个阶段在训练过程中交替进行。顾名思义，采集数据阶段是由 collector 负责的，而策略更新阶段是由 policy.update 负责的。

为了区分上述这些状态，可以通过检查 `policy.training` 和 `policy.updating` 来确定处于哪个状态，这边列了一张表方便查看：

State for policy		policy.training	policy.updating
Training state	Collecting state	True	False
	Updating state	True	True
Testing state		False	False

`policy.updating` 实际情况下主要用于 exploration，比如在各种 Q-Learning 算法中，在不同的 policy state 切换探索策略。

policy.forward

`forward` 函数接收 obs 计算 action，输入和输出由于算法的不同而不同，但大部分情况下是这样的：`(batch, state, ...) -> batch`。

输入的 Batch 是环境中给出的数据（observation、reward、done 和 info），要么来自 `tianshou.data.Collector.collect`（Collecting state），要么来自 “tianshou.data.ReplayBuffer.sample“（Updating state）。Batch 里面的所有数据第一维都是 batch-size。

输出也是一个 Batch，必须包含 `act` 关键字，可能包含 `state` 关键字（用于存放 hiddle state，RNN 使用）、`policy` 关键字（policy 计算过程中需要存储到 buffer 里面的中间结果，比如 logprob 之类的，后续更新网络需要用到），以及其他 key（只不过不会被存储到 buffer 里面）。

比如您想要使用 policy 单独来 evaluate 一个 episode，不用 collect 给出的函数，可以像下面这样做：

```
# env 是 gym.Env
obs, done = env.reset(), False
while not done:
    batch = Batch(obs=[obs])  # 第一维是 batch size
    act = policy(batch).act[0]  # policy.forward 返回一个 batch，使用 ".act" 来取出里面
action 的数据
    obs, rew, done, info = env.step(act)
```

这边 `Batch(obs=[obs])` 会自动为 obs 下面的所有数据创建第 0 维，让它为 batch size=1，否则神经网络没法确定 batch size。

policy.process_fn

`process_fn` 用于计算时间相关的序列信息，比如计算 n-step returns 或者 GAE returns。这边拿 2-step DQN 举例，公式是

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 \max_a Q(s_{t+2}, a)$$

γ 是 discount factor，$\gamma \in [0, 1]$。下面给出了未使用天授的训练过程伪代码：

```
s = env.reset()
buffer = Buffer(size=10000)
agent = DQN()
for i in range(int(1e6)):
    a = agent.compute_action(s)
    s_, r, d, _ = env.step(a)
    buffer.store(s, a, s_, r, d)
    s = s_
    if i % 1000 == 0:
        b_s, b_a, b_s_, b_r, b_d = buffer.get(size=64)
        # 计算 2-step returns, 咋算呢？
        b_ret = compute_2_step_return(buffer, b_r, b_d, ...)
        # 更新 DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)
```

从上面伪代码可以看出我们需要一个依赖于时间相关的接口来计算 2-step returns。`process_fn()` 就是用来做这件事的，给它一个 replay buffer、采样用的 index（相当于时间 t）和采样出来的 batch 就能计算。因为在 buffer 里面我们按照时间顺序存储各种数据，因此 2-step returns 的计算可以像下面这样简单：

```
class DQN_2step(BasePolicy):
    """ 其他的代码"""

    def process_fn(self, batch, buffer, indice):
        buffer_len = len(buffer)
        batch_2 = buffer[(indice + 2) % buffer_len]
        # 上面这个代码访问 batch_2.obs 就是 s_{t+2}, 也可以像下面这样访问:
        #   batch_2_obs = buffer.obs[(indice + 2) % buffer_len]
        # 总之就是 buffer.obs[i] 和 buffer[i].obs 是一个意思, 但是前面的这种写法效率更高
        Q = self(batch_2, eps=0)  # shape: (batch_size, action_shape)
        maxQ = Q.max(dim=-1)
        batch.returns = batch.rew \
            + self._gamma * buffer.rew[(indice + 1) % buffer_len] \
            + self._gamma ** 2 * maxQ
        return batch
```

上面这个代码并没考虑 `done = True` 的情况，因此正确性不能保证，但是它展示了两种能够访问到 s_{t+2} 的方法。

至于 policy 的其他功能，可以参考 tianshou.policy，在最下面给出了一个宏观解释：宏观解释。

1.3.4 Collector

`Collector` 负责 policy 与 env 的交互和数据存储。`collect()` 是 collector 的主要方法，它能够指定让 policy 和环境交互给定数目 `n_step` 个 step 或者 `n_episode` 个 episode，并把该过程中产生的数据存储到 buffer 中。

宏观解释 给出了一个宏观层面的解释，其他 collector 的功能可参考对应文档。此处列出一些常用用法：

```
policy = PGPolicy(...)  # 或者其他 policy 都可以
env = gym.make("CartPole-v0")

replay_buffer = ReplayBuffer(size=10000)

# 这里单个 env 对应 ReplayBuffer
collector = Collector(policy, env, buffer=replay_buffer)

# 多个 env 的话得用 VectorReplayBuffer, 但是 collector 仍然适用
```

(下页继续)

(续上页)

```
vec_buffer = VectorReplayBuffer(total_size=10000, buffer_num=3)
# buffer_num 推荐和 env 数量相等
envs = DummyVectorEnv([lambda: gym.make("CartPole-v0") for _ in range(3)])
collector = Collector(policy, envs, buffer=vec_buffer)

# 收集 3 个 episode
collector.collect(n_episode=3)
# 收集至少俩 step (这个会收集三个，因为有三个 env，每次收集的次数得是 3 的倍数)
collector.collect(n_step=2)
# 边收集变直播，使用 render 参数就可以 (render 传入的是时间间隔，以秒为单位)
collector.collect(n_episode=1, render=0.03)
```

还有个:class:*~tianshou.data.AsyncCollector*，继承了:class:*~tianshou.data.Collector*，它支持异步的环境采样（比如环境很慢或者 step 时间差异很大）。不过 AsyncCollector 的 collect 的语义和上面 Collector 有所不同，由于异步的特性，它只能保证 ** 至少 ** n_step 或者 n_episode 地收集数据。

1.3.5 Trainer

有了之前声明的 collector 和 policy 之后，就可以用 trainer 把它们包起来。Trainer 负责最上层训练逻辑的控制，例如训练多少次之后进行策略和环境的交互。现有的训练器包括同策略学习训练器（On-policy Trainer）和异策略学习训练器（Off-policy Trainer）。

天授未显式地将训练器抽象成一个类，因为在其他现有平台中都将类似训练器的实现抽象封装成一个类，导致用户难以二次开发。因此以函数的方式实现训练器，并提供了示例代码便于研究者进行定制化训练策略的开发。可以参考定制化训练器。

1.3.6 宏观解释

接下来将通过一段伪代码的讲解来阐释上述所有抽象模块的应用。

```
# pseudocode, cannot work                                          # 对应天授实现
s = env.reset()                                                    # 环境初始化，在 env 中实
现
buffer = Buffer(size=10000)                                        # buffer = tianshou.
↪data.ReplayBuffer(size=10000)
agent = DQN()                                                      # policy.__init__(...)
for i in range(int(1e6)):                                          # 在 Trainer 中实现
    a = agent.compute_action(s)                                    # act = policy(batch,␣
↪...).act
    s_, r, d, _ = env.step(a)                                      # collector.collect(..
↪.)
    buffer.store(s, a, s_, r, d)                                   # collector.collect(..
↪.)
    s = s_                                                         # collector.collect(..
↪.)
    if i % 1000 == 0:                                              # 在 Trainer 中实现
                                                                   # the following is␣
↪done in policy.update(batch_size, buffer)
        b_s, b_a, b_s_, b_r, b_d = buffer.get(size=64)             # batch, indice =␣
↪buffer.sample(batch_size)
        # 计算 2-step returns, 咋算呢？
        b_ret = compute_2_step_return(buffer, b_r, b_d, ...)       # policy.process_
↪fn(batch, buffer, indice)
```

(下页继续)

(续上页)

```
        # 更新 DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)           # policy.learn(batch,␣
↪...)
```

1.4 标准测试

1.4.1 Mujoco Benchmark

天授目前拥有市面上所有平台中最好的 Mujoco 结果（甚至比 SpinningUp 还要好!）

看这里：https://github.com/thu-ml/tianshou/tree/master/examples/mujoco

1.4.2 Atari Benchmark

戳这里：https://github.com/thu-ml/tianshou/tree/master/examples/atari

1.5 速查手册

本页面列举出了一些天授平台的常用使用方法。

1.5.1 搭建神经网络

参见建立网络。

1.5.2 构建策略

参见 BasePolicy。

1.5.3 定制化训练策略

参见定制化训练器。

1.5.4 环境并行采样

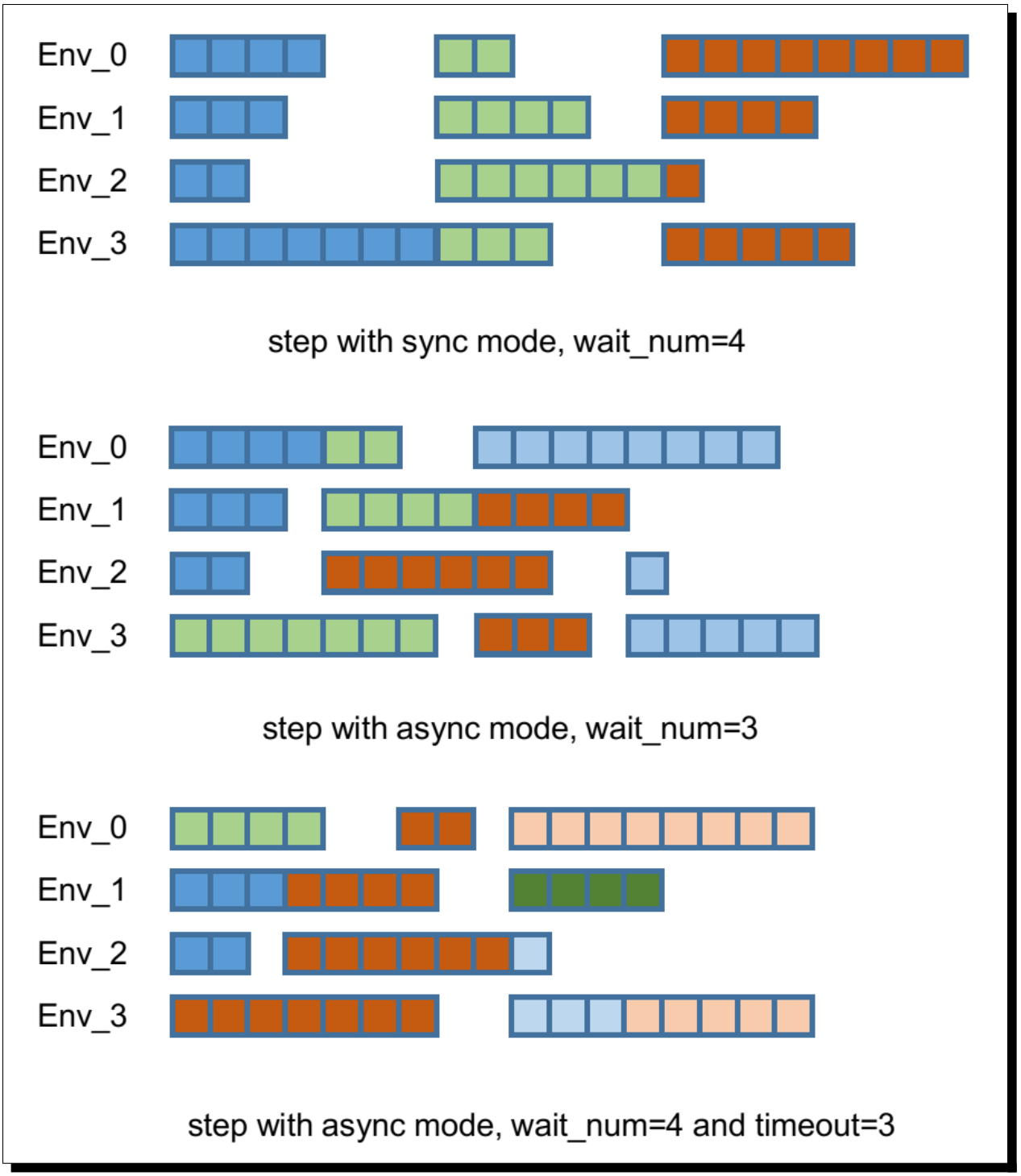
天授提供了四种类：

- `DummyVectorEnv` 使用原始的 for 循环实现，可用于 debug，小规模的环境用这个的开销会比其他三种小
- `SubprocVectorEnv` 用多进程来实现的，最常用
- `ShmemVectorEnv` 是上面这个多进程实现的一个改进：把环境的 obs 用一个 shared buffer 来存储，降低比较大的 obs 的开销（比如图片等）
- `RayVectorEnv` 基于 Ray 的实现，可以用于多机

这些类虽说是用于不同的场景中，但是他们的 API 都是一致的，只需要提供一个可调用的 env 列表即可，像这样：

```
env_fns = [lambda x=i: MyTestEnv(size=x) for i in [2, 3, 4, 5]]
venv = SubprocVectorEnv(env_fns)  # DummyVectorEnv/ShmemVectorEnv/RayVectorEnv 都行
venv.reset()  # 这个会返回每个环境最初的 obs
venv.step(actions)  # actions 长度是 env 的数量，返回也是这些 env 原始返回值 concat 起来之后的结果
```

一个 venv 同步或异步执行的例子，相同颜色代表这些 episode 会组合起来由 venv 返回出去



默认情况下是使用同步模式（sync mode），就像图中最上面那样。如果每个 env step 耗时差不多的话，这种模式开销最小。但如果每个 env step 耗时差别很大的话（比如通常 1s，偶尔会 10s），这个时候 async 就派上用场了（Issue 103）：只需要多提供两个参数（或者其中之一也行），一个是 `wait_num`，表示一旦达到这么多 env 结束就返回（比如 4 个 env，设置 `wait_num = 3` 的话，每一步 venv.step 只会返回 4 个 env 中的 3 个结果）；另一个是 `timeout`，表示一旦超过这个时间并且有 env 已经结束了的话就返回结果。

```
env_fns = [lambda x=i: MyTestEnv(size=x, sleep=x) for i in [2, 3, 4, 5]]
```

（下页继续）

(续上页)

```
venv = SubprocVectorEnv(env_fns, wait_num=3, timeout=0.2)
venv.reset()
# returns "wait_num" steps or finished steps after "timeout" seconds,
# whichever occurs first.
venv.step(actions, ready_id)
```

警告： 如果自定义环境的话，记得设置 seed 比如这样：

```
def seed(self, seed):
    np.random.seed(seed)
```

如果 seed 没有被重写，每个环境的 seed 都是全局的 seed，都会产生一样的结果，相当于一个 env 的数据复制了好多次，没啥用。

1.5.5 批处理数据组

本条目与 Issue 42 相关。

如果想收集训练 log、预处理图像数据（比如 Atari 要 resize 到 84x84x3 -- 不过这个推荐直接 wrapper 做）、根据环境信息修改奖励函数的值，可以在 Collector 中使用 `preprocess_fn` 接口，它会在数据存入 Buffer 之前被调用。

`preprocess_fn` 有两种输入接口：如果是 env.reset() 的话，它只会接收 obs；如果是正常的 env.step()，那么他会接收 5 个关键字"obs_next"/"rew"/"done"/"info"/"policy"。返回一个字典或者 Batch，里面包含着你想修改的东西。

```
import numpy as np
from collections import deque


class MyProcessor:
    def __init__(self, size=100):
        self.episode_log = None
        self.main_log = deque(maxlen=size)
        self.main_log.append(0)
        self.baseline = 0

    def preprocess_fn(**kwargs):
        """ 把 reward 给归一化"""
        if 'rew' not in kwargs:
            # 意味着 preprocess_fn 是在 env.reset() 之后被调用的，此时 kwargs 里面只有 obs
            return Batch()  # 没有变量需要更新，返回空
        else:
            n = len(kwargs['rew'])  # Collector 中的环境数量
            if self.episode_log is None:
                self.episode_log = [[] for i in range(n)]
            for i in range(n):
                self.episode_log[i].append(kwargs['rew'][i])
                kwargs['rew'][i] -= self.baseline
            for i in range(n):
                if kwargs['done']:
                    self.main_log.append(np.mean(self.episode_log[i]))
                    self.episode_log[i] = []
```

(下页继续)

(续上页)

```
                self.baseline = np.mean(self.main_log)
        return Batch(rew=kwargs['rew'])
```

最终只需要在 Collector 声明的时候加入一下这个 hooker：

```
test_processor = MyProcessor(size=100)
collector = Collector(policy, env, buffer, preprocess_fn=test_processor.preprocess_fn)
```

还有一些示例在 test/base/test_collector.py 中可以查看。

1.5.6 RNN 训练

本条目与 Issue 19 相关

首先在 ReplayBuffer 的声明中加入 `stack_num`（堆叠采样）参数，表示在训练 RNN 的时候要扔给网络多少个 timestep 进行训练：

```
buf = ReplayBuffer(size=size, stack_num=stack_num)
```

然后把神经网络模型中 `state` 参数用起来，可以参考 `Recurrent`、`RecurrentActorProb` 和 `RecurrentCritic`。

以上代码片段展示了如何修改 ReplayBuffer 和神经网络模型，从而使用堆叠采样的观测值（stacked-obs）来训练 RNN。如果想要堆叠别的值（比如 stacked-action 来训练 Q(stacked-obs, stacked-action)），可以使用一个 `gym.Wrapper` 来修改状态表示，比如 wrapper 把状态改为 [s, a] 的元组：

- 之前的数据存储：(s, a, s', r, d)，可以获得堆叠的 s
- 采用 wrapper 之后的存储：([s, a], a, [s', a'], r, d)，可以获得堆叠的 [s, a]，拆开来就是堆叠的 s 和 a

1.5.7 自定义环境与状态表示

本条目与 Issue 38 和 Issue 69 相关。

首先，自定义的环境必须遵守 OpenAI Gym 定义的 API 规范，下面列出了一些：

- reset() -> state
- step(action) -> state, reward, done, info
- seed(s) -> List[int]
- render(mode) -> Any
- close() -> None
- observation_space: gym.Space
- action_space: gym.Space

环境状态（state）可以是一个 `numpy.ndarray` 或者一个 Python 字典。比如以 `FetchReach-v1` 环境为例：

```
>>> e = gym.make('FetchReach-v1')
>>> e.reset()
{'observation': array([ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,  1.
↪97805133e-04,
        7.15193042e-05,  7.73933014e-06,  5.51992816e-08, -2.42927453e-06,
        4.73325650e-06, -2.28455228e-06]),
```

(下页继续)

(续上页)

```
 'achieved_goal': array([1.34183265, 0.74910039, 0.53472272]),
 'desired_goal': array([1.24073906, 0.77753463, 0.63457791])}
```

这个环境（GoalEnv）是个三个 key 的字典，天授会将其按照如下格式存储：

```
>>> from tianshou.data import ReplayBuffer
>>> b = ReplayBuffer(size=3)
>>> b.add(obs=e.reset(), act=0, rew=0, done=0)
>>> print(b)
ReplayBuffer(
    act: array([0, 0, 0]),
    done: array([0, 0, 0]),
    info: Batch(),
    obs: Batch(
             achieved_goal: array([[1.34183265, 0.74910039, 0.53472272],
                                   [0.        , 0.        , 0.        ],
                                   [0.        , 0.        , 0.        ]]),
             desired_goal: array([[1.42154265, 0.62505137, 0.62929863],
                                  [0.        , 0.        , 0.        ],
                                  [0.        , 0.        , 0.        ]]),
             observation: array([[ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,
                                   1.97805133e-04,  7.15193042e-05,  7.73933014e-06,
                                   5.51992816e-08, -2.42927453e-06,  4.73325650e-06,
                                  -2.28455228e-06],
                                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00],
                                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00]]),
         ),
    policy: Batch(),
    rew: array([0, 0, 0]),
)
>>> print(b.obs.achieved_goal)
[[1.34183265 0.74910039 0.53472272]
 [0.         0.         0.        ]
 [0.         0.         0.        ]]
```

也可以很方便地从 Buffer 中采样出数据：

```
>>> batch, indice = b.sample(2)
>>> batch.keys()
['act', 'done', 'info', 'obs', 'obs_next', 'policy', 'rew']
>>> batch.obs[-1]
Batch(
    achieved_goal: array([1.34183265, 0.74910039, 0.53472272]),
    desired_goal: array([1.42154265, 0.62505137, 0.62929863]),
    observation: array([ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,  1.
↪97805133e-04,
                         7.15193042e-05,  7.73933014e-06,  5.51992816e-08, -2.
↪42927453e-06,
                         4.73325650e-06, -2.28455228e-06]),
)
```

(下页继续)

(续上页)

```
>>> batch.obs.desired_goal[-1]  # 推荐，没有深拷贝
array([1.42154265, 0.62505137, 0.62929863])
>>> batch.obs[-1].desired_goal  # 不推荐
array([1.42154265, 0.62505137, 0.62929863])
>>> batch[-1].obs.desired_goal  # 不推荐
array([1.42154265, 0.62505137, 0.62929863])
```

因此只需在自定义的网络中，换一下 `forward` 函数的 state 写法：

```
def forward(self, s, ...):
    # s is a Batch
    observation = s.observation
    achieved_goal = s.achieved_goal
    desired_goal = s.desired_goal
    ...
```

当然如果自定义的环境中，状态是一个自定义的类，也是可以的。不过天授只会把它的地址进行存储，就像下面这样（状态是 nx.Graph）：

```
>>> # 这个例子可能现在不太能 work，因为 numpy 升级了，以及 nx.Graph 重写了__getitem__，导致 np.
↪array([nx.Graph()]) 会出来空的数组……
>>> # 不过正常的自定义 class 应该没啥问题
>>> import networkx as nx
>>> b = ReplayBuffer(size=3)
>>> b.add(obs=nx.Graph(), act=0, rew=0, done=0)
>>> print(b)
ReplayBuffer(
    act: array([0, 0, 0]),
    done: array([0, 0, 0]),
    info: Batch(),
    obs: array([<networkx.classes.graph.Graph object at 0x7f5c607826a0>, None,
                None], dtype=object),
    policy: Batch(),
    rew: array([0, 0, 0]),
)
```

由于只存储了引用，因此如果状态修改的话，有可能之前存储的状态也会跟着修改。为了确保不出 bug，建议在返回这个状态的时候加上深拷贝（deepcopy）：

```
def reset():
    return copy.deepcopy(self.graph)
def step(a):
    ...
    return copy.deepcopy(self.graph), reward, done, {}
```

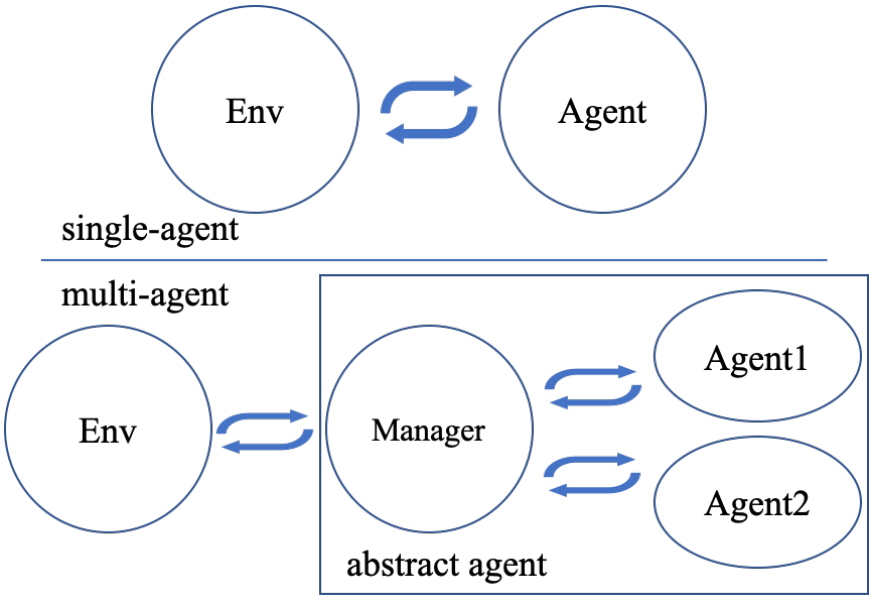
1.5.8 多智能体强化学习

本条目与 Issue 121 相关。

多智能体强化学习大概可以分为如下三类：

1. Simultaneous move：所有玩家在每个 timestep 都同时行动，比如 moba 游戏；
2. Cyclic move：每个玩家轮流行动，比如飞行棋；
3. Conditional move：每个玩家在当前 timestep 下面所能采取的行动受限于环境，比如 Pig Game。

这些基本上都能被转换为正常 RL 的形式。比如第一个 simultaneous move 只需要加一个 `num_agent` 标记一下，剩下代码都不用变；2 和 3 的话，可以统一起来：环境在每个 timestep 选择 id 为 `agent_id` 的玩家进行游戏，更近一步把“所有的玩家”看做一个抽象的玩家的话（可以称之为 MultiAgentPolicyManager，多智能体策略代理），就相当于单个玩家的情况，只不过每次多了个信息叫做 `agent_id`，由这个代理转发给下属的各个玩家即可。至于 3 的 condition，只需要多加一个信息叫做 mask 就行了。大概像下面这张图一样：



可以把上述文字描述形式化为下面的伪代码：

```
action = policy(state, agent_id, mask)
(next_state, next_agent_id, next_mask), reward = env.step(action)
```

于是只要创建一个新的 state：`state_ = (state, agent_id, mask)`，就可以使用之前正常的代码：

```
action = policy(state_)
next_state_, reward = env.step(action)
```

基于这种思路，我们写了个用 DQN 玩 四子棋 的 demo，可以在 这里 查看。

1.6 基于 PyTorch 的深度强化学习平台设计与实现

这是 PDF 版本的链接。

1.6.1 中文摘要

深度强化学习近年来取得了一系列的突破，在包括 Atari 游戏 [[MKS+15]] 、围棋 [[SHM+16]] 、蛋白质结构预测 [[SEJ+20]] 和策略游戏 Dota2 [[BBC+19]] 等多个领域取得了极大进展，提升了业界对深度强化学习的需求与信心。但是，目前主流深度强化学习平台框架无法很好地满足这一日渐增长的需求。无论是在学术研究领域还是工业应用，现有框架普遍存在缺乏灵活的可定制化接口、代码嵌套关系复杂、训练速度较慢、完整单元测试缺失等缺点。研究者们通常需要大幅改动框架结构，甚至需要从头开始编写算法程序才能满足自身需求，阻碍了强化学习技术的进一步应用。因此，一个灵活可定制、代码简洁、训练速度快、有着可靠测试的强化学习平台对整个领域而言十分重要。

针对以上问题，本项目构建了一个基于 PyTorch [[PGM+19]] 的深度强化学习平台 **天授**。天授平台仅通过 2000 余行代码，简洁地实现了基于策略梯度、基于 Q 价值函数、综合 Q 价值与策略梯度、模仿学习等 10 余种主流强化学习算法及其主要改进，支持了部分观测马尔科夫决策过程训练以及任意仿真环境的数据处理，将主流强化学习算法充分模块化并实现了需求可定制化，在和其它著名强化学习平台进行的性能对比评测中以显著优势胜出。天授旨在为用户提供一个更加友好的强化学习算法平台，降低强化学习算法的开发成本。平台代码已经在 GitHub 上开源：https://github.com/thu-ml/tianshou/，目前已获得超过 1500 个星标，受到学术界和产业界的广泛关注。

关键词：强化学习，算法，平台，PyTorch

1.6.2 主要符号对照表

符号	说明
RL	强化学习 (Reinforcement Learning)
MFRL	免模型强化学习 (Model-free Reinforcement Learning)
MBRL	基于模型的强化学习 (Model-based Reinforcement Learning)
MARL	多智能体强化学习 (Multi-agent Reinforcement Learning)
MetaRL	元强化学习（Meta Reinforcement Learning）
IL	模仿学习 (Imitation Learning)
On-policy	同策略
Off-policy	异策略
MDP	马尔科夫决策过程 (Markov Decision Process)
POMDP	部分可观测马尔科夫决策过程 (Partially Observable Markov Decision Process)
Agent	智能体
π，Policy	策略
Actor	动作（网络），又称作策略（网络）
Critic	评价（网络）
$s \in \mathcal{S}$，State	状态
$o \in \mathcal{O}$，Observation	观测值，为状态的一部分，$o \subseteq s$
$a \in \mathcal{A}$，Action	动作
$r \in \mathcal{R}$，Reward	奖励
$d \in \{0, 1\}$，Done	结束符，0 表示未结束，1 表示结束
s_t, o_t, a_t, r_t, d_t	在一个轨迹中时刻 t 的状态、观测值、动作、奖励和结束符
$P^a_{ss'} \in \mathcal{P}$	在当前状态 s 采取动作 a 之后，转移到状态 s' 的概率；$P^a_{ss'} = \mathbb{P}\{s_{t+1} = s' \vert s_t = s, a_t = a\}$
R^a_s	在当前状态 s 采取动作 a 之后所能获得的期望奖励；$R^a_s = \mathbb{E}[r_t \vert s_t = s, a_t = a]$
γ	折扣因子，作为对未来回报不确定性的一个约束项，$\gamma \in [0, 1]$
G_t，Return	累计折扣回报，$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$
$\pi(a\vert s)$	随机性策略，表示获取状态 s 之后采取的动作 a 的概率
$\pi(s)$	确定性策略，表示获取状态 s 之后采取的动作
$V(s)$	状态值函数（State-Value Function），表示状态 s 对应的期望累计折扣回报
$V^\pi(s)$	使用策略 π 所对应的状态值函数，$V^\pi(s) = \mathbb{E}_\pi[G_t \vert s_t = s]$
$Q(s, a)$	动作值函数（Action-Value Function），表示状态 s 下采取动作 a 所对应的期望累计折扣回报
$Q^\pi(s, a)$	使用策略 π 所对应的动作值函数，$Q^\pi(s, a) = \mathbb{E}_{a\sim\pi}[G_t \vert s_t = s, a_t = a]$
$A(s, a)$	优势函数，$A(s, a) = Q(s, a) - V(s)$
Batch	数据组
Buffer	数据缓冲区
Replay Buffer	重放缓冲区
RNN	循环神经网络（Recurrent Neural Network）

1.6.3 引言

深度强化学习研究背景

在 2012 年 AlexNet [[KSH12]] 夺得 ImageNet 图像分类比赛冠军之后，深度神经网络被应用于许多领域，如目标检测和跟踪 [[HGDollarG17], [RF18]]；并且在一系列的任务中，深度学习模型的准确率达到甚至超过了人类水准。大规模的商业化应用随之而来，如人脸识别 [[TYRW14]] 、医疗图像处理 [[RFB15]] 等领域都使用深度神经网络以提高识别的速度和精度。

强化学习概念的提出最早可追溯到 20 世纪，其在简单场景上的应用于上世纪 90 年代至本世纪初即被深入研究，比如 1992 年强化学习算法打败了人类西洋双陆棋玩家 [[Tes94]]。早期的强化学习算法大多使用线性回归来拟合策略函数，并且需要预先提取人为定义好的特征，因此实际效果不甚理想。2013 年之后，结合了深度学习的优势，深度强化学习使用深度神经网络进行函数拟合，展现出了其强大的威力，如使用 DQN [[MKS+15]] 玩 Atari 游戏的水平达到人类水准、AlphaGo [[SHM+16]] 与人类顶尖围棋选手的划时代人机对战、OpenAI Five [[BBC+19]] 在 Dota2 5v5 对战比赛中击败人类冠军团队等，无论是学术界还是工业界都对这一领域表现出了极大兴趣。深度强化学习如今不但被应用在游戏 AI 中，还被使用在如机械臂抓取、自动驾驶、高频交易、智能交通等实际场景中，前景十分广阔。

深度强化学习平台框架现状

现有深度强化学习平台简介

深度强化学习算法由于其计算模式不规则和高并发的特点，无法像计算机视觉、自然语言处理领域的计算框架一样，从训练数据流的角度进行设计与实现；而强化学习算法形式与实现细节难以统一，又进一步加大了平台的编写难度。尽管一些现有项目尝试搭建通用强化学习算法的框架，但其效果并不理想，深度强化学习领域的研究者往往需要从头编写一个特定强化学习算法的程序来满足自己的需求。

现有使用较为广泛的深度强化学习平台包括 OpenAI 的 Baselines [[DHK+17]]、SpinningUp [[Ach18]]，加州伯克利大学的开源分布式强化学习框架 RLlib [[LLN+18]]、rlpyt [[SA19]]、rlkit [[PDLN19]]、Garage [[gc19]]，谷歌公司的 Dopamine [[CMG+18]]、B-suite [[ODH+20]]，以及其他独立开发的平台 Stable-Baselines [[HRE+18]]、keras-rl [[Pla16]]、PyTorch-DRL [[Chr19]]、TensorForce [[KSF17]]。图 *1.1* 展示了若干主流强化学习算法平台的标志，表 *1.1* 列举了各个框架的基本信息。

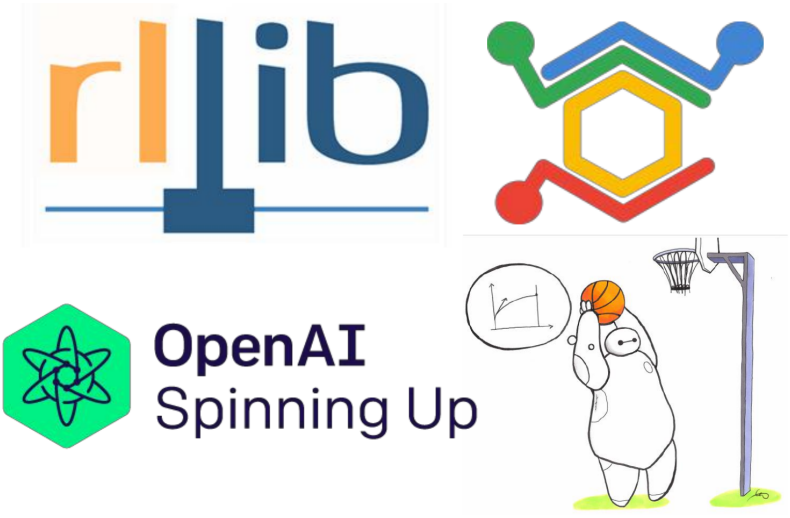


图 1: 图 1.1：目前较为主流的深度强化学习算法平台

几乎所有的强化学习平台都以 OpenAI Gym [[BCP+16]] 所定义的 API 作为智能体与环境进行交互的标准接口，以 TensorFlow [[ABC+16]] 作为后端深度学习框架的平台居多，支持至少 4 种免模型强化学习算法。大部

分平台支持对训练环境进行自定义配置。

PyTorch [[PGM+19]] 是 Facebook 公司推出的一款开源深度学习框架，由于其易用性、接口稳定性和社区活跃性，受到越来越多学术界和工业界研究者的青睐，大有超过 TensorFlow 框架的趋势。然而使用 PyTorch 编写的深度强化学习框架中，星标最多为 PyTorch-DRL [[Chr19]]（2400+ 星标），其活跃度远远不及 TensorFlow 强化学习社区中的开源框架。本文将在下一小节分析其原因。

现有深度强化学习平台不足

表 2: 表 1.1：深度强化学习平台总览，按照 GitHub 星标数从大到小排序，截止 2020/05/12

平台名称	星标数	后端框架	模块化	文档	代码质量	单元测试	上次更新
Ray/RLlib [[LLN+18]]	11460	TF/PyTorch	√	较全	10 / 24065	√	2020.5
Baselines [[DHK+17]]	9764	TF	×	无	2673 / 10411	√	2020.1
Dopamine [[CMG+18]]	8845	TF1	√	较全	180 / 2519	√	2019.12
SpinningUp [[Ach18]]	4630	TF1/PyTorch	×	全面	1656 / 3724	×	2019.11
keras-rl [[Pla16]]	4612	Keras	√	不全	522 / 2346	√	2019.11
Tensorforce [[KSF17]]	2669	TF	√	全面	3834 / 13609	√	2020.5
PyTorch-DRL [[Chr19]]	2424	PyTorch	√	无	2144 / 4307	√	2020.2
Stable-Baselines [[HRE+18]]	2054	TF1	×	全面	2891 / 10989	√	2020.5
天授	1529	PyTorch	√	全面	0 / 2141	√	2020.5
rlpyt [[SA19]]	1448	PyTorch	√	较全	1191 / 14493	×	2020.4
rlkit [[PDLN19]]	1172	PyTorch	√	不全	275 / 7824	×	2020.3
B-suite [[ODH+20]]	975	TF2	×	无	220 / 5353	×	2020.5
Garage [[gc19]]	709	TF1/PyTorch	√	不全	5 / 17820	√	2020.5

注：TF 为 TensorFlow 缩写，包含版本 v1 和 v2；TF1 为 TensorFlow v1 版本缩写，不包含版本 v2；TF2 为 TensorFlow v2 版本缩写，不包含版本 v1；代码质量一栏数据格式为 “PEP8 **不符合**规范数 / 项目 Python 文件行数”。

表 1.1 按照 GitHub 星标数目降序排列，从后端框架、是否模块化、文档完善程度、代码质量、单元测试和最后维护时间这些维度，对比了比较流行的深度强化学习开源平台框架。这些平台框架在不同评价维度上或多或少存在一些缺陷，降低了用户体验。此处列出一些典型问题，如下所示：

- **算法模块化不足：**以 OpenAI Baselines 为代表，将每个强化学习算法单独独立成一份代码，无法做到代码之间的复用。用户在使用相关代码时，必须逐一修改每份代码，带来了极大困难。
- **实现算法种类有限：**以 Dopamine 和 SpinningUp 为代表，Dopamine 框架只支持 DQN 算法族，并不支持策略梯度；SpinningUp 只支持策略梯度算法族，未实现 Q 学习的一系列算法。两个著名的平台所支持的强化学习算法均不全面。

- **代码实现复杂度过高：**以 RLlib 为代表，代码层层封装嵌套，用户难以进行二次开发。
- **文档不完整：**完整的文档应包含教程和代码注释，部分平台只实现了其一，甚至完全没有文档，十分影响平台框架的使用。
- **平台性能不佳：**强化学习算法本身难以调试，如果能够提升平台性能则将会大幅度降低调试难度。仍然以 OpenAI Baselines 为代表，该平台无法全面支持并行环境采样，十分影响训练效率。
- **缺少完整单元测试：**单元测试保证了代码的正确性和结果可复现性，但几乎所有平台都只做了功能性验证，而没有进行完整的训练过程验证。
- **环境定制支持不足：**许多非强化学习领域的研究者想使用强化学习算法来解决自己领域内问题，因此所交互的环境并不一定是 OpenAI Gym [[BCP+16]] 已经定制好的，这要求平台框架支持更多种类的环境，比如机械臂抓取所需的多模态环境。以 rlpyt 为例，该平台将环境进行封装，如果想使用非 Atari 的环境，研究者必须大费周折改动框架代码。

此外，另一个值得讨论的问题是 PyTorch 深度强化学习框架活跃程度不如 TensorFlow 社区的。不少使用 PyTorch 的研究者通过编写独立的强化学习算法来满足自己需求，虽然实现较 TensorFlow 简单很多，但却没有针对数据流、数据存储进行优化；从表 *1.1* 中也可以看出以 PyTorch-DRL 为代表的基于 PyTorch 的深度强化学习平台，文档不全面、代码质量不及独立手写的算法，亦或是封装程度过高、缺乏可靠的单元测试，这些问题一定程度上阻碍了这些平台的进一步发展。

主要贡献与论文结构

主要贡献

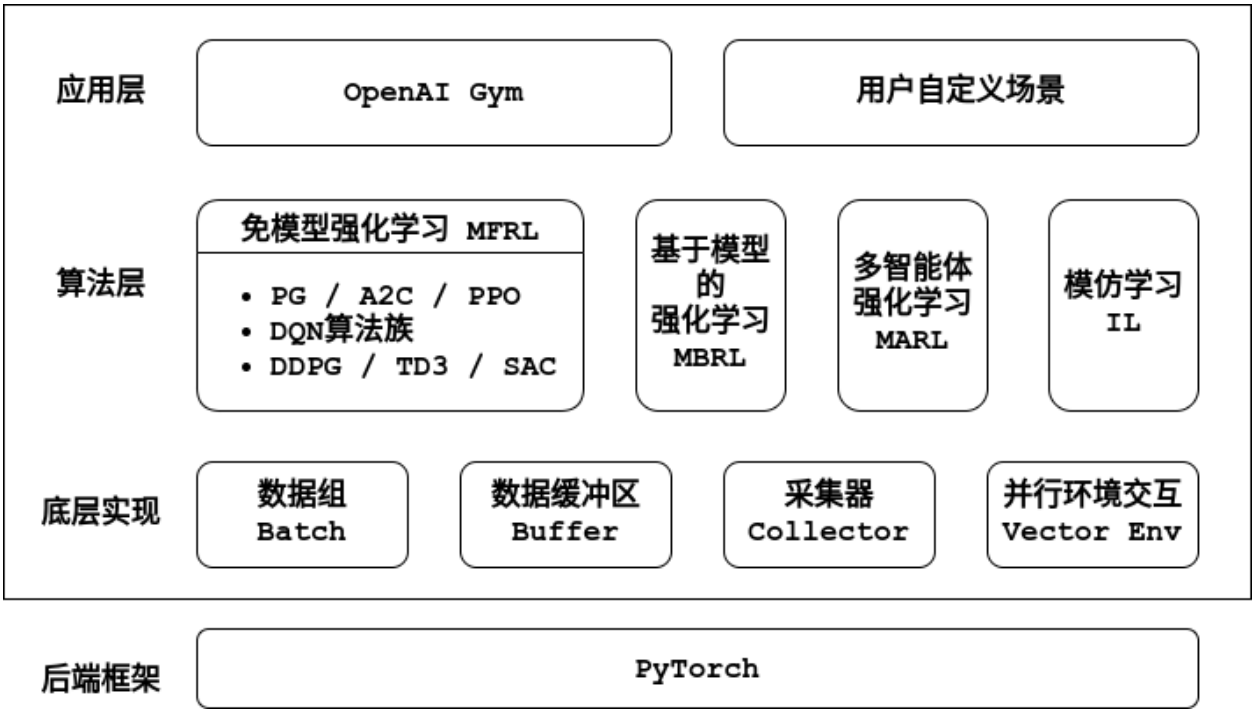


图 2: 图 1.2：天授平台总体架构

本文描述了“天授”，一个基于 PyTorch 的深度强化学习算法平台。图 *1.2* 描述了该平台的总体架构。天授平台以 PyTorch 作为深度学习后端框架，将各个强化学习算法加以模块化，在数据层面抽象出了数据组（Batch）、数据缓冲区（Buffer）、采集器（Collector）三个基本模块，实现了针对任意环境的并行交互与采样功能，算法

层面支持丰富多样的强化学习算法，如免模型强化学习（MFRL）中的一系列算法、模仿学习算法（IL）等，从而能够让研究者方便地使用不同算法来测试不同场景。

天授拥有创新的模块化设计，简洁地实现了各种强化学习算法，支持了用户各种各样的需求。在相关的性能实验评测中，天授在众多强化学习平台夺得头筹。种种亮点使其获得了强化学习社区不小的关注度，在 GitHub 上开源不到短短一个月，星标就超过了基于 PyTorch 的另一个著名的强化学习平台 rlpyt [[SA19]]。

论文结构

接下来的论文结构安排如下所示：

平台设计与实现：描述了天授平台的设计与实现，将强化学习算法加以抽象凝练，分析提取出共有部分，介绍模块化的实现；以及介绍平台的其他特点。

平台支持的深度强化学习算法：描述了天授平台目前所支持的各类深度强化学习算法，介绍各个算法的基本原理以及在天授平台中的实现细节。

平台对比评测：对比了天授平台与若干已有的著名深度强化学习平台的优劣之处，包括功能层面和性能层面的测试。

平台使用实例：列举出了若干天授平台的典型使用样例，使读者能够进一步了解平台的接口和使用方法。

总结：对天授平台特点进行总结，并指出后续的工作方向。

1.6.4 平台设计与实现

本章首先介绍深度强化学习的核心问题及其形式化建模，随后介绍平台的整体架构、模块设计和实现细节，最后简要提及平台的外围支持。

深度强化学习问题描述

强化学习问题不同于机器学习领域中的监督学习问题或无监督学习问题。给定一个输入 x，这三种学习类型的算法会有不同的输出：

- 监督学习：输出预测值 y；
- 无监督学习：输出 x 的潜在模式 z，如聚类、密度估计、降维、隐变量推理等；
- 强化学习：输出动作 a 使得能够最大化期望累计奖励。

强化学习算法是在不确定环境中，通过与环境的不断交互，来不断优化自身策略的算法。它和监督学习、非监督学习有着一些本质上的区别：

1. 获取的数据非独立同分布：大部分机器学习算法假设数据是独立同分布的，否则会有收敛性问题；然而智能体与环境进行交互产生的数据具有很强的时间相关性，无法在数据层面做到完全的解耦，不满足数据的独立同分布性质，因此强化学习算法训练并不稳定；智能体的行为同时会影响后续的数据分布；
2. 没有“正确”的行为，且无法立刻获得反馈：监督学习有专门的样本标签，而强化学习并没有类似的强监督信号，通常只有基于奖励函数的单一信号；强化学习场景存在延迟奖励的问题，智能体不能在单个样本中立即获得反馈，需要不断试错，还需要平衡短期奖励与长期奖励的权重；
3. 具有超人类的上限：传统的机器学习算法依赖人工标注好的数据，从中训练好的模型的性能上限是产生数据的模型（人类）的上限；而强化学习可以从零开始和环境进行不断地交互，可以不受人类先验知识的桎梏，从而能够在一些任务中获得超越人类的表现。

问题定义

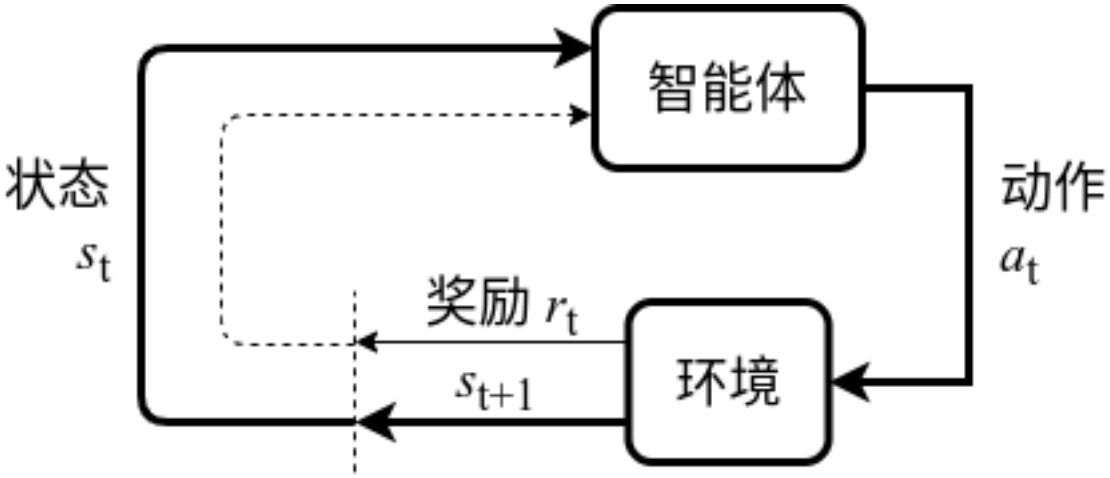


图 3: 图 2.1：强化学习算法中智能体与环境循环交互的过程

强化学习问题是定义在马尔科夫决策过程（Markov Decision Process，MDP）之上的。一个 MDP 是形如 $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \rho_0 \rangle$ 的五元组，其中：

- $\mathcal{S}$ 是所有合法状态的集合；
- $\mathcal{A}$ 是所有合法动作的集合；
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 是奖励函数，t 时刻的奖励函数 r_t 由 s_t, a_t 决定，使用 R_s^a 表示在当前状态 s 采取动作 a 之后所能获得的期望奖励；
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ 是状态转移概率函数，使用 $P_{ss'}^a$ 来表示当前状态 s 采取动作 a 转移到状态 s' 的概率；
- ρ_0 是初始状态的概率分布，$\sum_{s \in \mathcal{S}} \rho_0(s) = 1$。

MDP 还具有马尔科夫性质，即在任意 t 时刻，下一个状态 s_{t+1} 的概率分布只能由当前状态 s_t 决定，与过去的任何状态 $\{s_0, \ldots, s_{t-1}\}$ 均无关。

图 *2.1* 描述了在经典的强化学习场景中，智能体与环境不断交互的过程：在 t 时刻，智能体获得了环境状态 s_t，经过计算输出动作值 a_t 并在环境中执行，环境会返回 $t+1$ 时刻的环境状态 s_{t+1} 与上一个时刻产生的奖励 r_t。

在某些场景中，智能体无法获取到整个环境的状态，比如扑克、麻将等不完全信息对弈场景，此时称整个过程为部分可观测马尔科夫决策过程 (Partially Observable Markov Decision Process，POMDP)。在智能体与环境交互的每个回合中，智能体只能接收到观测值 o_t，为状态 s_t 的一部分。

定义累积折扣回报 G_t 为从 t 时刻起的加权奖励函数总和

$$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$$

其中 $\gamma \in [0, 1]$ 是折扣因子，衡量了智能体对短期回报与长期回报的权重分配。通常用 $\pi_\theta(\cdot)$ 表示一个以参数 θ 参数化的策略 π。强化学习算法优化目标是最大化智能体每一回合的累计折扣回报的期望，形式化如下：

$$\theta^* = \arg\max_\theta \mathbb{E}_{\pi_\theta}[G_t]$$

智能体的组成

一个智能体主要由策略函数（Policy Function）、价值函数（Value Function）和环境模型（Environment Model）三个部分组成。

策略函数：智能体根据当前环境状态，输出动作值的函数。策略函数分为确定性策略函数与随机性策略函数。

确定性策略函数通常有两种形式：（1）$a_t = \pi_\theta(s_t)$，直接输出动作值；（2）$a_t = \arg\max_a \pi_\theta(a|s_t)$，评估在状态 s_t 下所有可能的策略并从中选取最好的动作。

随机性策略函数的形式主要为计算一个概率分布 $\pi_\theta(a|s_t)$，从这个概率分布中采样出动作值 a_t。常用的分布有用于离散动作空间的类别分布（Categorical Distribution）、用于连续动作空间的对角高斯分布（Diagonal Gaussian Distribution）。

价值函数：价值函数是智能体对当前状态、或者是状态-动作对进行评估的函数，主要有三种形式：

1. 状态值函数（State-Value Function）$V(s)$：状态 s 对应的期望累计折扣回报，$V(s) = \mathbb{E}_\pi[G_t|s_t = s]$；
2. 动作值函数（Action-Value Function）$Q(s,a)$：状态 s 在采取动作 a 的时候对应的期望累计折扣回报，$Q(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$
3. 优势函数（Advantage Function）$A(s,a)$：状态 s 下采取动作 a 的情况下，比平均情况要好多少，$A(s,a) = Q(s,a) - V(s)$。

环境模型：智能体还可以对环境中的状态转移函数进行建模，比如使用映射 $\mathcal{F}: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ 进行对环境转移 $\max_{s'} P^a_{ss'}$ 的拟合，或者是对奖励函数 R^a_s 的分布进行拟合。

现有深度强化学习算法分类

强化学习算法按照是否对环境进行建模来划分，可分为免模型强化学习（Model-free Reinforcement Learning，MFRL）与基于模型的强化学习（Model-based Reinforcement Learning，MBRL）两大类；此外还有多智能体学习（Multi-agent Reinforcement Learning，MARL）、元强化学习（Meta Reinforcement Learning）、模仿学习（Imitation Learning，IL）这几个大类。现有深度强化学习平台主要实现免模型强化学习算法。

免模型强化学习算法按照模型的学习特性进行区分，可分为同策略学习（On-policy Learning）和异策略学习（Off-policy Learning）。同策略学习指所有与环境交互采样出来的轨迹立即拿去训练策略，训练完毕之后即丢弃；而异策略学习指将所有采集出来的数据存储在一个数据缓冲区中，训练策略时从缓冲区中采样出若干数据组进行训练。

深度强化学习问题的抽象凝练与平台整体设计

通过以上描述，现有强化学习算法可以被抽象成如下若干模块：

1. 数据缓冲区（Buffer）：无论是同策略学习，还是异策略学习方法，均需要将智能体与环境交互的数据进行封装与存储。例如在 DQN [[MKS+15]] 算法实现中，需要使用重放缓冲区（Replay Buffer）进行相应的数据处理，因此对数据存储的实现是平台底层不可或缺的一部分。

   更进一步，可以将同策略学习算法与异策略学习算法的数据存储用数据缓冲区（Buffer）进行统一：异策略学习算法是将缓冲区数据每次采样出一部分，而同策略学习算法可以看做一次性将缓冲区中所有数据采集出来并删除。

2. 策略（Policy）：策略是智能体决策的核心部分，将其形式化表示为

$$\pi_\theta : (o_t, h_t) \mapsto (a_t, h_{t+1}, p_t) \tag{1.1}$$

   其中 h_t 是 t 时刻策略的隐藏层状态，通常用于循环神经网络（Recurrent Neural Network，RNN）的训练；p_t 是 t 时刻策略输出的中间值，以备后续训练时使用。

此外不同策略在训练的时候所需要采样的数据模式不同，比如在计算 n 步回报的时候需要从数据缓冲区中采样出连续 n 帧的数据信息进行计算，因此策略需要有一个专门和数据缓冲区进行交互的接口。

策略中还包含模型（Model），包括表格模型、神经网络策略模型、环境模型等。模型可直接与策略进行交互，而不必和其他部分相互耦合。

3. 采集器（Collector）：采集器定义了策略与环境（Env）交互的过程。策略在与一个或多个环境交互的过程中会产生一定的数据，由采集器进行收集并存放至数据缓冲区中；在训练策略的时候由采集器从数据缓冲区中采样出数据并进行封装。

   在多智能体的情况下，采集器可以承担多个策略之间的交互，并分别存储至不同的数据缓冲区中。

4. 训练器（Trainer）：训练器是平台最上层的封装，定义了整个训练过程，与采集器和策略的学习函数进行交互，包含同策略学习与异策略学习两种训练模式。

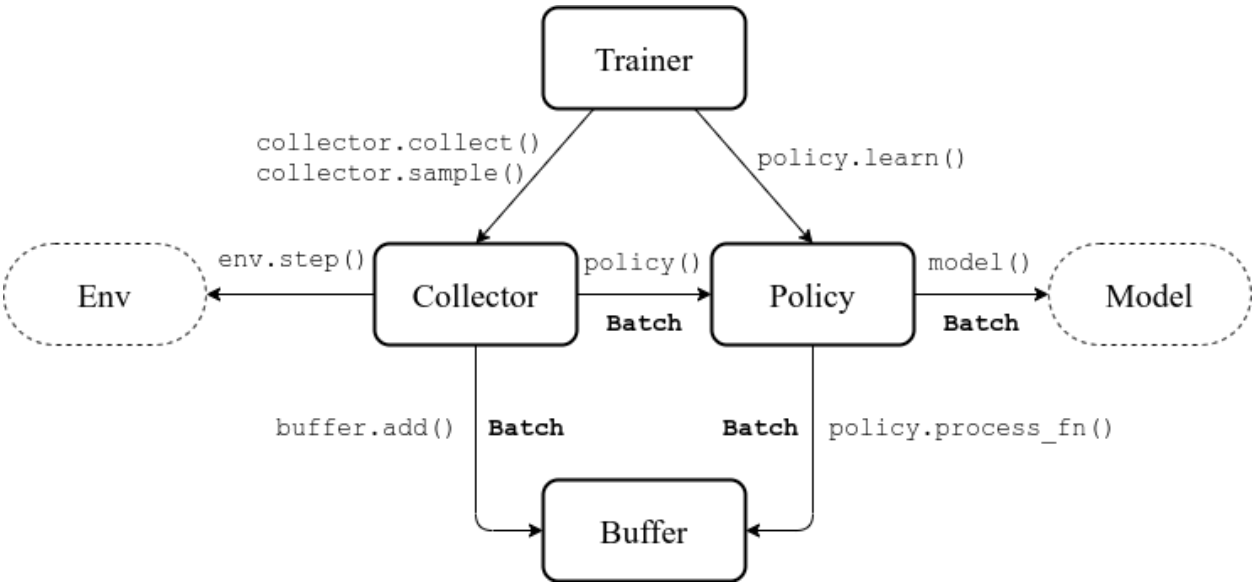


图 4: 图 2.2：深度强化学习算法模块抽象凝练，暨天授平台总体设计

图 2.2 较为直观地描述了上述抽象出的若干模块相互之间的调用关系。其中“Batch”（数据组）为模块之间传递数据信息的封装。平台的整体架构即按照该抽象模式进行设计，其中中间四个模块为平台核心模块。

平台实现

数据组（Batch）

数据组是平台内部各个模块之间传递数据的数据结构。它支持任意关键字初始化、对任意元素进行修改，以及嵌套调用和格式化输出的功能。如果数据组内各个元素值的第 0 维大小相等，还可支持切分（split）操作，从而方便地将一组大数据按照固定的大小拆分之后送入策略模块中处理。

平台的内部实现对数据组保留了如下 7 个关键字：

- `obs`：t 时刻的观测值 o_t；
- `act`：t 时刻策略采取的动作值 a_t；
- `rew`：t 时刻环境反馈的奖励值 r_t；
- `done`：t 时刻环境结束标识符 $d_t \in \{0, 1\}$，0 为未结束，1 为结束；
- `obs_next`：$t + 1$ 时刻的观测值 o_{t+1}；

- `info`：t 时刻环境给出的环境额外信息 i_t，以字典形式存储；
- `policy`：t 时刻策略在计算过程中产生的数据 p_t，可参考 (1.1)。

数据缓冲区（Buffer）

数据缓冲区存储了策略与环境交互产生的一系列数据，并且支持从已存储数据中采样出固定大小的数据组进行策略学习。底层数据结构主要采用 NumPy 数组进行存储，能够加快存储效率。

同数据组一样，数据缓冲区同样保留了其中 7 个保留关键字，其中关键字 `info` 不改变其中的数据结构，即在 NumPy 数组中仍然使用字典格式进行存储。在采样时，如果传入大小是 0，则返回整个缓冲区中的所有数据，以支持在同略学习算法的训练需求。

目前数据缓冲区的类型有：最基本的重放缓冲区（Replay Buffer）、优先级经验重放缓冲区（Prioritized Replay Buffer）支持优先权重采样、向量化重放缓冲区（Vector Replay Buffer）能够支持任意多的环境往里面添加数据而不破坏数据的时间顺序。此外数据缓冲区还支持历史数据堆叠采样（例如给定采样时间下标 t 和堆叠帧数 n，返回堆叠的观测值 $\{o_{t-n+1}, \ldots, o_t\}$）和多模态数据存储（需要存储的数据可以是一个字典）。在将来还将会支持事后经验回放算法 [[ACR+17]]（Hindsight Experience Replay，HER）。

环境（Env）

环境接口遵循 OpenAI Gym [[BCP+16]] 定义的通用接口，即每次调用 `step` 函数时，需要输入一个动作 a_t，返回一个四元组：下一个观测值 o_{t+1}、这个时刻采取动作值 a_t 所获得的奖励 r_t、环境结束标识符 d_t、以及环境返回的其他信息 i_t。

为使所有强化学习算法支持并行环境采样，天授封装了几个不同的向量化环境类，可以单线程循环执行每个环境，也可以多线程同时执行。每次调用 `step` 函数的语义和之前定义一致，区别在于增加了一步将所有信息堆叠起来组成一个 NumPy 数组的操作，并以第 0 个维度来区分是哪个环境产生的数据。

策略（Policy）

策略是强化学习算法的核心。智能体除了需要做出决策，还需不断地学习来自我改进。通过 *深度强化学习问题的抽象凝练与平台整体设计* 中对策略的抽象描述，可以将其拆分为 4 个模块：

1. `__init__`：策略的初始化，比如初始化自定义的模型（Model）、创建目标网络（Target Network）等；
2. `forward`：从给定的观测值 o_t 中计算出动作值 a_t，在图 *2.2* 中对应 Policy 到 Model 的调用和 Collector 到 Policy 的调用；
3. `process_fn`：在获取训练数据之前和数据缓冲区进行交互，在图 *2.2* 中对应 Policy 到 Buffer 的调用；
4. `learn`：使用一个数据组进行策略的更新训练，在图 图 *2.2* 中对应 Trainer 到 Policy 的调用。
5. `post_process_fn`：使用一个 Batch 的数据进行 Buffer 的更新（比如更新 PER）；
6. `update`：最主要的接口。这个 update 函数先是从 buffer 采样出一个 batch，然后调用 process_fn 预处理，然后 learn 更新策略，然后 post_process_fn 完成一次迭代：`process_fn -> learn -> post_process_fn`。

不同算法中策略的具体实现将在 *平台支持的深度强化学习算法* 中进行详细分析讲解。

模型（Model）

模型为策略的核心部分。为了支持任意神经网络结构的定义，天授并未像其他平台一样显式地定义若干基类（比如 MLPPolicy、CNNPolicy 等），而是规定了模型与策略进行交互的接口，从而让用户有更大的自由度编写代码和训练逻辑。

模型的接口定义如下：

- 输入
  1. `obs`：观测值，可以是 NumPy 数组、torch 张量、字典、或者是其他自定义的类型；
  2. `state`：隐藏状态表示，为 RNN 使用，可以为字典、NumPy 数组、torch 张量；
  3. `info`：环境信息，由环境提供，是一个字典；
- 输出
  1. `logits`：网络的原始输出，被策略用于计算动作值；比如在 DQN [[MKS+15]] 算法中 `logits` 可以为动作值函数，在 PPO [[SWD+17]] 中如果使用对角高斯策略，则 `logits` 可以为 `(mu, sigma)` 的二元组；
  2. `state`：下一个时刻的隐藏状态，为 RNN 使用；
  3. `policy`：策略输出的中间值，会被存储至重放缓冲区中，用于后续训练时使用。

采集器（Collector）

采集器定义了策略与环境交互的过程。采集器主要包含 “collect“函数：让给定的策略和环境交互 `n_step` 步、或者 `n_episode` 轮，并将交互过程中产生的数据存储进数据缓冲区中；

采集器理论上还可以支持多智能体强化学习的交互过程，将不同的数据缓冲区和不同策略联系起来，即可进行交互与数据采样。

天授还实现了异步采集器 AsyncCollector，它支持异步的环境采样（比如环境很慢或者 step 时间差异很大）。不过 AsyncCollector 的 collect 的语义和上面 Collector 有所不同，由于异步的特性，它只能保证 ** 至少 ** `n_step` 或者 `n_episode` 地收集数据。

训练器（Trainer）

训练器负责最上层训练逻辑的控制，例如训练多少次之后进行策略和环境的交互。现有的训练器包括同策略学习训练器（On-policy Trainer）、异策略学习训练器（Off-policy Trainer）和离线策略学习训练器（Offline Trainer）。

平台未显式地将训练器抽象成一个类，因为在其他现有平台中都将类似训练器的实现抽象封装成一个类，导致用户难以二次开发。因此以函数的方式实现训练器，并提供了示例代码便于研究者进行定制化训练策略的开发。

算法伪代码与对应解释

接下来将通过一段伪代码的讲解来阐释上述所有抽象模块的应用。

```
s = env.reset()
buf = Buffer(size=10000)
agent = DQN()
for i in range(int(1e6)):
    a = agent.compute_action(s)
    s_, r, d, _ = env.step(a)
    buf.store(s, a, s_, r, d)
    s = s_
    if i % 1000 == 0:
        bs, ba, bs_, br, bd = buf.get(size=64)
        bret = calc_return(2, buf, br, bd, ...)
        agent.update(bs, ba, bs_, br, bd, bret)
```

以上伪代码描述了一个定制化两步回报 DQN 算法的训练过程。*表 2.1* 描述了伪代码的解释与上述各个模块的具体对应关系。

表 3: 表 2.1：伪代码与天授模块具体对应关系

行	伪代码	解释	对应天授平台实现
1	s = env.reset()	环境初始化	在 Env 中实现
2	buf = Buffer(size=10000)	数据缓冲区初始化	buf = ReplayBuffer(size=10000)
3	agent = DQN()	策略初始化	policy.__init__(...)
4	for i in range(int(1e6)):	描述训练过程	在 Trainer 中实现
5	a = agent.compute_action(s)	计算动作值	policy(batch, ...)
6	s_, r, d, _ = env.step(a)	与环境交互	collector.collect(...)
7	buf.store(s, a, s_, r, d)	将交互过程中产生的数据存储到数据缓冲区中	collector.collect(...)
8	s = s_	更新观测值	collector.collect(...)
9	if i % 1000 == 0:	每一千步更新策略	在 Trainer 中实现
10	bs, ba, bs_, br, bd = buf.get(size=64)	从数据缓冲区中采样出数据	collector.sample(size=64)
11	bret = calc_return(2, buf, br, bd, ...)	计算两步回报	policy.process_fn(batch, buffer, indice)
12	agent.update(bs, ba, bs_, br, bd, bret)	训练智能体	policy.learn(batch, ...)

平台外围支持

命名由来

该强化学习平台被命名为“天授”。天授的字面含义是上天所授，引申含义为与生俱来的天赋。强化学习算法是不断与环境交互进行学习，在这个过程中没有人类的干预。取名“天授”是为了表明智能体没有向所谓的“老师”取经，而是通过与环境的不断交互自学成才。*图 2.3* 展示了天授平台的标志，左侧采用渐变颜色融合了青铜文明元素，是一个大写的字母“T”，右侧是天授拼音。

图 5: 图 2.3：天授平台标志

文档教程

天授提供了一系列针对平台的文档和教程，使用 ReadTheDocs[1] 第三方平台进行自动部署与托管服务。目前部署在 https://tianshou.readthedocs.io/ 中，预览页面如图 *2.4* 所示。

单元测试

天授具有较为完善的单元测试，使用 GitHub Actions[2] 进行持续集成。在每次单元测试中，均包含代码风格测试、类型测试、功能测试、性能测试、文档测试五个部分，其中性能测试是对所有天授平台中实现的强化学习算法进行整个过程的完整训练和测试，一旦没有在规定的训练限制条件内达到能够解决对应问题的效果，则不予通过测试。

目前天授平台的单元测试代码覆盖率达到了 94%，可以在第三方网站 https://codecov.io/gh/thu-ml/tianshou 中查看详细情况。图 *2.5* 展示了天授某次单元测试的具体结果。

发布渠道

目前天授平台的发布渠道为 PyPI[3] 和 Conda[4]。用户可以通过直接运行命令

```
$ pip install tianshou
```

或者

```
$ conda install tianshou -c conda-forge
```

进行平台的安装，十分方便。图 *2.6* 显示了天授在 PyPI 平台的发布界面。

[1] https://readthedocs.org/
[2] https://help.github.com/cn/actions
[3] https://pypi.org/
[4] https://anaconda.org/anaconda/conda

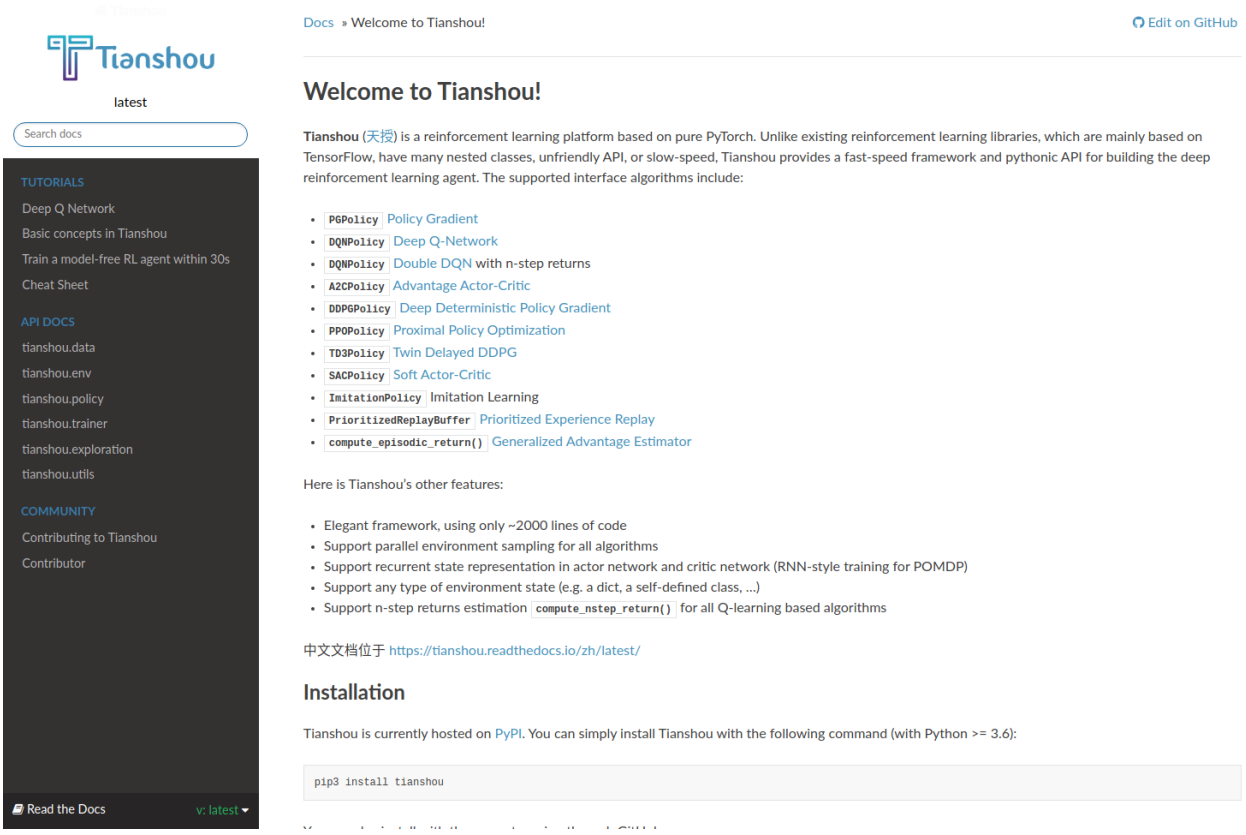


图 6: 图 2.4：天授文档页面

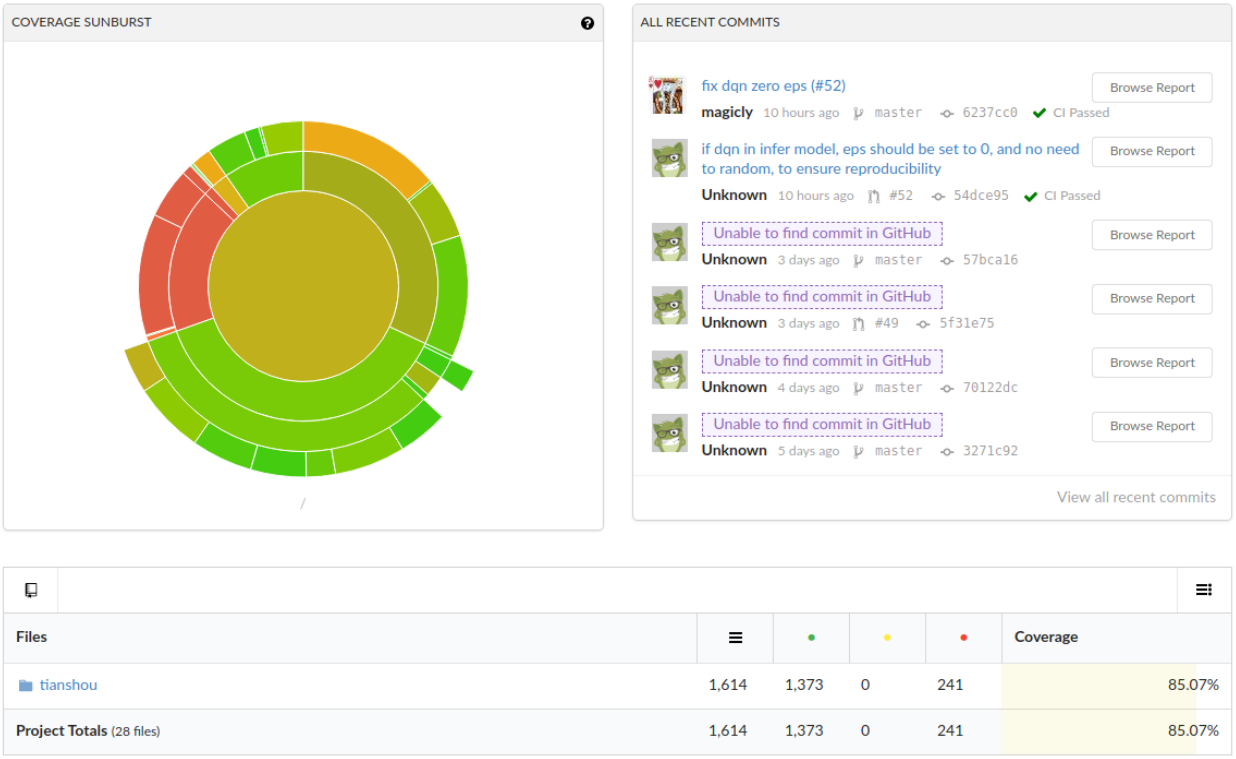


图 7: 图 2.5：天授单元测试结果

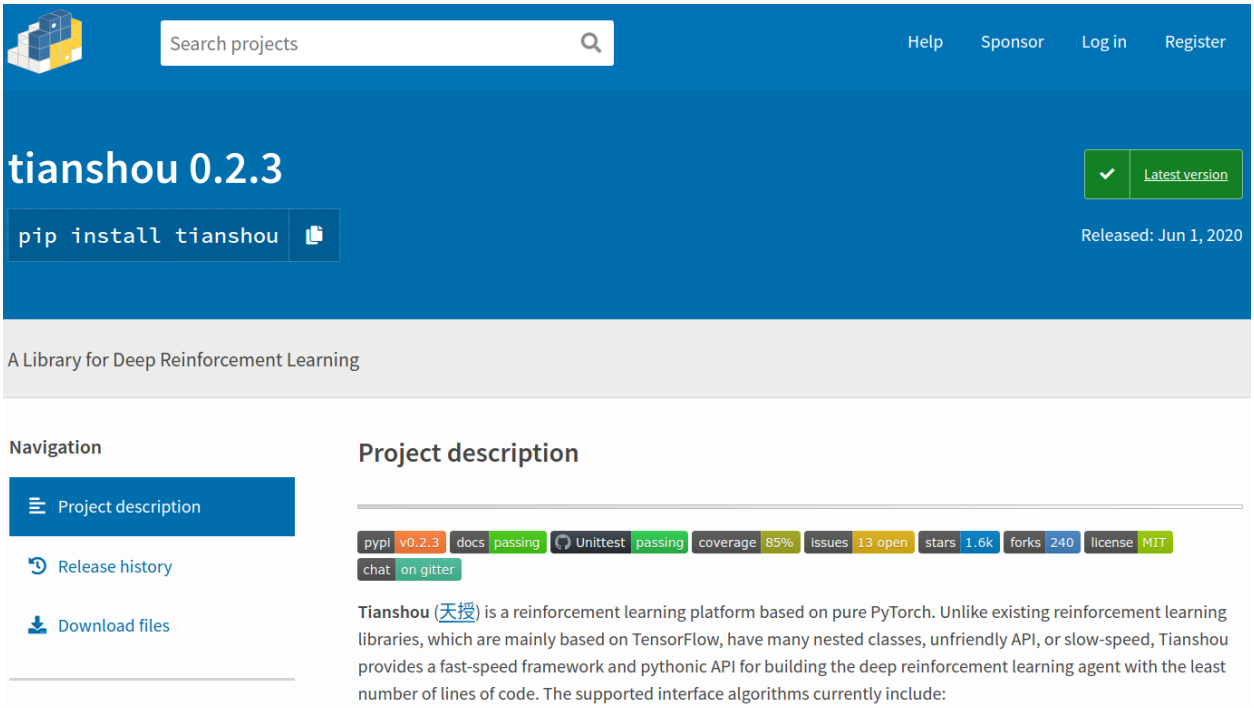


图 8: 图 2.6：天授在 PyPI 平台的发布界面

小结

本章节介绍了深度强化学习的基本定义与问题描述，将各种不同的强化学习算法进行模块化抽象，并据此阐述了平台各个模块的实现，最后简单介绍了平台的其他特点。

1.6.5 平台支持的深度强化学习算法

本章节将依次介绍天授平台所实现的强化学习算法的原理，以及这些算法在平台内部的具体实现细节。

强化学习的主要目标是让智能体学会一个能够最大化累计奖励的策略。以下仍然使用符号 $\pi_\theta(\cdot)$ 表示一个使用参数 θ 参数化的策略 π，优化目标为最大化 $J(\theta)$，定义为：

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t] = \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a) \tag{1.2}$$

其中 $d^\pi(s)$ 是使用策略 π 在马尔科夫链中达到状态 s 的概率。为力求简洁直观地表述各个算法，下文尽量不进行复杂的数学公式推导。此外，下文在描述算法时通常使用状态值 s_t，在描述具体实现时通常使用观测值 o_t。

基于策略梯度的深度强化学习算法

策略梯度（PG）

策略梯度算法（Policy Gradient [[SMSM99]]，又称 REINFORCE 算法、蒙特卡洛策略梯度算法，以下简称 PG）是一个较为直观简洁的强化学习算法。它于上世纪九十年代被提出，依靠蒙特卡洛采样直接进行对累计折扣回报的估计，并直接以公式 (1.2) 对 θ 进行求导，可推出梯度为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{1.3}$$

其中 a_t、s_t 均为具体采样值。将公式 (1.3) 反推为目标函数

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \log \pi_\theta(a_t|s_t)] \tag{1.4}$$

可以发现策略梯度算法本质上是最大化好动作的概率[1] 。因此实现 PG 算法只需求得累计回报 G_t、每次采样的数据点在策略函数中的对数概率 $\log \pi_\theta(a_t|s_t)$ 之后即可对参数 θ 进行求导，从而使用梯度上升方法更新模型参数。

一个被广泛使用的变种版本是在算法中将 G_t 减去一个基准值，在保证不改变偏差的情况下尽可能减小梯度估计的方差。比如减去一个平均值，或者是如果使用状态值函数 $V(s)$ 作为一个基准，那么实际所使用的即为优势函数 $A(s,a) = Q(s,a) - V(s)$，为动作值函数与状态值函数的差值。这将在后续描述的算法中进行使用。

策略梯度算法在天授中的实现十分简单：

- `process_fn`：计算 G_t，具体实现位于 *广义优势函数估计器（GAE）*；
- `forward`：给定 o_t 计算动作的概率分布，并从其中进行采样返回；
- `learn`：按照公式 (1.4) 计算 G_t 与动作的对数概率 $\log \pi_\theta(a_t|o_t)$ 的乘积，求导之后进行反向传播与梯度上升，优化参数 θ；
- 采样策略：使用同策略的方法进行采样。

[1] 这个视频详细地讲解了策略梯度算法的推导过程：https://youtu.be/XGmd3wcyDg8

优势动作评价（A2C）

优势动作评价算法（Advantage Actor-Critic [[MBM+16]]，又被译作优势演员-评论家算法，以下简称 A2C）是对策略梯度算法的一个改进。简单来说，策略梯度算法相当于 A2C 算法中评价网络输出恒为 0 的版本。该算法从公式 (1.4) 改进如下：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\hat{A}(s_t, a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

其中 $\hat{A}(s_t, a_t)$ 为估计的优势函数，具体定义以及实现见 *广义优势函数估计器（GAE）*。为了让评价网络的输出尽可能接近真实的状态值函数，在优化过程中还加上了对应的均方误差项；此外标准的实现中还有关于策略分布的熵正则化项。因此汇总的 A2C 目标函数为：

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\hat{A}(s_t, a_t)\log \pi_\theta(a_t|s_t) - c_1(\hat{V}(s_t) - G_t)^2 + c_2 H(\pi_\theta(\cdot)|s_t)\right] \tag{1.5}$$

其中 $\hat{V}(s_t)$ 为评价网络输出的状态值函数，c_1, c_2 是前述两项的对应超参数。

A2C 最大的特点就是支持同步的并行采样训练，但由于天授平台支持所有算法的并行环境采样，此处不再赘述。此外 A2C 相比于异步策略执行版本 A3C 而言，避免了算法中策略执行不一致的问题，具有更快的收敛速度。

A2C 算法在天授中的实现如下：

- `process_fn`：计算 $\hat{A}(s_t, a_t)$，具体实现位于 *广义优势函数估计器（GAE）*；
- `forward`：和策略梯度算法一致，给定观测值 o_t，计算输出的输出策略的概率分布，并从中采样；
- `learn`：按照公式 (1.5) 计算目标函数并求导更新参数；
- 采样策略：使用同策略的方法进行采样。

近端策略优化（PPO）

近端策略优化算法（Proximal Policy Optimization [[SWD+17]]，以下简称 PPO）是信任区域策略优化算法（Trust Region Policy Optimization [[SLA+15]]，TRPO）的简化版本。由于策略梯度算法对超参数较为敏感，二者对策略的更新进行了一定程度上的限制，避免策略性能在参数更新前后产生剧烈变化，从而导致采样效率低下等问题。

PPO 算法通过计算更新参数前后两次策略的比值来确保这个限制。具体目标函数为

$$J^{\text{CLIP}}(\theta) = \mathbb{E}_{\pi_\theta}\left[\min\left(r(\theta)\hat{A}_{\theta_{\text{old}}}(s_t, a_t), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s_t, a_t)\right)\right]$$

其中 $\hat{A}(\cdot)$ 表示估计的优势函数，因为真实的优势函数无法从训练过程所得数据中进行精确计算；$r(\theta)$ 是重要性采样权重，定义为新策略与旧策略的概率比值

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

函数 $\text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)$ 将策略的比值 $r(\theta)$ 限制在 $[1-\epsilon, 1+\epsilon]$ 之间，从而避免了策略性能上的剧烈变化。在将 PPO 算法运用在动作评价（Actor-Critic）架构上时，与 A2C 算法类似，目标函数通常会加入状态值函数项与熵正则化项

$$J(\theta) = \mathbb{E}_{\pi_\theta}[J^{\text{CLIP}}(\theta) - c_1(\hat{V}(s_t) - G_t)^2 + c_2 H(\pi_\theta(\cdot)|s_t)] \tag{1.6}$$

其中 c_1, c_2 为两个超参数，分别对应状态值函数估计与熵正则化两项。

天授中的 PPO 算法实现大致逻辑与 A2C 十分类似：

- `process_fn`：计算 $\hat{A}(s_t, a_t)$ 与 G_t，具体实现位于 *广义优势函数估计器（GAE）*；

- `forward`：按照给定的观测值 o_t 计算概率分布，并从中采样出动作 a_t；
- `learn`：重新计算每个数据组所对应的对数概率，并按照公式 (1.6) 进行目标函数的计算；
- 采样策略：使用同策略的方法进行采样。

广义优势函数估计器（GAE）

广义优势函数估计器（Generalized Advantage Estimator [[SML+16]]，以下简称 GAE）是将以上若干种策略梯度算法的优势函数的估计 $\hat{A}(s_t, a_t)$ 进行形式上的统一。一般而言，策略梯度算法的梯度估计都遵循如下形式：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\Big]$$

其中 Ψ_t 具有多种形式，比如 PG 中为 $\Psi_t = \sum_{i=t}^{\infty} r_i$，即累计回报函数；A2C 中为 $\Psi_t = \hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$；PPO 中是 $\Psi_t = \hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$，其中 δ_t 是时序差分误差项（Temporal Difference Error，TD Error），$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$。GAE 将上述若干种估计形式进行统一如下：

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} = \sum_{l=0}^{\infty} (\gamma\lambda)^l (r_t + \gamma V(s_{t+l+1}) - V(s_{t+l})) \tag{1.7}$$

其中 GAE$(\gamma, 0)$ 的情况为 $\hat{A}_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$，为 1 步时序差分误差，GAE$(\gamma, 1)$ 的情况为 $\hat{A}_t = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t)$，即为 A2C 中的估计项。PG 中的估计项即为 A2C 中 $V(s_t)$ 恒为 0 的特殊情况。

天授中 GAE 实现与其他平台有一些不同之处。比如在 OpenAI Baselines [[DHK+17]] 的实现中，对每个完整轨迹的最后一帧进行特殊判断处理。与此不同，天授使用轨迹中每项的下一时刻观测值 o_{t+1} 批量计算状态值函数，避免了特殊判断。天授的 GAE 实现将大部分操作进行向量化，并且支持同时计算多个完整轨迹的 GAE 函数，比 Baselines 使用正常 Python 循环语句的方式显著提高了运行速度。

基于 Q 价值函数的深度强化学习算法

深度 Q 网络（DQN）

深度 Q 网络算法（Deep Q Network [[MKS+15]]，以下简称 DQN）是强化学习算法中最经典的算法之一，它在 Atari 游戏中表现一鸣惊人，由此掀起了深度强化学习的新一轮浪潮。DQN 算法核心是维护 Q 函数并使用它进行决策。具体而言，$Q^\pi(s, a)$ 为在该策略 π 下的动作值函数；每次到达一个状态 s_t 之后，遍历整个动作空间，将动作值函数最大的动作作为策略：

$$a_t = \arg\max_a Q^\pi(s_t, a)$$

其动作值函数的更新采用贝尔曼方程（Bellman Equation）进行迭代

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(r_t + \gamma \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)) \tag{1.8}$$

其中 α 为学习率。通常在简单任务上，使用全连接神经网络来拟合 Q^π，但是在稍微复杂一点的任务上如 Atari 游戏，会使用卷积神经网络进行由图像到值函数的映射拟合，这也是深度 Q 网络中“深度”一词的由来。由于这种表达形式只能处理有限个动作值，因此 DQN 通常被用在离散动作空间任务中。

为了避免陷入局部最优解，DQN 算法通常采用 ϵ-贪心方法进行策略探索，即每次有 $\epsilon \in [0, 1]$ 的概率输出随机策略，$1 - \epsilon$ 的概率输出使用动作值函数估计的最优策略；此外通常把公式 (1.8) 中 $r_t + \gamma \max_a Q^\pi(s_{t+1}, a)$ 一项称作目标动作值函数 Q_{target}，它还可以拓展成不同的形式，比如 n 步估计：

$$Q_{\text{target}}^n(s_t, a_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q^\pi(s_{t+n}, a) \tag{1.9}$$

天授中的 DQN 算法实现如下：

- `process_fn`：使用公式 (1.9) 计算目标动作函数，与重放缓冲区交互进行计算；
- `forward`：给定观测值 o_t，输出每个动作对应的动作值函数 $Q(o_t, \cdot)$，并使用 ϵ-贪心算法添加噪声，输出动作 a_t；
- `learn`：使用公式 (1.8) 进行迭代，在特定时刻可调整 ϵ-贪心算法中的 ϵ 值；
- 采样策略：使用异策略的方法进行采样。

双网络深度 Q 学习（DDQN）

双网络深度 Q 学习算法（Double DQN [[vHGS16]]，以下简称 DDQN）是 DQN 算法的重要改进之一。由于在公式 (1.8) 中使用同一个动作值函数进行对目标动作值函数的估计，会导致策略网络产生过于乐观的估计，从而降低了算法的采样效率。DDQN 算法将动作评估与动作选择进行解耦，从而减少高估所带来的负面影响。它将公式 (1.8) 中的目标动作值函数加以改造如下

$$Q_{\text{target}}(s_t, a_t) = r_t + \gamma Q^{\pi_{\text{old}}}\left(s_{t+1}, \arg\max_a Q^{\pi}(s_{t+1}, a)\right) \tag{1.10}$$

其中 $Q^{\pi_{\text{old}}}$ 是目标网络（Target Network），为策略网络 Q^{π} 的历史版本，专门用来进行动作评估。公式 (1.10) 同样可以和公式 (1.9) 进行结合，推广到 n 步估计的情况，此处不再赘述。

由于 DDQN 与 DQN 仅有细微区别，因此在天授的实现中将二者封装在同一个类中，改动如下：

- `process_fn`：按照公式 (1.10) 计算目标动作函数；
- `learn`：在需要的时候更新目标网络的参数。

优先级经验重放（PER）

优先级经验重放（Prioritized Experience Replay [[SQAS16]]，以下简称 PER）是 DQN 算法的另一个重要改进。该算法也可应用在之后的 DDPG 算法族中。其核心思想是，根据策略网络输出的动作值函数 $Q^{\pi}(s_t, a_t)$ 与实际采样估计的动作值函数 $Q_{\text{target}}(s_t, a_t)$ 的时序差分误差来给每个样本不同的采样权重，将误差更大的数据能够以更大的概率被采样到，从而提高算法的采样与学习效率。

PER 的实现不太依赖于算法层的改动，比较和底层的重放缓冲区相关。相关改动如下：

- 算法层：加入一个接口，传出时序差分误差，作为优先经验重放缓冲区的更新权重；
- 数据层：新建优先经验重放缓冲区类，继承自重放缓冲区类，修改采样函数，并添加更新优先值权重的函数。

综合 Q 价值函数与策略梯度的深度强化学习算法

深度确定性策略梯度（DDPG）

深度确定性策略梯度算法（Deep Deterministic Policy Gradient [[LHP+16]]，以下简称 DDPG）是一种同时学习确定性策略函数 $\pi_\theta(s)$ 和动作值函数 $Q^{\pi}(s, a)$ 的算法。它主要解决的是连续动作空间内的策略训练问题。在 DQN 中，由于常规的 Q 函数只能接受可数个动作，因此无法拓展到连续动作空间中。

DDPG 算法假设动作值函数 $Q(s, a)$ 在连续动作空间中是可微的，将动作值 a 用一个函数 $\pi_\theta(s)$ 拟合表示，并将 $\pi_\theta(s)$ 称作动作网络，$Q^{\pi}(s, a)$ 称作评价网络。DDPG 算法评价网络的更新部分与 DQN 算法类似，动作网络的更新根据确定性策略梯度定理 [[SLH+14]]，直接对目标函数 $Q^{\pi}(s, \pi_\theta(s))$ 进行梯度上升优化即可。

为了更好地进行探索，原始 DDPG 算法添加了由 Ornstein-Uhlenbeck 随机过程[2] 产生的时间相关的噪声项，但在实际测试中，高斯噪声可以达到与其同样的效果 [[FvHM18]]；DDPG 还采用了目标网络以稳定训练过程，

[2] https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process

对目标动作网络和目标评价网络进行参数软更新，即 $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$，以 τ 的比例将新网络的权重 θ 更新至目标网络 θ' 中。

天授对 DDPG 算法的实现如下：

- `process_fn`：和 DQN 算法类似，其中动作 a 不进行全空间遍历，而是以动作网络的输出作为参考标准；
- `forward`：给定观测值 o_t，输出动作 $a_t = \pi_\theta(o_t)$，并添加噪声项；
- `learn`：分别计算贝尔曼误差项和 $Q^\pi(s, \pi_\theta(s))$ 并分别优化，之后软更新目标网络的参数；
- 采样策略：使用异策略的方法进行采样。

双延迟深度确定性策略梯度（TD3）

双延迟深度确定性策略梯度算法（Twin Delayed DDPG [[FvHM18]]，以下简称 TD3）是 DDPG 算法的改进版本。学习动作值函数 Q 的一系列方法一直以来都有过度估计的问题，DDPG 也不例外。TD3 算法做了如下几点对 DDPG 进行改进：

- 截断双网络 Q 学习：截断双网络 Q 学习使用两个动作值网络，取二者中的最小值作为动作值函数 Q 的估计，从而有利于减少过度估计：

$$Q_{\text{target}_i} = r + \min_{j=1,2} Q^\pi_{\phi_j}(s', \pi_\theta(s')),\ i = 1, 2$$

- 动作网络延迟更新：相关实验结果表明，同步训练动作网络和评价网络，却不使用目标网络，会导致训练过程不稳定；但是仅固定动作网络时，评价网络往往能够收敛到正确的结果。因此 TD3 算法以较低的频率更新动作网络，较高频率更新评价网络，通常每两次更新评价网络时，进行一次策略更新。
- 平滑目标策略：TD3 算法在动作中加入截断高斯分布产生的随机噪声，避免策略函数 $\pi_\theta(s)$ 陷入 Q 函数的极值点，从而更有利于收敛：

$$Q_{\text{target}} = r + \gamma Q^\pi(s', \pi_\theta(s') + \epsilon)$$
$$\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c)$$

与 DDPG 算法类似，天授在 TD3 的实现中继承了 DDPG 算法，只修改了 `learn` 部分，按照上述三点一一实现代码。

软动作评价（SAC）

软动作评价算法（Soft Actor-Critic [[HZH+18]]，以下简称 SAC）是基于最大熵强化学习理论提出的一个算法。SAC 算法同时具备稳定性好和采样效率高的优点，容易实现，同时融合了动作评价框架、异策略学习框架和最大熵强化学习框架，因此成为强化学习算法中继 PPO 之后的标杆算法。

SAC 的算法结构和 TD3 也十分类似，同样拥有一个动作网络和两个评价网络。单从最终推导得到的式子来看，和 TD3 的最大差别是在求目标动作值函数的时候，最后一项加上了较为复杂的熵正则化项，其余的实现十分类似。具体的推导可以在原论文中找到。

由于 SAC 的实现和 TD3 十分类似，故此处不再对其进行详细阐述。

部分可观测马尔科夫决策过程的训练

在实际场景中，智能体往往难以观测到环境中所有的信息，只能观测到状态 s 的一个子集 o 进行决策，这种场景被称作部分可观测马尔科夫决策过程（Partially Observable Markov Decision Process，简称 POMDP）。

POMDP 在深度强化学习领域通常有两种解决方案：（1）将过去一段时间内的信息（如过去的观测值、过去的动作和奖励）添加到当前状态中，按照常规方式进行处理；（2）将过去的信息利用循环神经网络（RNN）存储到中间状态中，可以传给后续状态进行使用。

第一种方法只需在重放缓冲区中添加时序采样功能，比如待采样下标是 t，需要采样连续 n 帧，那么在重放缓冲区中进行一定设置，返回观测值 $\{o_{t-n+1}, \ldots, o_{t-1}, o_t\}$，剩下的过程和正常的强化学习训练过程无异。第二种方法需要在第一种方法的基础上，在所有和神经网络相关的接口中添加对中间状态的支持。天授已经支持上述两种方法的实现。

模仿学习

模仿学习（Imitation Learning）更偏向于监督学习与半监督学习的范畴。它的核心思想是学习已有的数据，尽可能地还原产生这些数据的原始策略。比如给定一些 t 时刻的状态与动作数据对 (s_t, a_t)，那么可以使用神经网络来回归映射 $\mathcal{F}: \mathcal{S} \to \mathcal{A}$，从而进行模仿学习。更进一步地，还有逆强化学习（Inverse Reinforcement Learning [[FLA16]]，IRL）和生成式对抗模仿学习（Generative Adversarial Imitation Learning [[HE16]]，GAIL）等算法。

目前天授平台实现了最基本的模仿学习算法，具体实现如下：

- 连续动作空间：将其看作回归任务，直接对给定的动作进行回归；
- 离散动作空间：将其看作分类任务，最大化采取给定动作的概率；
- 采样策略：使用参考策略和异策略方法进行不断地采样补充数据。

小结

本章节介绍了深度强化学习算法的原理以及在天授平台上的具体实现，包括了 9 种免模型强化学习算法、循环神经网络模型训练和模仿学习。

1.6.6 平台对比评测

本章展示了天授与一些著名的强化学习算法平台对比评测的结果。

实验设定说明

我们在众多强化学习平台中选取了 5 个具有代表性的平台：RLlib [[LLN+18]]、OpenAI Baselines [[DHK+17]]、PyTorch-DRL [[Chr19]]、Stable-Baselines [[HRE+18]]（以下简写为 SB）、rlpyt [[SA19]]，选取评测的平台列表和原因如表 *4.1* 所示。

表 4: 表 4.1：选取对比评测的平台一览。其中，评测 ID 为该平台发布的最新版本名称，若无已发布版本，则选取最新的提交记录 ID。

平台名称	评测 ID	选取原因
RLlib [[LLN+18]]	`0.8.5`	GitHub 深度强化学习平台星标数目最多，算法实现全面
Baselines [[DHK+17]]	`ea25b9e`	GitHub 深度强化学习平台星标数目第二多
PyTorch-DRL [[Chr19]]	`49b5ec0`	GitHub PyTorch 深度强化学习平台星标数目第一多，算法实现全面
Stable-Baselines [[HRE+18]]	`2.10.0`	Baselines 的改进版本，算法实现全面，提供了一系列调优的参数
rlpyt [[SA19]]	`668290d`	GitHub PyTorch 深度强化学习平台星标数目第四多，算法实现全面
天授	`57bca16`	-

功能对比

接下来将从算法支持、模块化、定制化、单元测试和文档教程这五个维度来对包括天授在内的 6 个深度强化学习平台进行功能维度上的对比评测。

算法支持

深度强化学习算法主要分为免模型强化学习（MFRL）、基于模型的强化学习（MBRL）、多智能体学习（MARL）、模仿学习（IL）等。此外根据所解决的问题分类，还可分为马尔科夫决策过程（MDP）和部分可观测马尔科夫决策过程（POMDP），其中 POMDP 要求策略网络支持循环神经网络（RNN）的训练。如今研究者们使用最多的是免模型强化学习算法，以下会对其详细对比。

免模型强化学习算法

免模型强化学习算法主要分为基于策略梯度的算法、基于价值函数的算法和二者结合的算法。现在深度强化学习社区公认的强化学习经典算法有：（1）基于价值函数：DQN [[MKS+15]] 及其改进版本 Double-DQN [[vHGS16]]（DDQN）、DQN 优先级经验重放 [[SQAS16]]（PDQN）；（2）基于策略梯度：PG [[SMSM99]]、A2C [[MBM+16]]、PPO [[SWD+17]]；（3）二者结合：DDPG [[LHP+16]]、TD3 [[FvHM18]]、SAC [[HZH+18]]。

各个平台实现算法的程度如*表 4.2* 所示。可以看出，大部分平台支持的算法种类是全面的，有些平台如 Baselines 支持的算法类型并不全面。天授平台支持所有的这些算法。

表 5: 表 4.2：各平台支持的免模型深度强化学习算法一览

平台与算法	DQN	DDQN	PDQN	PG	A2C	PPO	DDPG	TD3	SAC	总计
RLlib	√	√	√	√	√	√	√	√	√	**9**
Baselines	√	×	√	×	√	√	√	×	×	5
PyTorch-DRL	√	√	√	√	√	√	√	√	√	**9**
SB	√	√	√	×	√	√	√	√	√	8
rlpyt	√	√	√	√	√	√	√	√	√	**9**
天授	√	√	√	√	√	√	√	√	√	**9**

其他类型强化学习算法

其他类型的强化学习算法包括基于模型的强化学习（MBRL）、多智能体强化学习（MARL）、元强化学习（MetaRL）、模仿学习（IL）。表 4.3 列出了各个平台的支持情况。可以看出，少有平台支持所有这些类型的算法。天授支持了模仿学习，但值得一提的是，基于模型的强化学习算法和多智能体强化学习算法都可以在现有的平台接口上完整实现。我们正在努力实现天授平台的 MBRL 和 MARL 算法中。

表 6: 表 4.3：各平台支持的其他类型强化学习算法一览

平台与算法类型	MBRL	MARL	MetaRL	IL
RLlib	√	√	×	×
Baselines	×	×	×	√
PyTorch-DRL	×	×	×	×
SB	×	×	×	√
rlpyt	×	×	×	×
天授	×	×	×	√

循环状态策略

针对不完全信息观测的马尔科夫决策过程（POMDP），通常有两种处理方式：第一种是直接当作完全信息模式处理，但可能会导致一些诸如收敛性难以保证的问题；第二种是在智能体中维护一个内部状态，具体而言，将循环神经网络模型（RNN）融合到策略网络中。表 4.4 列出了各个平台对循环神经网络的支持程度。从表 4.4 中可以看出，部分平台对 RNN 的支持程度并不大。天授平台中所有算法均支持 RNN 网络，还支持获取历史状态、历史动作和历史奖励，以及其他用户或者环境定义的变量的历史记录。

表 7: 表 4.4：各平台对 RNN 的支持

平台	RNN
RLlib	√
Baselines	×
PyTorch-DRL	×
SB	×
rlpyt	√
天授	√

并行环境采样

最初的强化学习算法仅是单个智能体和单个环境进行交互，这样的话采样效率较低，因为每一次网络前向都只能以单个样本进行计算，无法充分利用批处理加速的优势，从而导致了强化学习即使在简单场景中训练速度仍然较慢的问题。解决的方案是并行环境采样：智能体每次与若干个环境同时进行交互，将神经网络的前向数据量加大但又不增加推理时间，从而做到采样速率是之前的数倍。表 4.5 显示了各个平台、各个算法支持的并行环境采样的情况。从表 4.5 中可以看出，只有 RLlib、rlpyt、天授全面地支持了各种算法的并行环境采样功能，剩下的平台要么缺失部分算法实现、要么缺失部分算法的并行环境采样功能。这对于强化学习智能体的训练而言，性能方面可能会大打折扣。

表 8: 表 4.5：各平台各免模型深度强化学习算法支持并行环境采样情况一览

平台与算法	DQN	DDQN	PDQN	PG	A2C	PPO	DDPG	TD3	SAC
RLlib	√	√	√	√	√	√	√	√	√
Baselines	×	-	×	-	√	√	√	-	-
PyTorch-DRL	×	×	×	×	√	√	×	×	×
SB	×	√	×	-	√	√	√	×	×
rlpyt	√	√	√	√	√	√	√	√	√
天授	√	√	√	√	√	√	√	√	√

注：“-”表示算法未实现

模块化

模块化的强化学习算法框架能够让开发者以更少的代码量来更简单地实现新功能，在增加了代码的可重用性的同时也减少了出错的可能性。*表 4.6* 列出了各个平台模块化的详细情况。从表中可以看出，除 Baselines、Stable-Baselines 和 PyTorch-DRL 三个框架外，其余的平台都做到了模块化。天授平台并没有在训练策略上做完全的模块化，因为在训练策略模块化虽然会节省代码，但是会使得用户难以二次修改代码进行开发。天授为了能够让开发者有更好的体验，在这二者之中做了折中：提供了一个定制化的训练策略函数，但不是必须的。用户可以利用天授提供的接口，和正常写强化学习代码一样，自由地编写所需训练策略。

表 9: 表 4.6：各平台模块化功能实现一览，其中：（1）算法实现模块化，指实现强化学习算法的时候遵循一套统一的接口；（2）数据处理模块化，指将内部数据流进行封装存储；（3）训练策略模块化，指由专门的类或函数来处理如何训练强化学习智能体。

平台与模块化	算法实现	数据处理	训练策略
RLlib	√	√	√
Baselines	×	×	×
PyTorch-DRL	部分模块化	×	√
SB	×	×	×
rlpyt	√	√	√
天授	√	√	部分模块化

代码复杂度与定制化训练环境

强化学习平台除了具有作为社区研究者中复现其他算法结果的作用之外，还承担在新场景、新任务上的运用和新算法的开发的作用，此时一个平台是否具有清晰简洁的代码结构、是否支持二次开发、是否能够方便地运用于新的任务（比如多模态环境）上，就成为一个衡量平台易用性的一个标准。*表 4.7* 总结了各平台在代码复杂度与是否可定制化训练环境两个维度的测试结果，其中前者采用开源工具 cloc[1] 进行代码统计，除去了测试代码和示例代码；后者采用 Mujoco 环境中 FetchReach-v1 任务进行模拟测试，其观测状态为一个字典，包含三个元素。此处使用这个任务来模拟对定制化多模态环境的测试，凡是报异常错误或者直接使用装饰器 `gym.wrappers.FlattenObservation()` 对观测值进行数据扁平化处理的平台，都不被认为对定制化训练环境做到了很好的支持。可以看出，天授在易用性的这两个评价层面上相比其他平台都具有十分明显的优势，使用精简的代码却能够支持更多需求。

[1] GitHub 地址：https://github.com/AlDanial/cloc

表 10: 表 4.7：各平台易用性一览，代码复杂度一栏数据格式为 Python 文件数/代码行数

平台与易用性	代码复杂度	环境定制化	文档	教程
RLlib	250/24065	√	×	√
Baselines	110/10499	×	×	×
PyTorch-DRL	55/4366	×	×	×
SB	100/10989	×	√	√
rlpyt	243/14487	×	√	×
天授	**29/2141**	√	√	√

文档教程

文档与教程对于平台的易用性而言具有十分重要的意义。表 *4.7* 列举出了各个平台的 API 接口文档与教程的情况。尽管天授的文档与 Stable-Baselines 相比还有待提高，但相比其它平台而言仍然提供了丰富的教程，供使用者使用。

单元测试与覆盖率

单元测试对强化学习平台有着十分重要的作用：它在本身就难以训练的强化学习算法上加上了一个保险栓，进行代码正确性检查，避免了一些低级的错误发生，同时还保证了一些基础算法的可复现性。表 *4.8* 从代码风格测试、基本功能测试、训练过程测试、代码覆盖率这些维度展示了各个平台所拥有的单元测试。大部分平台满足代码风格测试和基本功能测试要求，只有约一半的平台有对完整训练过程进行测试（此处指从智能体的神经网络随机初始化至智能体完全解决问题），以及显示代码覆盖率。综合来看，天授平台是其中表现最好的。

表 11: 表 4.8：各平台单元测试情况一览

平台与单元测试	PEP8 代码风格	基本功能	训练过程	代码覆盖率
RLlib	√	√	部分	暂缺 *
Baselines	√	√	部分	53% **
PyTorch-DRL	不遵循 + 无测试	√	完整	62% **
SB	√	√	部分	85%
rlpyt	×	部分	部分	22%
天授	√	√	完整	94%

注：*：由于 RLlib 平台单元测试过于复杂，代码覆盖率并未集成至单元测试中，因此无法获取代码覆盖率；**：手动在其单元测试脚本中添加代码覆盖率开启选项，并在 Travis CI 第三方测试平台中获取测试结果。

基准性能测试

本章节将各个强化学习平台在 OpenAI Gym [[BCP+16]] 简单环境中进行性能测试。实验运行环境配置参数如表 *4.9* 所示。所有运行实验耗时取纯 CPU 和 CPU+GPU 混合使用的这两种运行状态模式下的时间的最优值。为减小测试结果误差，每组实验将会以不同的随机种子运行 5 次。

表 12: 表 4.9：实验运行环境参数

类型	参数
操作系统	Ubuntu 18.04
内核	5.3.0-53-generic
CPU	Intel i7-8750H (12) @ 4.100GHz
GPU	NVIDIA GeForce GTX 1060 Mobile
RAM	31.1 GiB DDR4
Disk	SAMSUNG MZVLB512HAJQ-000L2 SSD
NVIDIA 驱动版本	440.64.00
CUDA 版本	10.0
Python 版本	3.6.9
TensorFlow 版本	1.14.0
PyTorch 版本	1.4.0（PyTorch-DRL）或 1.5.0

离散动作空间免模型强化学习算法测试

离散动作空间的一系列强化学习任务中，最简单的任务是 OpenAI Gym 环境中的 CartPole-v0 任务：该任务要求智能体操纵小车，使得小车上的倒立摆能够保持垂直状态，一旦偏离超过一定角度、或者小车位置超出规定范围，则认为游戏结束。该任务观测空间为一个四维向量，动作空间取值为 0 或 1，表示在这个时间节点内将小车向左或是向右移动。图 *4.1* 对该任务进行了可视化展示。

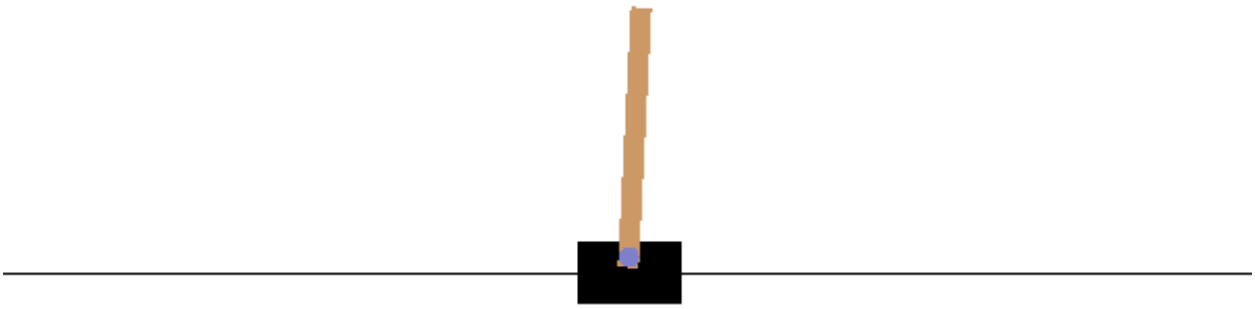


图 9: 图 4.1：CartPole-v0 任务可视化

该任务选取 PG [[SMSM99]]、DQN [[MKS+15]]、A2C [[MBM+16]]、PPO [[SWD+17]] 四种经典的免模型强化学习算法进行评测。根据 Gym 中说明的规则，每个算法必须在连续 100 次任务中，总奖励值取平均之后大

于等于 195 才算解决了这个任务。各个平台不同算法解决任务的测试结果如表 *4.10* 所示，原始数据见附表 *1*。天授与其他平台相比，有着令人惊艳的性能，尤其是 PG、DQN 和 A2C 算法，能够在平均不到 10 秒的时间内解决该问题。

表 13: 表 4.10：CartPole-v0 测试结果，运行时间单位为秒

平台与算法	PG	DQN	A2C	PPO
RLlib	19.26 ± 2.29	28.56 ± 4.60	57.92 ± 9.94	44.60 ± 17.04
Baselines	-	×	×	×
PyTorch-DRL *	×	31.58 ± 11.30	×	**23.99 ± 9.26**
SB	-	93.47 ± 58.05	57.56 ± 12.87	34.79 ± 17.02
rlpyt	**	**	**	**
天授	**6.09 ± 4.60**	**6.09 ± 0.87**	**10.59 ± 2.04**	31.82 ± 7.76

注：“-”表示算法未实现；“×”表示五组实验完成任务平均时间超过 1000 秒或未完成任务；

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为“训练过程中连续 20 次完整游戏的平均总奖励大于等于 195”；

**：rlpyt 对于离散动作空间非 Atari 任务的支持不友好，可参考 https://github.com/astooke/rlpyt/issues/135 。

连续动作空间免模型强化学习算法测试

连续动作空间的一系列强化学习任务中，最简单的任务是 OpenAI Gym 环境中的 Pendulum-v0 任务：该任务要求智能体操控倒立摆，使其尽量保持直立，奖励值最大对应着与目标保持垂直，并且旋转速度和扭矩均为最小的状态。该任务观测空间为一个三维向量，动作空间为一个二维向量，范围为 $[-2, 2]$。图 *4.2* 对该任务进行了可视化展示。

该任务选取 PPO [[SWD+17]]、DDPG [[LHP+16]]、TD3 [[FvHM18]]、SAC [[HZH+18]] 四种经典的免模型强化学习算法进行评测。和上一小节中的评测方法类似，每个算法必须在连续 100 次任务中，总奖励值取平均值后大于等于-250 才算解决该任务。各个平台不同算法解决任务的测试结果如表 *4.11* 所示，原始数据见附表 *2*。与之前结果类似，天授平台在各个算法中的测试都取得了不错的成绩。

表 14: 表 4.11：Pendulum-v0 测试结果，运行时间单位为秒

平台与算法	PPO	DDPG	TD3	SAC
RLlib	123.62 ± 44.23	314.70 ± 7.92	149.90 ± 7.54	97.42 ± 4.75
Baselines	745.43 ± 160.82	×	-	-
PyTorch-DRL *	**	59.05 ± 10.03	57.52 ± 17.71	63.80 ± 27.37
SB	259.73 ± 27.37	277.52 ± 92.67	99.75 ± 21.63	124.85 ± 79.14
rlpyt	***	123.57 ± 30.76	113.00 ± 13.31	132.80 ± 21.74
天授	**16.18 ± 2.49**	**37.26 ± 9.55**	**44.04 ± 6.37**	**36.02 ± 0.77**

注：“-”表示算法未实现；“×”表示五组实验完成任务平均时间超过 1000 秒或未完成任务；

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为“训练过程中连续 20 次完整游戏的平均总奖励大于等于-250”；

**：PyTorch-DRL 中的 PPO 算法在连续动作空间任务中会报异常错误；

***：rlpyt 并未提供使用 PPO 算法的任何示例代码，经尝试无法成功跑通。

图 10: 图 4.2：Pendulum-v0 任务可视化

小结

本章节将天授平台与比较流行的 5 个深度强化学习平台进行功能维度和性能维度的对比。实验结果表明天授与其他平台相比，具有模块化、实现简洁、代码质量可靠、用户易用、速度快等等优点。

实验原始数据

附表 1

平台	算法	1	2	3	4	5	平均值	标准差
RLlib	PG	19.43	17.62	18.27	17.38	23.61	19.26	2.29
	DQN	36.21	27.79	25.82	30.42	22.57	28.56	4.60
	A2C	42.84	55.18	63.22	55.45	72.91	57.92	9.94
	PPO	27.18	29.08	54.44	39.65	72.64	44.60	17.04
Baselines	PG	未实现						
	DQN	超过 1000 秒未完成任务，但在 1000 秒之后完成						
	A2C	超过 1000 秒未完成任务，且不能收敛						
	PPO							
PyTorch-DRL *	PG	超过 1000 秒未完成任务，且不能收敛						
	DQN	24.21	53.96	24.42	28.17	27.12	31.58	11.30
	A2C	超过 1000 秒未完成任务，且不能收敛						
	PPO	9.30	21.11	22.26	30.91	36.39	23.99	9.26
Stable-Baselines	PG	未实现						
	DQN	45.84	108.08	51.31	59.56	202.58	93.47	58.05
	A2C	81.00	44.06	56.70	47.81	58.23	57.56	12.87
	PPO	20.64	53.35	21.50	57.78	20.67	34.79	17.02
rlpyt	PG	rlpyt 对于离散动作空间非 Atari 任务的支持不友好，可参考 https://github.com/astooke/rlpyt/issues/135						
	DQN							
	A2C							
	PPO							
天授	PG	1.65	4.98	14.79	6.01	3.03	6.09	4.60
	DQN	5.14	6.32	7.62	5.41	5.97	6.09	0.87
	A2C	9.54	12.06	8.17	9.40	13.80	10.59	2.04
	PPO	30.12	25.21	43.53	22.63	37.59	31.82	7.76

附表 1：CartPole-v0 实验原始数据

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为“训练过程中连续 20 次完整游戏的平均总奖励大于等于 195”。

附表 2

平台	算法	1	2	3	4	5	平均值	标准差
RLlib	PPO	126.91	105.82	131.34	195.46	58.56	123.62	44.23
	DDPG	312.93	329.85	307.26	313.70	309.75	314.70	7.92
	TD3	139.18	158.29	144.52	158.24	149.29	149.90	7.54
	SAC	102.93	95.21	89.95	102.04	96.98	97.42	4.75
Baselines	PPO	804.92	832.88	444.79	733.01	911.53	745.43	160.82
	DDPG	超过 1000 秒未完成任务，且不能收敛						
	TD3	未实现						
	SAC							
PyTorch-DRL *	PPO	PyTorch-DRL 中的 PPO 算法在连续动作空间任务中会报异常错误						
	DDPG	42.50	56.21	69.02	57.53	69.99	59.05	10.03
	TD3	43.97	46.44	46.06	91.04	60.10	57.52	17.71
	SAC	113.88	37.82	40.08	64.38	62.84	63.80	27.37
Stable- Baselines	PPO	206.71	284.84	271.73	271.81	263.58	259.73	27.37
	DDPG	206.58	384.53	135.68	140.45	270.36	277.52	92.67
	TD3	86.22	142.88	91.53	88.77	89.34	99.75	21.63
	SAC	251.22	123.47	165.39	42.07	42.10	124.85	79.14
rlpyt	PPO	rlpyt 并未提供使用 PPO 的任何示例代码，经尝试无法成功跑通						
	DDPG	180.56	130.14	105.95	106.69	94.51	123.57	30.76
	TD3	106.37	98.42	136.02	119.05	105.12	113.00	13.31
	SAC	122.58	169.20	104.50	141.96	125.77	132.80	21.74
天授	PPO	17.64	14.97	20.29	13.28	14.70	16.18	2.49
	DDPG	24.34	51.15	30.25	36.46	44.09	37.26	9.55
	TD3	38.22	52.67	42.15	50.32	36.85	44.04	6.37
	SAC	35.56	35.08	35.61	36.83	37.04	36.02	0.77

附表 2：Pendulum-v0 实验原始数据

*：由于 PyTorch-DRL 中并未实现专门的评测函数，因此适当放宽条件为“训练过程中连续 20 次完整游戏的平均总奖励大于等于-250”。

1.6.7 平台使用实例

实例一：在 CartPole-v0 环境中运行 DQN 算法

本实例将使用天授平台从头搭建整个训练流程，并获得一个能在平均 10 秒之内解决 CartPole-v0 环境的的 DQN [[MKS+15]] 智能体。

首先导入相关的包，并且定义相关参数：

```
import gym, torch, numpy as np, torch.nn as nn
from torch.utils.tensorboard import SummaryWriter
import tianshou as ts

task = 'CartPole-v0'
lr = 1e-3
gamma = 0.9
n_step = 4
eps_train, eps_test = 0.1, 0.05
epoch = 10
step_per_epoch = 10000
```

(下页继续)

(续上页)

```
step_per_collect = 10
target_freq = 320
batch_size = 64
train_num, test_num = 10, 100
buffer_size = 20000
writer = SummaryWriter('log/dqn')
```

创建向量化环境从而能够并行采样：

```
# 也可以用 SubprocVectorEnv
train_envs = ts.env.DummyVectorEnv([
    lambda: gym.make(task) for _ in range(train_num)])
test_envs = ts.env.DummyVectorEnv([
    lambda: gym.make(task) for _ in range(test_num)])
```

使用 PyTorch 原生定义的网络结构，并定义优化器：

```
class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(*[
            nn.Linear(np.prod(state_shape), 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape))
        ])
    def forward(self, s, state=None, info={}):
        if not isinstance(s, torch.Tensor):
            s = torch.tensor(s, dtype=torch.float)
        batch = s.shape[0]
        logits = self.model(s.view(batch, -1))
        return logits, state

env = gym.make(task)
state_shape = env.observation_space.shape or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)
```

初始化策略（Policy）和采集器（Collector）：

```
policy = ts.policy.DQNPolicy(
    net, optim, gamma, n_step,
    target_update_freq=target_freq)
train_collector = ts.data.Collector(
    policy, train_envs, ts.data.VectorReplayBuffer(buffer_size, train_num),
    exploration_noise=True)
test_collector = ts.data.Collector(policy, test_envs, exploration_noise=True)
```

开始训练：

```
result = ts.trainer.offpolicy_trainer(
    policy, train_collector, test_collector, max_epoch=epoch,
    step_per_epoch=step_per_epoch, step_per_collect=step_per_collect,
    update_per_step=1 / step_per_collect, episode_per_test=test_num,
```

(下页继续)

(续上页)

```
    batch_size=batch_size, logger=ts.utils.BasicLogger(writer),
    train_fn=lambda epoch, env_step: policy.set_eps(0.1),
    test_fn=lambda epoch, env_step: policy.set_eps(0.05),
    stop_fn=lambda mean_rewards: mean_rewards >= env.spec.reward_threshold)
print(f'Finished training! Use {result["duration"]}')
```

会有进度条显示，并且在大约 10 秒内训练完毕，结果如下：

```
Epoch #1:  95%|#8| 9480/10000 [00:04<00:00, ..., rew=200.00]
Finished training! Use 4.79s
```

可以将训练完毕的策略模型存储至文件中或者从已有文件中导入模型权重：

```
torch.save(policy.state_dict(), 'dqn.pth')
policy.load_state_dict(torch.load('dqn.pth'))
```

可以以每秒 35 帧的速率查看智能体与环境交互的结果：

```
policy.eval()
policy.set_eps(0.05)
collector = ts.data.Collector(policy, env, exploration_noise=True)
collector.collect(n_episode=1, render=1 / 35)
```

查看 TensorBoard 中存储的结果：

```
tensorboard --logdir log/dqn
```

结果如图 *5.1* 所示。

当然，如果想要定制化训练策略而不使用训练器提供的现有逻辑，也是可以的。下面的代码展示了如何定制化训练策略：

```
# 在正式训练前先收集 5000 帧数据
train_collector.collect(n_step=5000, random=True)

policy.set_eps(0.1)
for i in range(int(1e6)):  # 训练总数
    collect_result = train_collector.collect(n_step=10)

    # 如果收集的 episode 平均总奖励回报超过了阈值，或者每隔 1000 步，
    # 就会对 policy 进行测试
    if collect_result['rews'].mean() >= env.spec.reward_threshold or i % 1000 == 0:
        policy.set_eps(0.05)
        result = test_collector.collect(n_episode=100)
        if result['rews'].mean() >= env.spec.reward_threshold:
            print(f'Finished training! Test mean returns: {result["rews"].mean()}')
            break
        else:
            # 重新设置 eps 为 0.1，表示训练策略
            policy.set_eps(0.1)

    # 使用采样出的数据组进行策略训练
    losses = policy.update(64, train_collector.buffer)
```

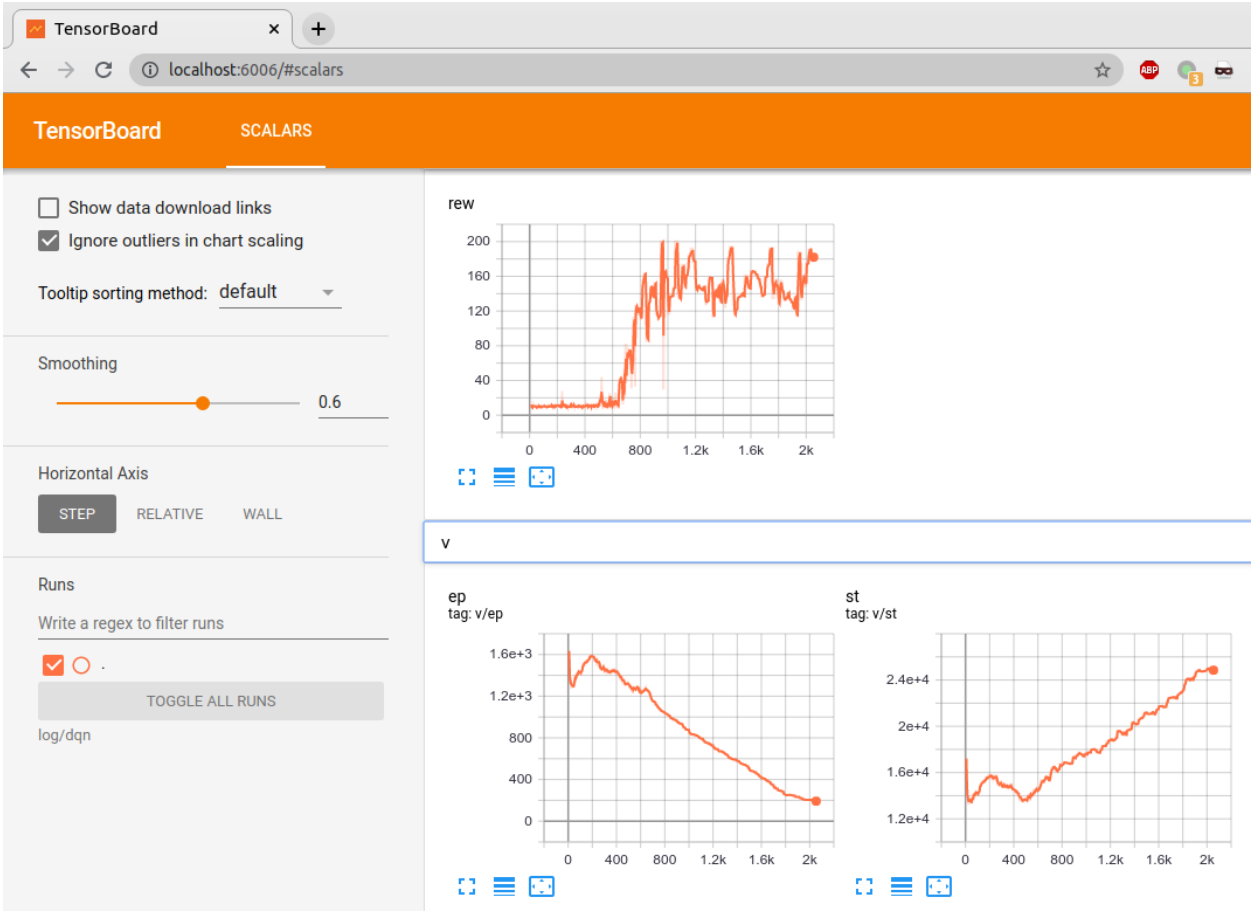


图 11: 图 5.1：TensorBoard 可视化训练过过程

实例二：循环神经网络的训练

在 POMDP 场景中往往需要循环神经网络的训练支持。此处为简单起见，仍然以实例一中的场景和代码为基础进行展示。需要的改动如下：

首先修改模型为 LSTM：

```
class Recurrent(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.fc1 = nn.Linear(np.prod(state_shape), 128)
        self.nn = nn.LSTM(input_size=128, hidden_size=128,
                          num_layers=3, batch_first=True)
        self.fc2 = nn.Linear(128, np.prod(action_shape))

    def forward(self, s, state=None, info={}):
        if not isinstance(s, torch.Tensor):
            s = torch.tensor(s, dtype=torch.float)
        # s [bsz, len, dim] (training)
        # or [bsz, dim] (evaluation)
        if len(s.shape) == 2:
            bsz, dim = s.shape
            length = 1
        else:
            bsz, length, dim = s.shape
        s = self.fc1(s.view([bsz * length, dim]))
        s = s.view(bsz, length, -1)
        self.nn.flatten_parameters()
        if state is None:
            s, (h, c) = self.nn(s)
        else:
            # we store the stack data with [bsz, len, ...]
            # but pytorch rnn needs [len, bsz, ...]
            s, (h, c) = self.nn(s, (
                state['h'].transpose(0, 1).contiguous(),
                state['c'].transpose(0, 1).contiguous()))
        s = self.fc2(s[:, -1])
        # make sure the 0-dim is batch size: [bsz, len, ...]
        return s, {'h': h.transpose(0, 1).detach(),
                   'c': c.transpose(0, 1).detach()}
```

其次重新定义策略，并将 `train_collector` 中的重放缓冲区设置成堆叠采样模式，堆叠帧数 n 为 4：

```
env = gym.make(task)
state_shape = env.observation_space.shape or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Recurrent(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)

policy = ts.policy.DQNPolicy(
    net, optim, gamma, n_step,
    target_update_freq=target_freq)
train_collector = ts.data.Collector(
    policy, train_envs,
    ts.data.VectorReplayBuffer(buffer_size, train_num, stack_num=4),
    exploration_noise=True)
test_collector = ts.data.Collector(policy, test_envs, exploration_noise=True)
```

即可使用实例一中的代码进行正常训练，结果如下：

```
Epoch #2:  84%|#4| 8420/10000 [00:21<00:03, ..., rew=200.00]
Finished training! Use 37.22s
```

实例三：多模态任务训练

在像机器人抓取之类的任务中，智能体会获取多模态的观测值。天授完整保留了多模态观测值的数据结构，以数据组的形式给出，并且能方便地支持分片操作。以 Gym 环境中的“FetchReach-v1”为例，每次返回的观测值是一个字典，包含三个元素“observation”、“achieved_goal”和“desired_goal”。

在实例一代码的基础上进行修改：

```
task = 'FetchReach-v1'
train_envs = ts.env.DummyVectorEnv([
    lambda: gym.make(task) for _ in range(train_num)])
test_envs = ts.env.DummyVectorEnv([
    lambda: gym.make(task) for _ in range(test_num)])

class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(*[
            nn.Linear(np.prod(state_shape), 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape))
        ])
    def forward(self, s, state=None, info={}):
        o = s.observation
        # s.achieved_goal, s.desired_goal are also available
        if not isinstance(o, torch.Tensor):
            o = torch.tensor(o, dtype=torch.float)
        batch = o.shape[0]
        logits = self.model(o.view(batch, -1))
        return logits, state

env = gym.make(task)
env.spec.reward_threshold = 1e10
state_shape = env.observation_space.spaces['observation']
state_shape = state_shape.shape
action_shape = env.action_space.shape
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=lr)
```

剩下的代码与实例一一致，可以直接运行。通过对比可以看出，只需改动神经网络中 `forward` 函数的 s 参数的处理即可。

1.6.8 总结

本论文描述了一个基于 PyTorch 的深度强化学习算法平台“天授”。该平台支持了诸多主流的强化学习算法（主要为免模型强化学习算法），支持各种不同的环境的并行采样、数据存储、定制化，还同时做到了模块化、实现简洁、可复现性强、接口灵活等特性，并且在基准测试中，天授的速度优于其他已有平台。

天授平台旨在提供一个用户友好的标准化的强化学习平台，降低算法开发成本。如今天授已在 GitHub 上开源[1]，并且提供了一系列教程和代码文档[2]，目前已经拥有 1500 多颗星标，收到了众多使用者的一致好评。

后续的工作将围绕如下方面进行：

- **算法：**（1）加入更多免模型强化学习算法，比如 Rainbow DQN [[HMvH+18]]；（2）加入基于模型的强化学习算法，比如 MCTS 与 AlphaGo [[SHM+16]]（目前平台接口已经支持，代码正在完善中）；（3）加入更多模仿学习算法，比如 GAIL [[HE16]]；（4）加入多智能体训练的接口；
- **环境：**加入更多种类的环境并行接口，比如共享内存的环境接口，做到更高效的并行采样；
- **文档：**完善教程；
- **示例：**提供更多任务上（如 Atari、Mujoco 各个任务）调优过的示例代码，方便开箱即用与二次开发。

1.6.9 参考文献列表

参考文献

1.7 参与贡献

1.7.1 安装开发版本

假设现在在仓库的根目录下，使用命令

```
pip3 install -e .[dev]
```

可以将天授仓库以可编辑模式安装（就是可以直接在源代码中进行改动，而不用重新 pip install 就可以用）。使用命令

```
python3 setup.py develop --uninstall
```

1.7.2 PEP8 代码风格检测

本项目遵循原始的 PEP8 代码风格规定，在项目根目录运行

```
flake8 . --count --show-source --statistics
```

即可获得检测报告。

[1] GitHub 项目地址：https://github.com/thu-ml/tianshou/
[2] 文档地址：http://tianshou.readthedocs.io/

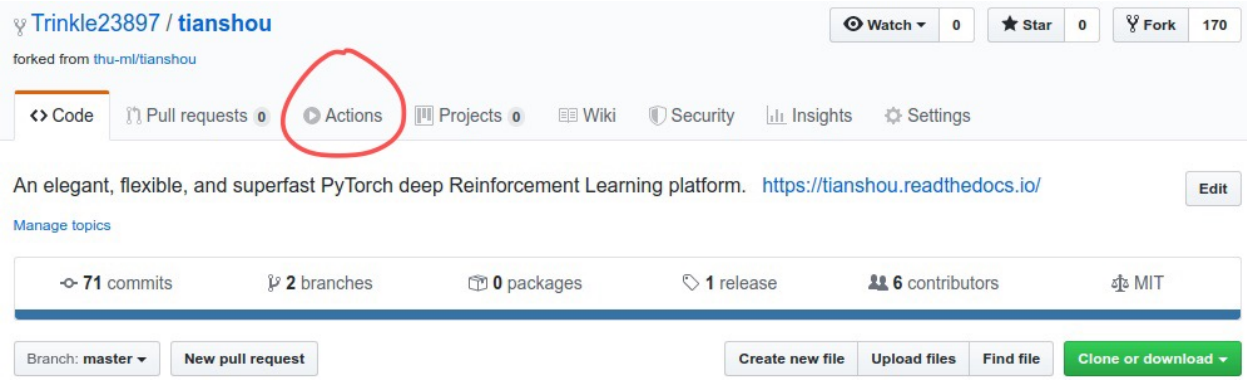
1.7.3 本地测试

运行如下命令，即可在本地自动对项目进行单元测试并给出报告结果：

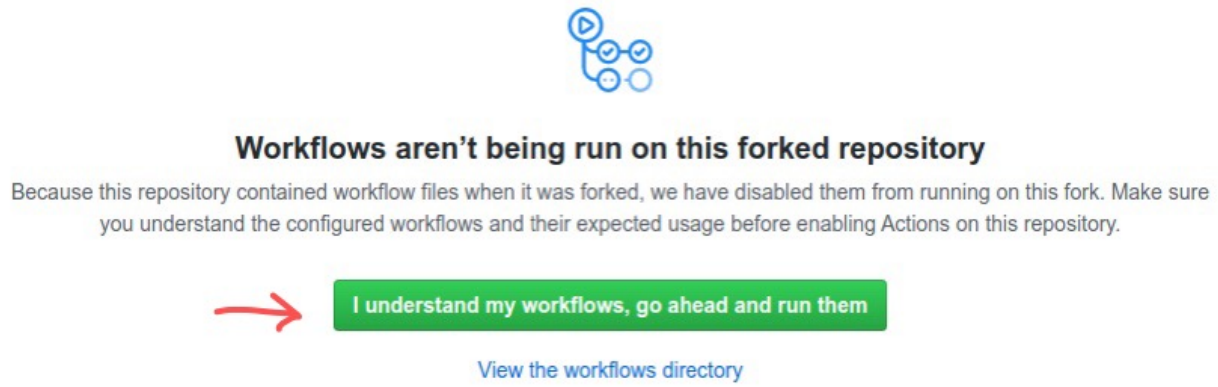
```
pytest test --cov tianshou -s --durations 0 -v
```

1.7.4 使用 GitHub Actions 进行测试

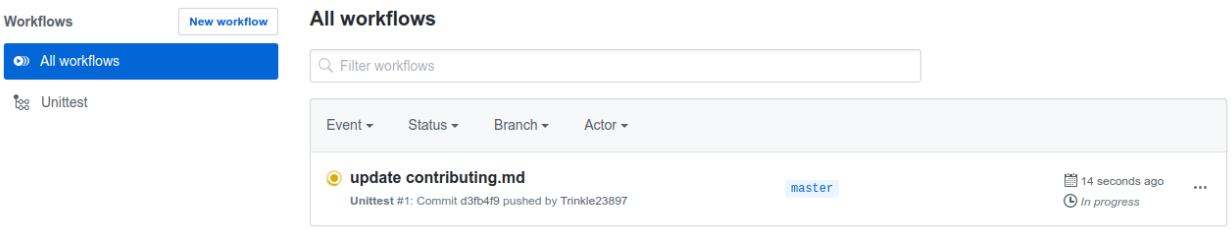
1. 点击您 fork 出来的仓库的 `Actions` 图标：



2. 点击绿色按钮：



3. 会看到 `Actions Enabled.` 显示
4. 在此之后，一旦有新的 commit 被 push 上来，GitHub Actions 会自动帮您运行单元测试：



1.7.5 英文文档

文档在 `docs/` 文件夹下，以 `.rst` 格式纂写。关于 ReStructuredText 的教程可参考 这里.

API 文档由 Sphinx 自动生成，`docs/api/` 目录下列出了需要生成的 API 文档。

如果需要编译出整个文档并以网页版形式预览，需要在 `docs/` 文件夹下运行

```
make html
```

它会将文档生成在 `docs/_build` 目录中，可以使用浏览器直接预览。

1.7.6 中文文档

中文文档在 另外一个仓库中。

与英文文档不同，此处不提供具体 API 文档的对应（因为变动可能不同步），而只是提供了一个从宏观层面来了解天授平台的一个教程。文档的编译与发布和英文文档无异。

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. URL: https://doi.org/10.1038/nature14236, doi:10.1038/nature14236.

[LHP+16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1509.02971.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, 2017. URL: http://arxiv.org/abs/1707.06347, arXiv:1707.06347.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. URL: https://doi.org/10.1038/nature16961, doi:10.1038/nature16961.

[SEJ+20] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, and others. Improved protein structure prediction using potentials from deep learning. *Nature*, pages 1–5, 2020.

[BBC+19] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, 2019. URL: http://arxiv.org/abs/1912.06680, arXiv:1912.06680.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*

2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada, 8024–8035. 2019. URL: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, 1106–1114. 2012. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[HGDollarG17] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, 2980–2988. 2017. URL: https://doi.org/10.1109/ICCV.2017.322, doi:10.1109/ICCV.2017.322.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: an incremental improvement. *CoRR*, 2018. URL: http://arxiv.org/abs/1804.02767, arXiv:1804.02767.

[TYRW14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, 1701–1708. 2014. URL: https://doi.org/10.1109/CVPR.2014.220, doi:10.1109/CVPR.2014.220.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, 234–241. 2015. URL: https://doi.org/10.1007/978-3-319-24574-4__xunadd_text_character:nN{\textbackslash}{\}{}_28, doi:10.1007/978-3-319-24574-4_28.

[Tes94] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994. URL: https://doi.org/10.1162/neco.1994.6.2.215, doi:10.1162/neco.1994.6.2.215.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[Ach18] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. https://github.com/openai/spinningup, 2018.

[LLN+18] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 3059–3068. 2018. URL: http://proceedings.mlr.press/v80/liang18b.html.

[SA19] Adam Stooke and Pieter Abbeel. Rlpyt: A research code base for deep reinforcement learning in pytorch. *CoRR*, 2019. URL: http://arxiv.org/abs/1909.01500, arXiv:1909.01500.

[PDLN19] Vitchyr H. Pong, Murtaza Dalal, Steven Lin, and Ashvin Nair. Rlkit: collection of reinforcement learning algorithms. https://github.com/vitchyr/rlkit, 2019.

[gc19] The garage contributors. Garage: a toolkit for reproducible reinforcement learning research. https://github.com/rlworkgroup/garage, 2019.

[CMG+18] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *CoRR*, 2018. URL: http://arxiv.org/abs/1812.06110, arXiv:1812.06110.

[ODH+20] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, Benjamin Van Roy, Richard S. Sutton, David Silver, and Hado van Hasselt. Behaviour suite for reinforcement learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. 2020. URL: https://openreview.net/forum?id=rygf-kSYwH.

[HRE+18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[Pla16] Matthias Plappert. Keras-rl. https://github.com/keras-rl/keras-rl, 2016.

[Chr19] Petros Christodoulou. Deep reinforcement learning algorithms with pytorch. https://github.com/p-christ/Deep-Reinforcement-Learning-Algorithms-with-PyTorch, 2019.

[KSF17] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. https://github.com/tensorforce/tensorforce, 2017.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, 2016. URL: http://arxiv.org/abs/1606.01540, arXiv:1606.01540.

[ABC+16] Mart\'ın Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, 265–283. 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[ACR+17] Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 5048–5058. 2017. URL: http://papers.nips.cc/paper/7090-hindsight-experience-replay.

[SMSM99] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, 1057–1063. 1999. URL: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 1928–1937. 2016. URL: http://proceedings.mlr.press/v48/mniha16.html.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, 1889–1897. 2015. URL: http://proceedings.mlr.press/v37/schulman15.html.

[SML+16] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1506.02438.

[vHGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, 2094–2100. 2016. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389.

[SQAS16] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1511.05952.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learn-*

ing, ICML 2014, Beijing, China, 21-26 June 2014, 387–395. 2014. URL: http://proceedings.mlr.press/v32/silver14.html.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 1582–1591. 2018. URL: http://proceedings.mlr.press/v80/fujimoto18a.html.

[HZH+18] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *CoRR*, 2018. URL: http://arxiv.org/abs/1812.05905, arXiv:1812.05905.

[FLA16] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: deep inverse optimal control via policy optimization. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 49–58. 2016. URL: http://proceedings.mlr.press/v48/finn16.html.

[HE16] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 4565–4573. 2016. URL: http://papers.nips.cc/paper/6391-generative-adversarial-imitation-learning.

[HMvH+18] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 3215–3222. 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204.