GenRL

Release 0.1

Feb 22, 2021

Reinforcement Learning

CHAPTER 1

Features

- Unified Trainer and Logging class: code reusability and high-level UI
- Ready-made algorithm implementations: ready-made implementations of popular RL algorithms.
- Extensive Benchmarking
- Environment implementations
- Heavy Encapsulation useful for new algorithms

CHAPTER 2

Contents

2.1 Installation

2.1.1 PyPI Package

GenRL is compatible with Python 3.6 or later and also depends on `pytorch` and `openai-gym`. The easiest way to install GenRL is with pip, Python's preferred package installer.

```
$ pip install genrl
```

Note that GenRL is an active project and routinely publishes new releases. In order to upgrade GenRL to the latest version, use pip as follows.

```
$ pip install -U genrl
```

2.1.2 From Source

If you intend to install the latest unreleased version of the library (i.e from source), you can simply do:

```
$ git clone https://github.com/SforAiDl/genrl.git
$ cd genrl
$ python setup.py install
```

2.2 About

2.2.1 Introduction

Reinforcement Learning has taken massive leaps forward in extending current AI research. David Silver's paper on playing Atari with Deep Reinforcement Learning can be considered one of the seminal papers in establishing a completely new landscape of Reinforcement Learning Research. With applications in Robotics, Healthcare and numerous other domains, RL has become the prime mechanism of modelling Sequential Decision Making through AI.

Yet, current libraries and resources in Reinforcement Learning are either very limited, messy and/or are scattered. OpenAI's Spinning Up is a great resource for getting started with Deep Reinforcement Learning but it fails to cover more basic concepts in Reinforcement Learning for e.g. Multi Armed Bandits. garage is a great resource for reproducing and evaluating RL algorithms but it fails to introduce a newbie to RL.

With GenRL, our goal is three-fold: - To educate the user about Reinforcement learning. - Easy to understand implementations of State of the Art Reinforcement Learning Algorithms. - Providing utilities for developing and evaluating new RL algorithms. Or in a sense be able to implement any new RL algorithm in less than 200 lines.

2.2.2 Policies and Values

Modern research on Reinforcement Learning is majorly based on Markov Decision Processes. Policy and Value Functions are one of the core parts of such a problem formulation. And so, polices and values form one of the core parts of our library.

2.2.3 Trainers and Loggers

Trainers

Most current algorithms follow a standard procedure of training. Considering a classification between On-Policy and Off-Policy Algorithms, we provide high level APIs through Trainers which can be coupled with Agents and Environments for training seamlessly.

Lets take the example of an On-Policy Algorithm, Proximal Policy Optimization. In our Agent, we make sure to define three methods: `collect_rollouts`, `get_traj_loss` and finally `update_policy`.

The `OnPolicyTrainer` simply calls these functions and enables high level usage by simple defining of three methods.

Loggers

At the moment, we support three different types of Loggers. `HumanOutputFormat`, `TensorboardLogger` and `CSVLogger`. Any of these loggers can be initialized really easily by the top level `Logger` class and specifying the individual formats in which logging should performed.

```
logger = Logger(logdir='logs/', formats=['stdout', 'tensorboard'])
```

After which logger can perform logging easily by providing it with dictionaries of data. For e.g.

```
logger.write({"logger":0})
```

Note: The Tensorboard logger requires an extra x-axis parameter, as it plots data rather than just show it in a tabular format.

2.2.4 Agent Encapsulation

WIP

2.2.5 Environments

Wrappers

2.3 Tutorials

2.3.1 Bandit Tutorials

Multi Armed Bandit Overview

Training an EpsilonGreedy agent on a Bernoulli Multi Armed Bandit

Multi armed bandits is one of the most basic problems in RL. Think of it like this, you have 'n' levers in front of you and each of these levers will give you a different reward. For the purposes of formalising the problem the reward is written down in terms of a reward function i.e., the probability of getting a reward when a lever is pulled.

Suppose you try out one of the levers and get a positive reward. What do you do next? Should you just keep pulling that lever every time or think what if there might be a better reward to pulling one of the other levers? This is the exploration - exploitation dilemma.

Exploitation - Utilise the information you have gathered till now, to make the best decision. In this case, after 1 try you know a lever is giving you a positive reward and you just *exploit* it further. Since you do not care about other arms if you keep *exploiting*, it is known as the greedy action.

Exploration - You explore the untried levers in an attempt to maybe discover another one which has a higher payout than the one you currently have some knowledge about. This is exploring all your options without worrying about the short-term rewards, in hope of finding a lever with a bigger reward, in the long run.

You have to use an algorithm which correctly trades off exploration and exploitation as we do not want a 'greedy' algorithm which only exploits and does not explore at all, because there are very high chances that it will converge to a sub-optimal policy. We do not want an algorithm that keeps exploring either as this would lead to sub-optimal rewards inspite of knowing the best action to be taken. In this case, the optimal policy will be to always pull the lever with the highest reward, but at the beginning we do not know the probability distribution of the rewards.

So, we want a policy which explores actively at the beginning, building up an estimate for the reward values(defined as *quality*) of all the actions, and then exploiting that from that time onwards.

A Bernoulli Multi-Armed Bandit has multiple arms with each having a different bernoulli distribution over its reward. Basically each arm has a probabilty associated with it which is the probability of getting a reward if that arm is pulled. Our aim is to find the arm which has the highest probabilty, thus giving us the maximum return.

Notation:

$Q_t(a)$: Estimated quality of action 'a' at timestep 't'.

$q(a)$: True value of action 'a'.

We want our estimate $Q_t(a)$ to be as close to the true value $q(a)$ as possible, so we can make the correct decision.

Let the action with the maximum quality be a^*:s

$$q^* = q(a^*)$$

Our goal is to find this q^*.

The ‘regret function’ is defined as the sum of ‘regret’ accumulated over all timesteps. This regret is the cost of not choosing the optimal arm and instead of exploring. Mathematically it can be written as:

$$L = E[\sum_{t=0}^{T} q^* - Q_t(a)]$$

Some policies which are effective at exploring are: 1. Epsilon Greedy 2. Gradient Algorithm 3. UCB(Upper Confidence Bound) 4. Bayesian 5. Thompson Sampling

Epsilon Greedy is the most basic exploratory policy which follows a simple principle to balance exploration and exploitation. It ‘exploits’ the current knowledge of the bandit most of the times, i.e. takes the action with the largest q value. But with a small probability epsilon, it also explores a random action. The value of epsilon signifies how much you want the agent explore. Higher the value, the more it explores. But remember you do not want an agent to explore too much even after it has a pretty confident estimate of the reward function, so the value of epislon should neither be too high nor too low!

For the bandit, you can set the number of bandits, number of arms, and also reward probabilities of each of these arms seperately.

Code to train an Epsilon Greedy agent on a Bernoulli Multi-Armed Bandit:

```
import gym
import numpy as np

from genrl.bandit import BernoulliMAB, EpsGreedyMABAgent, MABTrainer

reward_probs = np.random.random(size=(bandits, arms))
bandit = BernoulliMAB(arms=5, reward_probs=reward_probs, context_type="int")
agent = EpsGreedyMABAgent(bandit, eps=0.05)

trainer = MABTrainer(agent, bandit)
trainer.train(timesteps=10000)
```

More details can be found in the docs for BernoulliMAB, EpsGreedyMABAgent, MABTrainer.

You can also refer to the book “Reinforcement Learning: An Introduction”, Chapter 2 for further information on bandits.

Contextual Bandits Overview

Problem Setting

To get some background on the basic multi armed bandit problem, we recommend that you go through the *Multi Armed Bandit Overview* first. The contextual bandit (CB) problem varies from the basic case in that at each timestep, a context vector $x \in \mathbb{R}^d$ is presented to the agent. The agent must then decide on an action $a \in \mathcal{A}$ to take based on that context. After the action is taken, the reward $r \in \mathbb{R}$ for only that action is revealed to the agent (a feature of all reinforcement learning problems). The aim of the agent remains the same - minimising regret and thus finding an optimal policy.

Here you still have the problem of exploration vs exploitation, but the agent also needs to find some relation between the context and reward.

A Simple Example

Lets consider the simplest case of the CB problem. Instead of having only one k-armed bandit that needs to be solved, say we have m different k-armed Bernoulli bandits. At each timestep, the context presented is the number of the bandit for which an action needs to be selected: $i \in \mathbb{I}$ where $0 < i \leq m$

Although real life CB problems usually have much higher dimensional contexts, such a toy problem can be usefull for testing and debugging agents.

To instantiate a Bernoulli bandit with $m = 10$ and $k = 5$ (10 different 5-armed bandits) -

```
from genrl.bandit import BernoulliMAB

bandit = BernoulliMAB(bandits=10, arms=5, context_type="int")
```

Note that this is using the same `BernoulliMAB` as in the simple bandit case except that instead of the `bandits` argument defaulting to `1`, we are explicitly saying we want multiple bandits (a contexutal case)

Suppose you want to solve this bandit with a UCB based policy.

```
from genrl.bandit import UCBMABAgent

agent = UCBMABAgent(bandit)
context = bandit.reset()

action = agent.select_action(context)
new_context, reward = bandit.step(action)
```

To train the agent, you an set up a loop which calls the `update_params` method on the agent whenever you want to agent to learn from actions it has taken. For convinience it is highly recommended to use the `MABTrainer` in such cases.

Data based Conextual Bandits

Lets consider a more realistic class of CB problem. I real life, you the CB setting is usually used to model recommendation or classification problems. Here, instead of getting an integer as the context, you will get a d-dimensional feature vector $\mathbf{x} \in \mathbb{R}^d$. This is also different from regular classification since you only get the reward $r \in \mathbb{R}$ for the action you have taken.

While tabular solutions can work well for integer contexts (see the implentation of any `genrl.bandit.MABAgent` for details), when you have a high dimensional vector, the agent should be able to infer the complex relation between the contexts and rewards. This can be done by modelling a conditional distribution over rewards for each action given the context.

$$P(r|a, \mathbf{x})$$

There are many ways to do this. For a detailed explanation and comparison of contextual bandit methods you can refer to this paper.

The following are the agents implemented in `genrl`

- Linear Posterior Inference
- Neural Network based Linear
- Variational
- Neural Netowork based Espilon Greedy
- Bootstrap
- Parameter noise Sampling

You can find the tutorials for most of these in *Bandit Tutorials*.

All the methods which use neural networks, provide an option to train and evaluate with dropout, have a decaying learning rate and a limit for gradient clipping. The sizes of hidden layers for the networks can also be specified. Refer to docs of the specific agents to see how to use these options.

Individual agents will have other method specific paramters to control behavior. Although default values have been provided, it may be neccessary to tune these for individual use cases.

The following bandits based on datasets are implemented in `genrl`

- Adult Census Income Dataset
- US Census Dataset
- Forest covertype Datset
- MAGIC Gamma Telescope dataset
- Mushroom Dataset
- Statlog Space Shuttle Dataset

For each bandit, while instatiating an object you can either specify a path to the data file or pass `download=True` as an argument to download the data directly.

Data based Bandit Example

For this example, we'll model the Statlog dataset as a bandit problem. You can read more about the bandit in the Statlog docs. In brief we have the number of arms as $k = 7$ and dimension of context vector as $d = 9$. The agent will get a reward $r = 1$ if it selects the correct arm else $r = 0$.

```python
from genrl.bandit import StatlogDataBandit

bandit = StatlogDataBandit(download=True)
context = bandit.reset()
```

Suppose you want to solve this bandit with a Greedy neural network based policy.

```python
from genrl.bandit import NeuralLinearPosteriorAgent

agent = NeuralLinearPosteriorAgent(bandit)
context = bandit.reset()

action = agent.select_action(context)
new_context, reward = bandit.step(action)
```

To train the agent, we highly reccomend using the `DCBTrainer`. You can refer to the implementation of the `train` function to get an idea of how to implemente your own training loop.

```python
from genrl.bandit import DCBTrainer

trainer = DCBTrainer(agent, bandit)
trainer.train(timesteps=5000, batch_size=32)
```

Further material about bandits

1. Deep Contextual Multi-armed Bandits, Collier and Llorens, 2018

2. Deep Bayesian Bandits Showdown, Riquelme et al, 2018
3. A Contextual Bandit Bake-off, Bietti et al, 2020

UCB

Training a UCB algorithm on a Bernoulli Multi-Armed Bandit

For an introduction to Multi Armed Bandits, refer to *Multi Armed Bandit Overview*

The UCB algorithm follows a basic principle - 'Optimism in the face of uncertainty'. What this means is that we should always select the action whose reward we are most uncertain of. We quantify the uncertainty of taking an action by calculating an upper bound of the quality(reward) for that action. We then select the greedy action with respect to this upper bound.

Hoeffding's inequality:

$$P[q(a) > Q_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

,

q(a) is the quality of that action,

$Q_t(a)$ is the estimate of the quality of action 'a' at time 't',

$U_t(a)$ is the upper bound for uncertainty for that action at time 't',

$N_t(a$ is the number of times action 'a' has been selected

$$e^{-2N_t(a)U_t(a)^2} = t^{-4}$$

$$U_t(a) = \sqrt{\frac{2logt}{N_t(a)}}$$

Action taken: a = argmax$(Q_t(a) + U_t(a))$

As we can see, the less an action has been tried, more the uncertainty is (due to $N_t(a)$ being in the denominator), which leads to that action having a higher chance of being explored. Also, theoretically, as $N_t(a)$ goes to infinity, the uncertainty decreases to 0 giving us the true value of the quality of that action: q(a). This allows us to 'exploit' the greedy action a^* from then.

Code to train a UCB agent on a Bernoulli Multi-Armed Bandit:

```
import gym
import numpy as np

from genrl.bandit import BernoulliMAB, MABTrainer, UCBMABAgent

bandits = 10
arms = 5

reward_probs = np.random.random(size=(bandits, arms))
bandit = BernoulliMAB(bandits, arms, reward_probs, context_type="int")
agent = UCBMABAgent(bandit, confidence=1.0)

trainer = MABTrainer(agent, bandit)
trainer.train(timesteps=10000)
```

More details can be found in the docs for BernoulliMAB, UCB and MABTrainer.

Thompson Sampling

Using Thompson Sampling on a Bernoulli Multi-Armed Bandit

For an introduction to Multi Armed Bandits, refer to *Multi Armed Bandit Overview*

Thompson Sampling is one of the best methods for solving the Bernoulli multi-armed bandits problem. It is a 'sample-based probability matching' method.

We initially *assume* an initial distribution(prior) over the quality of each of the arms. We can model this prior using a Beta distribution, parametrised by alpha(α) and beta(β).

$$PDF = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$$

Let's just think of the denominator as some normalising constant, and focus on the numerator for now. We initialise α = β = 1. This will result in a uniform distribution over the values (0, 1), making all the values of quality from 0 to 1 equally probable, so this is a fair initial assumption. Now think of α as the number of times we get the reward '1' and β as the number of times we get '0', for a particular arm. As our agent interacts with the environment and gets a reward for pulling any arm, we will update our prior for that arm using Bayes Theorem. What this does is that it gives a posterior distribution over the quality, according to the rewards we have seen so far.

At each timestep, we sample the quality: $Q_t(a)$ for each arm from the posterior and select the sample with the highest value. The more an action is tried out, the narrower is the distribution over its quality, meaning we have a confident estimate of its quality (q(a)). If an action has not been tried out that often, it will have a more wider distribution (high variance), meaning we are uncertain about our estimate of its quality (q(a)). This wider variance of an arm with an uncertain estimate creates opportunities for it to be picked during sampling.

As we experience more successes for a particular arm, the value of α for that arm increases and similiarly, the more failures we experience, the value of β increases. Higher the value of one of the parameters as compared to the other, the more skewed is the distribution in one of the directions. For eg. if α = 100 and β = 50, we have seen considerably more successes than failures for this arm and so our estimate for its quality should be >0.5. This will be reflected in the posterior of this arm, i.e. the mean of the distribution, characterised by $\frac{\alpha}{\alpha+\beta}$ will be 0.66, which is >0.5 as we expected.

Code to use Thompson Sampling on a Bernoulli Multi-Armed Bandit:

```python
import gym
import numpy as np

from genrl.bandit import BernoulliMAB, MABTrainer, ThompsonSamplingMABAgent

bandits = 10
arms = 5
alpha = 1.0
beta = 1.0

reward_probs = np.random.random(size=(bandits, arms))
bandit = BernoulliMAB(bandits, arms, reward_probs, context_type="int")
agent = ThompsonSamplingMABAgent(bandit, alpha, beta)

trainer = MABTrainer(agent, bandit)
trainer.train(timesteps=10000)
```

More details can be found in the docs for BernoulliMAB, UCB and MABTrainer.

Bayesian

Using Bayesian Method on a Bernoulli Multi-Armed Bandit

For an introduction to Multi Armed Bandits, refer to *Multi Armed Bandit Overview*

This method is also based on the prinicple - 'Optimism in the face of uncertainty', like UCB. We initially *assume* an initial distribution(prior) over the quality of each of the arms. We can model this prior using a Beta distribution, parametrised by alpha(α) and beta(β).

$$PDF = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

Let's just think of the denominator as some normalising constant, and focus on the numerator for now. We initialise $\alpha = \beta = 1$. This will result in a uniform distribution over the values (0, 1), making all the values of quality from 0 to 1 equally probable, so this is a fair initial assumption. Now think of α as the number of times we get the reward '1' and β as the number of times we get '0', for a particular arm. As our agent interacts with the environment and gets a reward for pulling any arm, we will update our prior for that arm using Bayes Theorem. What this does is that it gives a posterior distribution over the quality, according to the rewards we have seen so far.

This is quite similar to Thompson Sampling. But what is different here is that we explicity try to calculate the uncertainty of a particular action by calculating the standard deviation(σ) of its posterior. We add this std. dev to the mean of the posterior, giving us an *upper bound* of the quality of that arm. At each timestep we select a greedy action based on this upper bound we calculated.

$$a_t = argmax(q_t(a) + \sigma_{q_t})$$

As we try out an action more and more, the standard deviation of the posterior decreases, corresponding to a decrease in the uncertainty of that action, which is exactly what we want. If an action has not been tried that often, it will have a wider posterior, meaning higher chances of it getting selected based on its upper bound.

Code to use Bayesian method on a Bernoulli Multi-Armed Bandit:

```python
import gym
import numpy as np

from genrl.bandit import BayesianUCBMABAgent, BernoulliMAB, MABTrainer

bandits = 10
arms = 5
alpha = 1.0
beta = 1.0

reward_probs = np.random.random(size=(bandits, arms))
bandit = BernoulliMAB(bandits, arms, reward_probs, context_type="int")
agent = BayesianUCBMABAgent(bandit, alpha, beta)

trainer = MABTrainer(agent, bandit)
trainer.train(timesteps=10000)
```

More details can be found in the docs for BernoulliMAB, BayesianUCBMABAgent and MABTrainer.

Gradients

Using Gradient Method on a Bernoulli Multi-Armed Bandit

For an introduction to Multi Armed Bandits, refer to *Multi Armed Bandit Overview*

This method is different compared to others. In other methods, we explicity attempt to estimate the 'value' of taking an action (its quality) whereas in this method we approach the problem in a different way. Here, instead of estimating how good an action is through its quality, we only care about its preference of being selected compared to other actions. We denote this preference by $H_t(a)$. The larger the preference of an action 'a', more are the chances of it being selected, but this preference has no interpretation in terms of the reward for that action. Only the relative preference is important.

The action probabilites are related to these action preferences $H_t(a)$ by a softmax function. The probability of taking action a_j is given by:

$$P(a_j) = \frac{e^{H_t(a_j)}}{\sum_{i=1}^{A} e^{H_t(a_i)}} = \pi_t(a_j)$$

where, A is the total number of actions and $\pi_t(a)$ is the probability of taking action 'a' at timestep 't'.

We initialise the preferences for all the actions to be 0, meaning $\pi_t(a) = \frac{1}{A}$ for all actions.

After computing $\pi_t(a)$ for all actions at each timestep, the action is sampled using this probability. Then that action is performed and based on the reward we get, we update our preferences.

The update rule bacially performs stochastic gradient ascent:

$H_{t+1}(a_t) = H_t(a_t) + \alpha(R_t - \bar{R_t})(1 - \pi_t(a_t))$, for a_t: action taken at time 't'

$H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R_t})(\pi_t(a))$ for rest of the actions

where, α is the step size, R_t is the reward obtained at time 't' and $\bar{R_t}$ is the mean reward obtained upto time t. If current reward is larger than the mean reward, we increase our preference for that action taken at time 't'. If it is lower than the mean reward, we decrease our preference for that action. The preferences for the rest of the actions are updated in the opposite direction.

For a more detailed mathematical analysis and derivation of the update rule, refer to chapter 2 of Sutton & Barto.

Code to use the Gradient method on a Bernoulli Multi-Armed Bandit:

```python
import gym
import numpy as np

from genrl.bandit import BernoulliMAB, GradientMABAgent, MABTrainer

bandits = 10
arms = 5

reward_probs = np.random.random(size=(bandits, arms))
bandit = BernoulliMAB(bandits, arms, reward_probs, context_type="int")
agent = GradientMABAgent(bandit, alpha=0.1, temp=0.01)

trainer = MABTrainer(agent, bandit)
trainer.train(timesteps=10000)
```

More details can be found in the docs for BernoulliMAB, BayesianUCBMABAgent and MABTrainer.

Linear Posterior Inference

For an introduction to the Contextual Bandit problem, refer to *Contextual Bandits Overview*.

In this agent we assume a linear relationship between context and reward distribution of the form

$$Y = X^T\beta + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

We can utilise bayesian linear regression to find the parameters β and σ. Since our agent is continually learning, the parameters of the model will being updated according the ($\mathbf{x}$, a, r) transitions it observes.

For more complex non linear relations, we can make use of neural networks to transform the context into a learned embedding space. The above method can then be used on this latent embedding to model the reward.

An example of using a neural network based linear posterior agent in `genrl` -

```
from genrl.bandit import NeuralLinearPosteriorAgent, DCBTrainer

agent = NeuralLinearPosteriorAgent(bandit, lambda_prior=0.5, a0=2, b0=2, device="cuda
↪")

trainer = DCBTrainer(agent, bandit)
trainer.train()
```

Note that the priors here are used to parameterise the initial distribution over β and σ. More specificaly `lambda_prior` is used to parameterise a guassian distribution for β while `a0` and `b0` are paramters of an inverse gamma distribution over σ^2. These are updated over the course of exploring a bandit. More details can be found in Section 3 of this paper.

All hyperparameters can be tuned for individual use cases to improve training efficiency and achieve convergence faster.

Refer to the LinearPosteriorAgent, NeuralLinearPosteriorAgent and DCBTrainer docs for more details.

Variational Inference

For an introduction to the Contextual Bandit problem, refer to *Contextual Bandits Overview*.

In this method, we try find a distribution $P_\theta(r|\mathbf{x}, a)$ by minimising the KL divergence with the true distribution. For the model we take a neueral network where each weight is modelled by independant gaussians, also known as Bayesian Neural Nets.

An example of using a variational inference based agent in `genrl` with bayesian net of hidden layer of 128 neurons and standard deviation of 0.1 for al the weights -

```
from genrl.bandit import VariationalAgent, DCBTrainer

agent = VariationalAgent(bandit, hidden_dims=[128], noise_std=0.1, device="cuda")

trainer = DCBTrainer(agent, bandit)
trainer.train()
```

Refer to the VariationalAgent, and DCBTrainer docs for more details.

Bootstrap

For an introduction to the Contextual Bandit problem, refer to *Contextual Bandits Overview*.

In the bootstrap agent multiple different neural network based models are trained simultaneously. Different transition databases are maintained for each model and every time we observe a transition it is added to each dataset with some probability. At each timestep, the model used to select an action is chosen randomly from the set of models.

By having multiple different models initialised with different random weights, we promote the exploration of the loss landscape which may have multiple different local optima.

An example of using a bootstrap based agent in `genrl` with 10 models with a hidden layer of 128 neurons which also uses dropout for training -

```
from genrl.bandit import BootstrapNeuralAgent, DCBTrainer

agent = BootstrapNeuralAgent(bandit, hidden_dims=[128], n=10, dropout_p=0.5, device=
↪"cuda")

trainer = DCBTrainer(agent, bandit)
trainer.train()
```

Refer to the BootstrapNeuralAgent and DCBTrainer docs for more details.

Parameter Noise Sampling

For an introduction to the Contextual Bandit problem, refer to *Contextual Bandits Overview*.

One of the ways to improve exploration of our algorithms is to introduce noise into the weights of the neural network while selecting actions. This does not affect the gradients but will have a similar effect to epsilon greedy exploration.

The noise distribution is regularly updated during training to keep the KL divergence of the prediction and noise predictions within certain limits.

An example of using a noise sampling based agent in `genrl` with noise standard deviation as 0.1, KL divergence limit as 0.1 and batch size for updating the noise distribution as 128 -

```
from genrl.bandit import BootstrapNeuralAgent, DCBTrainer

agent = NeuralNoiseSamplingAgent(bandit, hidden_dims=[128], noise_std_dev=0.1, eps=0.
↪1, noise_update_batch_size=128, device="cuda")

trainer = DCBTrainer(agent, bandit)
trainer.train()
```

Refer to the NeuralNoiseSamplingAgent, and DCBTrainer docs for more details.

Adding a new Data Bandit

The `bandit` submodule like all of `genrl` has been designed to be easily extensible for custom additions. This tutorial will show how to create a dataset based bandit which will work with the rest of `genrl.bandit`

For this tutorial, we will use the Wine dataset which is a simple datset often used for testing classifiers. It has 178 examples each with 14 features, the first of which gives the cultivar of the wine (the feature we need to classify each wine sample into) (this can be one of three) and the rest give the properties of the wine itself. Formulated as a bandit problem we have a bandit with 3 arms and a 13-dimensional context. The agent will get a reward of 1 if it correctly selects the arm else 0.

To start off with lets import necessary modules, specify the data URL and make a class which inherits from `genrl.utils.data_bandits.base.DataBasedBandit`

```
from typing import Tuple

import pandas as pd
import torch

from genrl.utils.data_bandits.base import DataBasedBandit
from genrl.utils.data_bandits.utils import download_data
```

(continues on next page)

(continued from previous page)

```
URL = "http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"

class WineDataBandit(DataBasedBandit):
    def __init__(self, **kwargs):

    def reset(self) -> torch.Tensor:

    def _compute_reward(self, action: int) -> Tuple[int, int]:

    def _get_context(self) -> torch.Tensor:
```

We will need to implement `__init__`, `reset`, `_compute_reward` and `_get_context` to make the class functional.

For dataset based bandits, we can generally load the data into memory during initialisation. This can be in some tabular form (`numpy.array`, `torch.Tensor` or `pandas.DataFrame`) and maintaining an index. When reset, the bandit would set its index to 0 and reshuffle the rows of the table. For stepping, the bandit can compute rewards from the current row of the table as given by the index and then increment the index to move to the next row.

Lets start with `__init__`. Here we need to download the data if specified and load it into memory. Many utility functions are available in `genrl.utils.data_bandits.utils` including `download_data` to download data from a URL as well as functions to fetch data from memory.

For most cases, you can load the data into a `pandas.DataFrame`. You also need to specify the `n_actions`, `context_dim` and `len` here.

```
def __init__(self, **kwargs):
    super(WineDataBandit, self).__init__(**kwargs)

    path = kwargs.get("path", "./data/Wine/")
    download = kwargs.get("download", None)
    force_download = kwargs.get("force_download", None)
    url = kwargs.get("url", URL)

    if download:
        path = download_data(path, url, force_download)

    self._df = pd.read_csv(path, header=None)
    self.n_actions = len(self._df[0].unique())
    self.context_dim = self._df.shape[1] - 1
    self.len = len(self._df)
```

The `reset` method will shuffle the indices of the data and return the counting index to 0. You must have a call to `_reset` here to reset any metrics, counters etc… (which is implemented in the base class)

```
def reset(self) -> torch.Tensor:
    self._reset()
    self.df = self._df.sample(frac=1).reset_index(drop=True)
    return self._get_context()
```

The new bandit does not explicitly need to implement the `step` method since this is already implmented in the base class. We do however need to implement `_compute_reward` and `_get_context` which `step` uses.

In `_compute_reward`, we need to figure out whether the given action corresponds to the correct label for this index or not and return the reward appropriately. This method also return the maxium possible reward in the current context which is used to compute regret.

```
def _compute_reward(self, action: int) -> Tuple[int, int]:
    label = self._df.iloc[self.idx, 0]
    r = int(label == (action + 1))
    return r, 1
```

The `_get_context` method should return a 13-dimensional `torch.Tensor` (in this case) corresponding to the context for the current index.

```
def _get_context(self) -> torch.Tensor:
    return torch.tensor(
        self._df.iloc[self.idx, 1:].values,
        device=self.device,
        dtype=torch.float,
    )
```

Once you are done with the above, you can use the `WineDataBandit` class like you would any other bandit from from `genrl.utils.data_bandits`. You can use it with any of the `cb_agents` as well as training on it with genrl.bandit.DCBTrainer.

Adding a new Deep Contextual Bandit Agent

The `bandit` submodule like all of `genrl` has been designed to be easily extensible for custom additions. This tutorial will show how to create a deep contextual bandit agent which will work with the rest of `genrl.bandit`

For the purpose of this tutorial we will consider a simple neural network based agent. Although this is a simplictic agent, implementation of any level of agent will need to have the following steps.

To start off with lets import necessary modules and make a class which inherits from `genrl.agents.bandits.contextual.base.DCBAgent`

```
from typing import Optional

import torch

from genrl.agents.bandits.contextual.base import DCBAgent
from genrl.agents.bandits.contextual.common import NeuralBanditModel, TransitionDB
from genrl.utils.data_bandits.base import DataBasedBandit

class NeuralAgent(DCBAgent):
    """Deep contextual bandit agent based on a neural network."""

    def __init__(self, bandit: DataBasedBandit, **kwargs):

    def select_action(self, context: torch.Tensor) -> int:

    def update_db(self, context: torch.Tensor, action: int, reward: int):

    def update_params(
        self,
        action: Optional[int] = None,
        batch_size: int = 512,
        train_epochs: int = 20,
    ):
```

We will need to implement `__init__`, `select_action`, `update_db` and `update_param` to make the class functional.

Lets start off with `__init__`. Here we will need to initialise some required parameters (`init_pulls`, `eval_with_dropout`, `t` and `update_count`) along with our transition database and the neural network. For the neural network, you can use the `NeuralBanditModel` class. It packages together many of the functionalities a neural network might require. Refer to the docs for more details.

```
def __init__(self, bandit: DataBasedBandit, **kwargs):
    super(NeuralAgent, self).__init__(bandit, **kwargs)
    self.model = (
        NeuralBanditModel(
            context_dim=self.context_dim,
            n_actions=self.n_actions,
            **kwargs
        )
        .to(torch.float)
        .to(self.device)
    )
    self.eval_with_dropout = kwargs.get("eval_with_dropout", False)
    self.db = TransitionDB(self.device)
    self.t = 0
    self.update_count = 0
```

For the select action function, the agent will pass the context vector through the neural network to produce logits for each action. It will then select the action with highest logit value. Note that it must also increment the timestep, and if take every action atleast `init_pulls` number of times initially.

```
def select_action(self, context: torch.Tensor) -> int:
    """Selects action for a given context"""
    self.model.use_dropout = self.eval_with_dropout
    self.t += 1
    if self.t < self.n_actions * self.init_pulls:
        return torch.tensor(
            self.t % self.n_actions, device=self.device, dtype=torch.int
        )

    results = self.model(context)
    action = torch.argmax(results["pred_rewards"]).to(torch.int)
    return action
```

For updating the databse we can use the `add` method of `TransitionDB` class.

```
def update_db(self, context: torch.Tensor, action: int, reward: int):
    """Updates transition database."""
    self.db.add(context, action, reward)
```

In `update_params` we need to train the model on the observations seen so far. Since the `NeuralBanditModel` class already hass a train function, we just need to call that. However if you are writing your own model, this is where the updates to the parameters would happen.

```
def update_params(
    self,
    action: Optional[int] = None,
    batch_size: int = 512,
    train_epochs: int = 20,
):
    """Update parameters of the agent."""
    self.update_count += 1
    self.model.train_model(self.db, train_epochs, batch_size)
```

Note that some of these functions have unused arguments. The signatures have been decided so as such to ensure generality over all classes of algorithms.

Once you are done with the above, you can use the `NeuralAgent` class like you would any other agent from `genrl.bandit`. You can use it with any of the bandits as well as training it with genrl.bandit.DCBTrainer.

2.3.2 Classical

Q-Learning using GenRL

What is Q-Learning?

Q-Learning is one of the stepping stones for many reinforcement learning algorithms like DQN. AlphaGO is also one of the famous examples that use Q-Learning at the heart.

Essentially, a RL agent take an action on the environment and then collect rewards and update its policy, and over time gets better at collecting higher rewards.

In Q-Learning, we generally maintain a "Q-table" of *Q-values* by mapping them to a (state, action) pair.

A natural question is, What are these *Q-values* ? It is nothing but the "Quality" of an action taken from a particular state. The more the *Q-value* the more chances of getting a better reward.

Q-Table is often initialized with random values/with zeros and as the agent collects rewards via performing actions on the environment we update this Q-Table at the i th step using the following formulation -

$$Q_i(s,a) = (1-\alpha)Q_{i-1}(s,a) + \alpha * (reward + \gamma * max_{a'}Q_{i-1}(s',a'))$$

Here α is the learning rate in ML terms, γ is the discount factor for the rewards and s' is the state reached after taking action a from state s.

FrozenLake-v0 environment

So to demonstrate how easy it is to train a Q-Learning approach in GenRL, we are taking a very simple gym environment.

Description of the environment (from the documentation) -

"The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.

The surface is described using a grid like the following:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise."

Code

Let's import all the usefull stuff first.

```
import gym
from genrl import QLearning                               # for the agent
from genrl.classical.common import Trainer                # for training the agent
```

Now that we have imported all the necessary stuff let's go ahead and define the environment, the agent and an object for the Trainer class.

```
env = gym.make("FrozenLake-v0")
agent = QLearning(env, gamma=0.6, lr=0.1, epsilon=0.1)
trainer = Trainer(
    agent,
    env,
    model="tabular",
    n_episodes=3000,
    start_steps=100,
    evaluate_frequency=100,
)
```

Great so far so good! Now moving towards the training process it is just calling the train method in the trainer class.

```
trainer.train()
trainer.evaluate()
```

That's it! You have successfully trained a Q-Learning agent. You can now go ahead and play with your own environments using GenRL!

SARSA using GenRL

What is SARSA?

SARSA is an acronym for State-Action-Reward-State-Action. It is an on-policy TD control method. Our aim is basically to estimate the Q-value or the utility value for state-action pair using the TD update rule given below.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * Q(S_{t+1}, A_{t+2}) - Q(S_t, A_t)]$$

FrozenLake-v0 environment

So to demonstrate how easy it is to train a SARSA approach in GenRL, we are taking a very simple gym environment.

Description of the environment (from the documentation) -

“The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.

The surface is described using a grid like the following:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise."

Code

Let's import all the usefull stuff first.

```
import gym
from genrl import SARSA                                # for the agent
from genrl.classical.common import Trainer             # for training the agent
```

Now that we have imported all the necessary stuff let's go ahead and define the environment, the agent and an object for the Trainer class.

```
env = gym.make("FrozenLake-v0")
agent = SARSA(env, gamma=0.6, lr=0.1, epsilon=0.1)
trainer = Trainer(
    agent,
    env,
    model="tabular",
    n_episodes=3000,
    start_steps=100,
    evaluate_frequency=100,
)
```

Great so far so good! Now moving towards the training process it is just calling the train method in the trainer class.

```
trainer.train()
trainer.evaluate()
```

That's it! You have successfully trained a SARSA agent. You can now go ahead and play with your own environments using GenRL!

2.3.3 Deep RL Tutorials

Deep Reinforcement Learning Background

Background

The goal of Reinforcement Learning Algorithms is to maximize reward. This is usually achieved by having a policy π_θ perform optimal behavior. Let's denote this optimal policy by π_θ^*. For ease, we define the Reinforcement Learning problem as a Markov Decision Process.

Markov Decision Process

An Markov Decision Process (MDP) is defined by (S, A, r, P_a) where,

- S is a set of States.

- A is a set of Actions.
- $r : S \rightarrow \mathbb{R}$ is a reward function.
- $P_a(s, s')$ is the transition probability that action a in state s leads to state s'.

Often we define two functions, a policy function $\pi_\theta(s, a)$ and $V_{\pi_\theta}(s)$.

Policy Function

The policy is the agent's strategy, we our goal is to make it optimal. The optimal policy is usually denoted by π^*_θ. There are usually 2 types of policies:

Stochastic Policy

The Policy Function is a stochastic variable defining a probability distribution over actions given states i.e. likelihood of every action when an agent is in a particular state. Formally,

$$\pi : S \times A \rightarrow [0, 1]$$

$$a \sim \pi(a|s)$$

Deterministic Policy

The Policy Function maps from States directly to Actions.

$$\pi : S \rightarrow A$$

$$a = \pi(s)$$

Value Function

The Value Function is defined as the expected return obtained when we follow a policy π starting from state S. Usually there are two types of value functions defined State Value Function and a State Action Value Function.

State Value Function

The State Value Function is defined as the expected return starting from only State s.

$$V^\pi(s) = E[R_t]$$

State Action Value Function

The Action Value Function is defined as the expected return starting from a state s and a taking an action a.

$$Q^\pi(s, a) = E[R_t]$$

The Action Value Function is also known as the **Quality** Function as it would denote how good a particular action is for a state s.

Approximators

Neural Networks are often used as approximators for Policy and Value Functions. In such a case, we say these are **parameterised** by θ. For e.g. π_θ.

Objective

The objective is to choose/learn a policy that will maximize a cumulative function of rewards received at each step, typically the discounted reward over a potential infinite horizon. We formulate this cumulative function as

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

where we choose an action according to our policy, $a_t = \pi_\theta(s_t)$.

Vanilla Policy Gradient

For background on Deep RL, its core definitions and problem formulations refer to Deep RL Background

Objective

The objective is to choose/learn a policy that will maximize a cumulative function of rewards received at each step, typically the discounted reward over a potential infinite horizon. We formulate this cumulative function as

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

where we choose the action $a_t = \pi_\theta(s_t)$.

Algorithm Details

Collect Experience

To make our agent learn, we first need to collect some experience in an online fashion. For this we make use of the `collect_rollouts` method. This method is defined in the `OnPolicyAgent` Base Class.

For updation, we would need to compute advantages from this experience. So, we store our experience in a Rollout Buffer. Action Selection ——————-

Note: We sample a **stochastic action** from the distribution on the action space by providing `False` as an argument to `select_action`.

For practical purposes we would assume that we are working with a finite horizon MDP.

Update Equations

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy.

At each update timestep, we get value and log probabilities:

Now, that we have the log probabilities we calculate the gradient of $J(\pi_\theta)$ as:

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right],$$

where τ is the trajectory.

We then update the policy parameters via stochastic gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

The key idea underlying vanilla policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy.

Training through the API

```python
import gym

from genrl.agents import VPG
from genrl.trainers import OnPolicyTrainer
from genrl.environments import VectorEnv

env = VectorEnv("CartPole-v0")
agent = VPG('mlp', env)
trainer = OnPolicyTrainer(agent, env, log_mode=['stdout'])
trainer.train()
```

```
timestep          Episode           loss              mean_reward
0                 0                 9.1853            22.3825
20480             10                24.5517           80.3137
40960             20                24.4992           117.7011
61440             30                22.578            121.543
81920             40                20.423            114.7339
102400            50                21.7225           128.4013
122880            60                21.0566           116.034
143360            70                21.628            115.0562
163840            80                23.1384           133.4202
184320            90                23.2824           133.4202
204800            100               26.3477           147.87
225280            110               26.7198           139.7952
245760            120               30.0402           184.5045
266240            130               30.293            178.8646
286720            140               29.4063           162.5397
307200            150               30.9759           183.6771
327680            160               30.6517           186.1818
348160            170               31.7742           184.5045
368640            180               30.4608           186.1818
389120            190               30.2635           186.1818
```

Advantage Actor Critic

For background on Deep RL, its core definitions and problem formulations refer to Deep RL Background

Objective

The objective is to maximize the discounted cumulative reward function:

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

This comprises of two parts in the Adantage Actor Critic Algorithm:

1. To choose/learn a policy that will increase the probability of landing an action that has higher expected return than the value of just the state and decrease the probability of landing an action that has lower expected return than the value of the state. The Advantage is computed as:

$$A(s, a) = Q(s, a) - V(s)$$

2. To learn a State Action Value Function (in the name of **Critic**) that estimates the future cumulative rewards given the current state and action. This function helps the policy in evaluation potential state, action pairs.

where we choose the action $a_t = \pi_\theta(s_t)$.

Algorithm Details

Action Selection and Values

`ac` here is an object of the `ActorCritic` class, which defined two methods: `get_value` and `get_action` and ofcourse they return the value approximation from the Critic and action from the Actor.

Note: We sample a **stochastic action** from the distribution on the action space by providing `False` as an argument to `select_action`.

For practical purposes we would assume that we are working with a finite horizon MDP.

Collect Experience

To make our agent learn, we first need to collect some experience in an online fashion. For this we make use of the `collect_rollouts` method. This method is defined in the `OnPolicyAgent` Base Class.

For updation, we would need to compute advantages from this experience. So, we store our experience in a Rollout Buffer.

Compute discounted Returns and Advantages

Next we can compute the advantages and the actual discounted returns for each state. This can be done very easily by simply calling `compute_returns_and_advantage`. Note this implementation of the rollout buffer is borrowed from Stable Baselines.

Update Equations

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy.

At each update timestep, we get value and log probabilities:

Now, that we have the log probabilities we calculate the gradient of $J(\pi_\theta)$ as:

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t | s_t) A^{\pi_\theta}(s_t, a_t) \right],$$

where τ is the trajectory.

We then update the policy parameters via stochastic gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

The key idea underlying Advantage Actor Critic Algorithm is to push up the probabilities of actions that lead to higher return than the expected return of that state, and push down the probabilities of actions that lead to lower return than the expected return of that state, until you arrive at the optimal policy.

Training through the API

```
import gym

from genrl.agents import A2C
from genrl.trainers import OnPolicyTrainer
from genrl.environments import VectorEnv

env = VectorEnv("CartPole-v0")
agent = A2C('mlp', env)
trainer = OnPolicyTrainer(agent, env, log_mode=['stdout'])
trainer.train()
```

Proximal Policy Optimization

For background on Deep RL, its core definitions and problem formulations refer to Deep RL Background

Objective

The objective is to maximize the discounted cumulative reward function:

$$E \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

The Proximal Policy Optimization Algorithm is very similar to the Advantage Actor Critic Algorithm except we add multiply the advantages with a ratio between the log probability of actions at experience collection time and at updation time. What this does is - helps in establishing a trust region for not moving too away from the old policy and at the same time taking gradient ascent steps in the directions of actions which result in positive advantages.

where we choose the action $a_t = \pi_\theta(s_t)$.

Algorithm Details

Action Selection and Values

`ac` here is an object of the `ActorCritic` class, which defined two methods: `get_value` and `get_action` and ofcourse they return the value approximation from the Critic and action from the Actor.

Note: We sample a **stochastic action** from the distribution on the action space by providing `False` as an argument to `select_action`.

For practical purposes we would assume that we are working with a finite horizon MDP.

Collect Experience

To make our agent learn, we first need to collect some experience in an online fashion. For this we make use of the `collect_rollouts` method. This method is defined in the `OnPolicyAgent` Base Class.

For updation, we would need to compute advantages from this experience. So, we store our experience in a Rollout Buffer.

Compute discounted Returns and Advantages

Next we can compute the advantages and the actual discounted returns for each state. This can be done very easily by simply calling `compute_returns_and_advantage`. Note this implementation of the rollout buffer is borrowed from Stable Baselines.

Update Equations

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy.

At each update timestep, we get value and log probabilities:

In the case of PPO our loss function is:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \;\; \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right),$$

where τ is the trajectory.

We then update the policy parameters via stochastic gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

Training through the API

```
import gym

from genrl.agents import PPO1
from genrl.trainers import OnPolicyTrainer
from genrl.environments import VectorEnv

env = VectorEnv("CartPole-v0")
agent = PPO1('mlp', env)
trainer = OnPolicyTrainer(agent, env, log_mode=['stdout'])
trainer.train()
```

Deep Q-Networks (DQN)

For background on Deep RL, its core definitions and problem formulations refer to Deep RL Background

Objective

The DQN uses the concept of Q-learning. When the state space is too huge, it require a large number of epochs to explore and update the Q-value of every state even at least once. Hence, we make use of function approximators. DQN uses a neural network as a function approximator and objective is to get as close to the Bellman Expectation of the Q-value function as possible. This is done by minimising the loss function which is defined as

$$E_{(s,a,s',r) \sim D}[r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)]^2$$

Unlike in regular Q-learning, DQNs need more stability while updating so we often use a second neural network which we call our target model.

Algorithm Details

Epsilon-Greedy Action Selection

We choose the greedy action with a probability of $1 - \epsilon$ and the rest of the time, we sample the action randomly. During evaluation, we use only greedy actions to judge how well the agent performs.

Experience Replay

Whenever an experience is played through (during the training loop), the experience is stored in what we call a Replay Buffer.

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

The transitions are later sampled in batches from the replay buffer for updating the network.

Update Q-value Network

Once our Replay Buffer has enough experiences, we start updating the Q-value networks in the following code according to the above objective.

```

        for timestep in range(0, self.max_timesteps, self.env.n_envs):
            self.agent.update_params_before_select_action(timestep)

            action = self.get_action(state, timestep)
            next_state, reward, done, info = self.env.step(action)
```

(continues on next page)

(continued from previous page)

```

            if self.render:
                self.env.render()

            # true_dones contains the "true" value of the dones (game over statuses).␣
↪It is set
            # to False when the environment is not actually done but instead reaches␣
↪the max
            # episode length.
            true_dones = [info[i]["done"] for i in range(self.env.n_envs)]
            self.buffer.push((state, action, reward, next_state, true_dones))

            state = next_state.detach().clone()

            if self.check_game_over_status(done):
                self.noise_reset()

                if self.episodes % self.log_interval == 0:
                    self.log(timestep)

                if self.episodes == self.epochs:
                    break

            if timestep >= self.start_update and timestep % self.update_interval == 0:
                self.agent.update_params(self.update_interval)

            if (
                timestep >= self.start_update
                and self.save_interval != 0
                and timestep % self.save_interval == 0
            ):
                self.save(timestep)

        self.env.close()
        self.logger.close()
```

The function *get_q_values* calculates the Q-values of the states sampled from the replay buffer. The *get_target_q_values* function will get the Q-values of the same states as calculated by the target network. The *update_params* function is used to calculate the MSE Loss between the Q-values and the Target Q-values and updated using Stochastic Gradient Descent.

Training through the API

```
from genrl.agents import DQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = DQN("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

Variants of DQN

Some of the other variants of DQN that we have implemented in the repo are: - Double DQN - Dueling DQN - Prioritized Replay DQN - Noisy DQN - Categorical DQN

For some extensions of the DQN (like DoubleDQN) we have provided the methods in a file under genrl/agents/dqn/utils.py

```
class DuelingDQN(DQN):
    def __init__(self, *args, **kwargs):
        super(DuelingDQN, self).__init__(*args, **kwargs)
        self.dqn_type = "dueling"  # You can choose "noisy" for NoisyDQN and
↪"categorical" for CategoricalDQN
        self._create_model()

    def get_target_q_values(self, *args):
        return ddqn_q_target(self, *args)
```

The above two snippets define the same class. You can find similar APIs for the other variants in the *genrl/deep/agents/dqn* folder.

Double Deep Q-Network

Objective

Double DQN builds upon the notion of Double Q-Learning and extends it to Deep Q-networks. We use function approximators for predicting the Q-values of the states and a function approximator is always corrupted with some noise. Now, when we maximise over the values of state-action pairs while calculating the target for the TD-update, the maximum is taken over the true values plus the noise. Thus, the maximum of a noisy function is always bigger than the maximum of the true function:

$$E[max(X_1, X_2)] \geq max[E(X_1), E(X_2)]$$

where X_1 and X_2 are two random variables. This leads to overestimations of the values of state-action pairs and cnsequently suboptimal action selection. This overestimation is bound to propagate and increase over the course of multiple updates because the same approximator is used to select the maximum action and to estimate it's Q-value.

$$max_{a'}Q_{\phi'}(s', a') = Q_{\phi'}(s', argmax_{a'}Q_{\phi'}(s', a'))$$

This problem can be solved by decoupling the action selection and the value estimation using two separate function approximators(and hence different noise distributions) for both the purposes which is what a Double-DQN does. The loss function is defined as:

$$E_{s,a\sim\rho(.)}[(y^{DoubleDQN} - Q(s, a; \theta))^2]$$

Algorithm Details

Epsilon-Greedy Action Selection

The action exploration is stochastic wherein the greedy action is chosen with a probability of $1 - \epsilon$ and rest of the time, we sample the action randomly. During evaluation, we use only greedy actions to judge how well the agent performs.

Experience Replay

Every transition occuring during the training is stored in a separate *Replay Buffer*

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

The transitions are later sampled in batches from the replay buffer for updating the network.

Update the Q-Network

Doble DQN decouples the selection of the action from the evaluation of the Q-values while calculating the target value for the update. The loss function for a time step t is defined as:

$$L_t(\theta_t) = E_{s,a\sim\rho(.)}[(y_t^{DoubleDQN} - Q(s,a;\theta_t))^2]$$
$$y_t^{DoubleDQN} = R_{t+1} + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t), \theta_t^-)$$

The only thing that differs with DoubleDQN is the *get_target_q_values* function as shown below.

```
from genrl.agents import DQN
from genrl.trainers import OffPolicyTrainer

class DoubleDQN(DQN):
    def __init__(self, *args, **kwargs):
        super(DoubleDQN, self).__init__(*args, **kwargs)
        self._create_model()

    def get_target_q_values(self, next_states, rewards, dones):
        next_q_value_dist = self.model(next_states)
        next_best_actions = torch.argmax(next_q_value_dist, dim=-1).unsqueeze(-1)

        rewards, dones = rewards.unsqueeze(-1), dones.unsqueeze(-1)

        next_q_target_value_dist = self.target_model(next_states)
        max_next_q_target_values = next_q_target_value_dist.gather(2, next_best_
↪actions)
        target_q_values = rewards + agent.gamma * torch.mul(
            max_next_q_target_values, (1 - dones)
        )
        return target_q_values
```

Training through the API

```
from genrl.agents import DoubleDQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = DoubleDQN("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

```
timestep          Episode           value_loss        epsilon           Episode Reward
24                0.0               0                 0.9766            0
720               25.0              0                 0.5184            26.96
1168              50.0              0.49              0.1646            18.6
3248              75.0              4.1546            0.0326            74.88
7512              100.0             7.3164            0.0102            166.36
12424             125.0             12.3175           0.01              200.0
Evaluated for 10 episodes, Mean Reward: 200.0, Std Deviation for the Reward: 0.0
```

Dueling Deep Q-Network

Objective

The main objective of DQN is to learn a function approximator for the Q-function using a neural network. This is done by training the approximator to get as close to the Bellman Expectation of the Q-value function as possible by minimising the loss which is defined as:

$$E_{(s,a,s',r) \sim D}[r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)]^2$$

Dueling Deep Q-network modifies the architecture of a simple DQN into one better suited for model-free RL

Algorithm Details

Network architechture

The Dueling DQN architechture splits the single stream of fully connected layers in a normal DQN into two separate streams : one representing the value function and the other representing the advantage function. Advantage function.

$$A(s, a) = Q(s, a) - V(s, a)$$

The advantage for a state action pair represents how beneficial it is to take an action over others when in a particular state. The dueling architechture can learn which states are or are not valuable without having to learn the effect of action for each state. This is useful in instances when taking any action would affect the environment in any significant way.

Another layer combines the value stream and the advantage stream to get the Q-values

Combining the value and the advantage streams

- Value Function : $V(s; \theta, \beta)$

- Advantage Function : $A(s, a; \theta, \alpha)$

where θ denotes the parameters of the underlying convolutional layers whereas α and β are the parameters of the two separate streams of fully connected layers

The two stream cannot be simply added ($Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$) to get the Q-values because:

- $Q(s, a; \theta, \alpha, \beta)$ is only a parameterized estimate of the true Q-function
- It would be wrong to assume that $V(s; \theta, \beta)$ and $Q(s, a; \theta, \alpha)$ are reasonable estimates of the value and the advantage functions

To address these concerns, we train in order to force the expected value of the advantage function to be zero (the expectation of advantage is always zero)

Thus, the combining module combines the value and advantage streams to get the Q-values in the following fashion:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - max_{a' \in |A|} A(s, a'; \theta, \alpha))$$

Epsilon-Greedy Action Selection

Similar to a normal DQN, the action exploration is stochastic wherein the greedy action is chosen with a probability of $1 - \epsilon$ and rest of the time, we sample the action randomly. During evaluation, we use only greedy actions to judge how well the agent performs.

Experience Replay

Every transition occuring during the training is stored in a separate *Replay Buffer*

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

The transitions are later sampled in batches from the replay buffer for updating the network

Update the Q Network

Once enough number of transitions ae stored in the replay buffer, we start updating the Q-values according to the given objective. The loss function is defined in a fashion similar to a DQN. This allows any new improvisations in training techniques of DQN such as Double DQN or NoisyNet DQN to be readily adapted in the dueling architechture.

```

        for timestep in range(0, self.max_timesteps, self.env.n_envs):
            self.agent.update_params_before_select_action(timestep)
```

(continues on next page)

(continued from previous page)

```

            action = self.get_action(state, timestep)
            next_state, reward, done, info = self.env.step(action)

            if self.render:
                self.env.render()

            # true_dones contains the "true" value of the dones (game over statuses).␣
↪It is set
            # to False when the environment is not actually done but instead reaches␣
↪the max
            # episode length.
            true_dones = [info[i]["done"] for i in range(self.env.n_envs)]
            self.buffer.push((state, action, reward, next_state, true_dones))

            state = next_state.detach().clone()

            if self.check_game_over_status(done):
                self.noise_reset()

                if self.episodes % self.log_interval == 0:
                    self.log(timestep)

                if self.episodes == self.epochs:
                    break

            if timestep >= self.start_update and timestep % self.update_interval == 0:
                self.agent.update_params(self.update_interval)

            if (
                timestep >= self.start_update
                and self.save_interval != 0
                and timestep % self.save_interval == 0
            ):
                self.save(timestep)

        self.env.close()
        self.logger.close()
```

Training through the API

```
from genrl.agents import DuelingDQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = DuelingDQN("mlp", env)
trainer = OffpolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

Deep Q Networks with Noisy Nets

Objective

NoisyNet DQN is a variant of DQN which uses fully connected layers with noisy parameters to drive exploration. Thus, the parametrised action-value function can now be seen as a random variable. The new loss function which needs to minimised is defined as:

$$E[E_{(x,a,r,y)\sim D}[r + \gamma max_{b\in A}Q(y,b,\epsilon';\zeta^{-}) - Q(x,a,\epsilon;\zeta)]^2]$$

where ζ is a set of learnable parameters for the noise.

Algorithm Details

Action Selection

The action selection is no longer epsilon-greedy since the exploration is driven by the noise in the neural network layers. The action selection is done greedily.

Noisy Parameters

A noisy parameter θ is defined as:

$$\theta := \mu + \Sigma \odot \epsilon$$

where Σ and μ are vectors of trainable parameters and ϵ is a vector of zero mean noise. Hence, the loss function is now defined with respect to Σ and μ and the optimization now takes place with respect to Σ and μ. ϵ is sampled from factorised gaussian noise.

Experience Replay

Every transition occuring during the training is stored in a separate *Replay Buffer*

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

The transitions are later sampled in batches from the replay buffer for updating the network

Update the Q-Network

Once enough number of transitions ae stored in the replay buffer, we start updating the Q-values according to the given objective. The loss function is defined in a fashion similar to a DQN. This allows any new improvisations in training techniques of DQN such as Double DQN or NoisyNet DQN to be readily adapted in the dueling architechture.

```

        for timestep in range(0, self.max_timesteps, self.env.n_envs):
            self.agent.update_params_before_select_action(timestep)

            action = self.get_action(state, timestep)
            next_state, reward, done, info = self.env.step(action)

            if self.render:
                self.env.render()

            # true_dones contains the "true" value of the dones (game over statuses).␣
↪It is set
            # to False when the environment is not actually done but instead reaches␣
↪the max
            # episode length.
            true_dones = [info[i]["done"] for i in range(self.env.n_envs)]
            self.buffer.push((state, action, reward, next_state, true_dones))

            state = next_state.detach().clone()

            if self.check_game_over_status(done):
                self.noise_reset()

                if self.episodes % self.log_interval == 0:
                    self.log(timestep)

                if self.episodes == self.epochs:
                    break

            if timestep >= self.start_update and timestep % self.update_interval == 0:
                self.agent.update_params(self.update_interval)

            if (
                timestep >= self.start_update
                and self.save_interval != 0
                and timestep % self.save_interval == 0
            ):
                self.save(timestep)

        self.env.close()
        self.logger.close()
```

Training through the API

```
from genrl.agents import NoisyDQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = NoisyDQN("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

Prioritized Deep Q-Networks

Objective

The main motivation behind using prioritized experience replay over uniformly sampled experience replay stems from the fact that an agent may be able to learn more from some transitions than others. In uniformly sampled experience replay, some transitions which might not be very useful for the agent or that might be redundant will be replayed with the same frequency as those having more learning potential. Prioritized experience replay solves this problem by replaying more useful transitions more frequently.

The loss function for prioritized DQN is defined as

$$E_{(s,a,s',r,p)\sim D}[r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)]^2$$

Algorithm Details

Epsilon-Greedy Action Selection

The action exploration is stochastic wherein the greedy action is chosen with a probability of $1 - \epsilon$ and rest of the time, we sample the action randomly. During evaluation, we use only greedy actions to judge how well the agent performs.

Prioritized Experience Replay

The replay buffer is no longer uniformly sampled, but is sampled according to the *priority* of a transition. Transitions with greater scope of learning are assigned a higher priorities. Priority of a particular transition is decided using the TD-error since the measure of the magnitude of the TD error can be interpreted as how unexpected the transition is.

$$\delta = R + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)$$

The transition with the maximum TD-error is given the maximum priority. Every new transition is given the highest priority to ensure that each transition is considered at-least once.

Stochastic Prioritization

Sampling transition greedily has some disadvantages such as transitions having a low TD-error on the first replay might not be sampled ever again, higher chances of overfitting since only a small set of transitions with high priorities are replayed over and over again and sensitivity to noise spikes. To tackle these problems, instead of sampling transitions greedily everytime, we use a stochastic approach wherein each transition is assigned a certain probability with which it is sampled. The sampling probability is defined as

$$P(i) = \frac{p_i^\alpha}{\Sigma_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. α determines the amount of prioritization done. The priority of the transition can be defined in the following two ways:

- $p_i = |\delta_i| + \epsilon$
- $p_i = \frac{1}{rank(i)}$

where ϵ is a small positive constant to ensure that the sampling probability is not zero for any transition and $rank(i)$ is the rank of the transition when the replay buffer is sorted with respect to priorities.

We also use importance sampling (IS) weights to correct certain bais introduced by prioritized experience replay.

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta$$

Update the Q-value Networks

The importance sampling weights can be folded into the Q-learning update by using $w\delta_i$ instead of δ_i. Once our Replay Buffer has enough experiences, we start updating the Q-value networks in the following code according to the above objective.

```

        for timestep in range(0, self.max_timesteps, self.env.n_envs):
            self.agent.update_params_before_select_action(timestep)

            action = self.get_action(state, timestep)
            next_state, reward, done, info = self.env.step(action)

            if self.render:
                self.env.render()

            # true_dones contains the "true" value of the dones (game over statuses).␣
↪It is set
            # to False when the environment is not actually done but instead reaches␣
↪the max
            # episode length.
            true_dones = [info[i]["done"] for i in range(self.env.n_envs)]
            self.buffer.push((state, action, reward, next_state, true_dones))

            state = next_state.detach().clone()

            if self.check_game_over_status(done):
                self.noise_reset()

                if self.episodes % self.log_interval == 0:
                    self.log(timestep)

                if self.episodes == self.epochs:
                    break

            if timestep >= self.start_update and timestep % self.update_interval == 0:
                self.agent.update_params(self.update_interval)

            if (
                timestep >= self.start_update
                and self.save_interval != 0
                and timestep % self.save_interval == 0
            ):
                self.save(timestep)

        self.env.close()
        self.logger.close()
```

Training through the API

```
from genrl.agents import PrioritizedReplayDQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = PrioritizedReplayDQN("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

Deep Deterministic Policy Gradients

Objective

Deep Deterministic Policy Gradients (DDPG) is a model-free actor-critic algorithm which deals with continuous action spaces. One simple approach of dealing with continuous action spaces can be discretizing the action space. However, this gives rise to several problems, the most significant being that the size of the action-space increases exponentially with the number of degrees of freedom. DDPG builds up on *Deterministic Policy Gradients* to learn deterministic policies in high-dimensional continuous action-spaces.

Algorithms Details

Deterministic Policy Gradient

In cases with continuous action-spaces, using Q-learning like approach (greedy policy improvement) to learn deterministic policies is not feasible since it involves selecting the action with the maximum action value function at every step and it is not possible to check the action value for every possible action in case of continuous action spaces.

$$\mu^{k+1}(s) = argmax_a Q^{\mu^k}(s, a)$$

This problem can be solved by considering the fact that a policy can be improved by moving it in the direction of increasing action-value function:

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t, \theta^\mu)}]$$

Action Selection

To ensure sufficient exploration, noise is added to the action selected using the current policy. The noise is sampled from a noise process $\mathcal{N}$:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

$\mathcal{N}$ can be chosen to suit the environment (for eg. Ornstein-Uhlenbeck process, Gaussian noise, etc.)

```
    def select_action(
        self, state: torch.Tensor, deterministic: bool = True
    ) -> torch.Tensor:
        """Select action given state
```

(continues on next page)

(continued from previous page)

```

        Deterministic Action Selection with Noise

        Args:
            state (:obj:`torch.Tensor`): Current state of the environment
            deterministic (bool): Should the policy be deterministic or stochastic

        Returns:
            action (:obj:`torch.Tensor`): Action taken by the agent
        """
        action, _ = self.ac.get_action(state, deterministic)
        action = action.detach()

        # add noise to output from policy network
        if self.noise is not None:
            action += self.noise()

        return torch.clamp(
            action, self.env.action_space.low[0], self.env.action_space.high[0]
        )
```

Experience Replay

Similar to DQNs, DDPG being an off-policy algorithm, makes use of *Replay Buffers*. Whenever a transition (s_t, a_t, r_t, s_{t+1}) is encountered, it is stored into the replay buffer. Batches of these transitions are sampled while updating the network parameters. This helps in breaking the strong correlation between the updates that would have been present had the transitions been trained and discarded immediately after they are encountered and also helps to avoid the rapid forgetting of the possibly rare transitions that would be useful later on.

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

Update the Value and Policy Networks

DDPG makes use of target networks for the actor(policy) and the critic(value) networks to stabilise the training. The Q-network is update using TD-learning updates. The target and the loss function for the same are defined as:

$$L(\theta^Q) = \mathbb{E}_{(s_t \sim \rho^\beta, a_t \sim \beta, t_t \sim R)}[(Q(s_t, a_t|\theta^Q) - y_t)^2]$$

$$y_t = r(s_t, a_t) + \gamma Q_{targ}(s_{t+1}, \mu_{targ}(s_{t+1})|\theta^Q)$$

Buliding up on Deterministic Policy Gradients, the gradient of the policy can be determined using the action-value function as

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]$$

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}]$$

The target networks are updated at regular intervals␣

```

        for timestep in range(0, self.max_timesteps, self.env.n_envs):
            self.agent.update_params_before_select_action(timestep)

            action = self.get_action(state, timestep)
            next_state, reward, done, info = self.env.step(action)

            if self.render:
                self.env.render()

            # true_dones contains the "true" value of the dones (game over statuses).␣
↪It is set
            # to False when the environment is not actually done but instead reaches␣
↪the max
            # episode length.
            true_dones = [info[i]["done"] for i in range(self.env.n_envs)]
            self.buffer.push((state, action, reward, next_state, true_dones))

            state = next_state.detach().clone()

            if self.check_game_over_status(done):
                self.noise_reset()

                if self.episodes % self.log_interval == 0:
                    self.log(timestep)

                if self.episodes == self.epochs:
                    break

            if timestep >= self.start_update and timestep % self.update_interval == 0:
                self.agent.update_params(self.update_interval)

            if (
                timestep >= self.start_update
                and self.save_interval != 0
                and timestep % self.save_interval == 0
            ):
                self.save(timestep)

        self.env.close()
        self.logger.close()
```

Training through the API

```
from genrl.agents import DDPG
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer
```

(continues on next page)

(continued from previous page)

```
env = VectorEnv("MountainCarContinuous-v0")
agent = DDPG("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

Twin Delayed DDPG

Objective

Similar to Deep Q-Networks, the problem of overestimation of the state values, occuring due to noisy function approximators and using the same function approximator for action selection and value estimation also persists in actor-critic algorithms with continuous action-spaces. Double DQN, the solution for this problem in Deep Q-Networks is not effective in actor-critic algorithms due to the slow rate of change of the policy. Twin Delayed DDPG (TD3) uses *Clipped Double Q-Learning* to address this problem. TD3 uses two Q function approximators and the loss function for each is given by

$$L(\phi_1, \mathcal{D}) = E_{(s,a,r,s',d)\sim\mathcal{D}}[(Q_{\phi_1}(s,a) - y(r,s',d))^2]$$

$$L(\phi_2, \mathcal{D}) = E_{(s,a,r,s',d)\sim\mathcal{D}}[(Q_{\phi_2}(s,a) - y(r,s',d))^2]$$

Algorithm Details

Clipped Double Q-Learning

Double DQNs are not effective in actor-critic algorithms due to the slow change in the policy and the original double Q-Learning (although being somewhat effective) does not completely solve the problem of overestimation. To tackle this TD3 uses *Clipped Double Q-Learning* Clipped Double Q-Learning proposes to upper bound the less biased critic network by the more biased one and hence no additional overestimation can be introdiced. Although, this may introduce underestimation, it is not much of a concern since underestimation errors don't propagate through policy updates. The target function calculated usign Clipped Double Q-Learning for the updates can be written as

$$y = r + \gamma min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi_1}(s'))$$

Both of the critic networks are updated using the loss functions mentioned above.

Experience Replay

TD3 being an off-policy algorithm, makes use of *Replay Buffer*. Whenever a transition (s_t, a_t, r_t, s_{t+1}) is encountered, it is stored into the replay buffer. Batches of these transitions are sampled while updating the network parameters. This helps in breaking the strong correlation between the updates that would have been present had the transitions been trained and discarded immediately after they are encountered and also helps to avoid the rapid forgetting of the possibly rare transitions that would be useful later on.

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.
```

(continues on next page)

(continued from previous page)

```

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

Target Policy Smoothing Regularization

TD3 adds noise to the target action to reduce the variance induced by function approximation error. This acts as a form of regularization which smoothens the changes in the action-values along changes in action

$$a = \pi_{\phi'}(s') + \epsilon$$

$$\epsilon \sim clip(\mathcal{N}(0, \sigma), -c, c)$$

Delayed Policy updates

TD3 uses target networks similar to DDPG and DQNs for the two critics and the actors to stabilise learning. Apart from this, it also promotes updating the policy networks at a lower frequency as compared to the Q-networks to avoid divergent behaviour for the policy. The paper recommends one policy update for every two Q-function updates.

```

    def update_params(self, update_interval: int) -> None:
        """Update parameters of the model

        Args:
            update_interval (int): Interval between successive updates of the target␣
↪model
        """
        for timestep in range(update_interval):
            batch = self.sample_from_buffer()

            value_loss = self.get_q_loss(batch)

            self.optimizer_value.zero_grad()
            value_loss.backward()
            self.optimizer_value.step()

            # Delayed Update
            if timestep % self.policy_frequency == 0:
                policy_loss = self.get_p_loss(batch.states)

                self.optimizer_policy.zero_grad()
                policy_loss.backward()
                self.optimizer_policy.step()

                self.logs["policy_loss"].append(policy_loss.item())
                self.logs["value_loss"].append(value_loss.item())

```

Training through the API

```
from genrl.agents import TD3
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("MountainCarContinuous-v0")
agent = TD3("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=4000)
trainer.train()
trainer.evaluate()
```

Soft Actor-Critic

Objective

Deep Reinforcement Learning Algorithms suffer from two main problems : one being high sample complexity (large amounts of data needed) and the other being thier brittleness with respect to learning rates, exporation constants and other hyperparameters. Algorithms such as DDPG and Twin Delayed DDPG are used to tackle the challenge of high sample complexity in actor-critic frameworks with continuous action-spaces. However, they still suffer from brittle stability with respect to their hyperparameters. Soft-Actor Critic introduces a actor-critic framework for arrangements with continuous action spaces wherein the standard objective of reinforcement learning, i.e., maximizing expected cumulative reward is augmented with an additional objective of entropy maximization which provides a substantial improvement in exploration and robustness. The objective can be mathematically represented as

$$J(\pi) = \Sigma_{t=0}^{T}\gamma^{t}\mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))]$$

where α also known as the temperature parameter determines the relative importance of the entropy term against the reward, and thus controls the stochasticity of the optimal policy and $\mathcal{H}$ represents the entropy function. The entropy of a random variable $\S$ following a probability distribution P is defined as

$$\mathcal{H}(P) = \mathbb{E}_{\S\sim P}[-logP(\S)]$$

Algorithm Details

Soft Actor-Critic is mostly used in two variants depending on whether the temperature constant α is kept constant throughout the learning process or if it is learned as a parameter over the course of learning. GenRL uses the latter one.

Action-Value Networks

SAC learns a ploicy π_θ and two Q functions Q_{ϕ_1}, Q_{ϕ_2} and their target networks concurrently. The two Q-functions are learned in a fashion similar to TD3 where a common target is considered for both the Q functions and *Clipped Double Q-learning* is used to train the network. However, unlike TD3, the next-state actions used in the target are calculated using the current policy. Since, the optimisation objective also involves maximising the entropy, the new Q-value can be expressed as

$$Q^{\pi}(s, a) = \mathbb{E}_(s' \sim P, a' \sim \pi)[R(s, a, s') + \gamma(Q^{\pi}(s', a') + \alpha\mathcal{H}(\pi(\cdot|s')))]$$

$$Q^{\pi}(s, a) = \mathbb{E}_(s' \sim P, a' \sim \pi)[R(s, a, s') + \gamma(Q^{\pi}(s', a') + \alpha log\pi(a'|s'))]$$

Thus, the action-value for one state-action pair can be approximated as

$$Q^{\pi}(s,a) \approx r + \gamma(Q^{\pi}(s',\tilde{a}') - \alpha log\pi(\tilde{a}'|s'))$$

where $\tilde{a}'$ (action taken in next state) is sampled from the policy.

Experience Replay

SAC also uses *Replay Buffer* like other off-policy algorithms. Whenever a transition (s_t, a_t, r_t, s_{t+1}) is encountered, it is stored into the replay buffer. Batches of these transitions are sampled while updating the network parameters. This helps in breaking the strong correlation between the updates that would have been present had the transitions been trained and discarded immediately after they are encountered and also helps to avoid the rapid forgetting of the possibly rare transitions that would be useful later on.

```
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

Q-Network Optimisation

Just like TD3, SAC uses *Clipped Double Q-Learning* to calculate the target values for the Q-value network

$$y^t(r,s',d) = r + \gamma(min_{j=1,2}Q_{\phi_{targ,j}}(s',\tilde{a}') - \alpha log\pi_{\theta}(\tilde{a}'|s'))$$

where $\tilde{a}'$ is sampled from the policy. The loss function can then be defined as

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}[(Q_{\phi_i}(s,a) - y^t(r,s',d))^2]$$

Action Selection and Policy Optimisation

The main aim of policy optimisation will be maximise the value function which in this case can be defined as

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a) - log\pi(a|s)]$$

In SAC, a **reparameterisation trick** is used to sample actions from the policy to ensure that sampling from the policy is a differentiable process. The policy is now parameterised as

$$\tilde{a}'_t = \{_{\theta}(\xi_t; s_t)$$

$$\tilde{a}'_{\theta}(s,\xi) = tanh(\mu_{\theta}(s) + \sigma_{\theta}(s) \odot \xi)$$

$$\xi \sim \mathcal{N}(0,1)$$

The maximisation objective is now defined as

$$max_{\theta}\mathbb{E}_{(s\sim\mathcal{D},\xi\sim\mathcal{N})}[min_{j=1,2}Q_{\phi_j}(s,\tilde{a}_{\theta}(s,\xi)) - \alpha log\pi_{\theta}(\tilde{a}_{\theta}(s,\xi)|s)]$$

Training through the API

```python
from genrl.agents import SAC
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("MountainCarContinuous-v0")
agent = SAC("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=4000)
trainer.train()
trainer.evaluate()
```

Categorical Deep Q-Networks

Objective

The main objective of Categorical Deep Q-Networks is to learn the distribution of Q-values as unlike to other variants of Deep Q-Networks where the goal is is to approximate the *expectations* of the Q-values as closely as possible. In complicated environments, the Q-values can be stochastic and in that case, simply learning the expectation of Q-values will not be able to capture all the information needed (for eg. variance of the distribution) to make an optimal decision.

Distributional Bellman

The bellman equation can be adapted to this form as

$$Z(x, a) \overset{D}{=} R(x, a) + \gamma Z(x', a')$$

where $Z(s, a)$ (the value distribution) and $R(s, a)$ (the reward distribution) are now probability distributions. The equality or similarity of two distributions can be effectivelyevaluated using the Kullback-Leibler(KL) - divergence or the cross-entropy loss.

$$Q^{\pi}(x, a) := \mathbb{E}Z^{\pi}(x, a) = \mathbb{E}\left[\sum_{t=0}^{\inf} \gamma^t R(x_t, a_t)\right]$$

z sim P(odot vert x_{t-1}, a_{t-1}). a_t sim pi(odot vert x_t), x_0 = x, a_0 =a

The transition operator $P^{\pi} : \mathcal{Z} \rightarrow \mathcal{Z}$ and the bellman operator $\mathcal{T} : \mathcal{Z} \rightarrow \mathcal{Z}$ can be defined as

$$P^{\pi} Z(x, a) :\overset{D}{=} Z(X', A'); X' \sim P(\odot | x, a), A' \sim \pi(\odot | X')$$

$$\mathcal{T}^{\pi} Z(x, a) :\overset{D}{=} R(x, a) + \gamma P^{\pi} Z(x, a)$$

Algorithm Details

Parametric Distribution

Categorical DQN uses a discrete distribution parameterized by a set of supports/*atoms* (discrete values) to model the value distribution. This set of atoms is determined as

$$\ddagger_i = V_{MIN} + i\nabla\ddagger : 0 \leq i < N; \nabla\ddagger := \frac{V_{MAX} - V_{MIN}}{N - 1}$$

where $N \in \mathbb{N}$ and $V_{MAX}, V_{MIN} \in \mathbb{R}$ are the distribution parameters. The probability of each atom is modeled as

$$Z_\theta(x,a) = \ddagger_i w.p. p_i(x,a) := \frac{\exp \theta_i(x,a)}{\sum_j \exp \theta_j(x,a)}$$

Action Selection

GenRL uses greedy action selection for categorical DQN wherein the action with the highest Q-values for all discrete regions is selected.

```python
def categorical_greedy_action(agent: DQN, state: torch.Tensor) -> torch.Tensor:
    """Greedy action selection for Categorical DQN

    Args:
        agent (:obj:`DQN`): The agent
        state (:obj:`torch.Tensor`): Current state of the environment

    Returns:
        action (:obj:`torch.Tensor`): Action taken by the agent
    """
    q_value_dist = agent.model(state.unsqueeze(0)).detach()  # .numpy()
    # We need to scale and discretise the Q-value distribution obtained above
    q_value_dist = q_value_dist * torch.linspace(
        agent.v_min, agent.v_max, agent.num_atoms
    )
    # Then we find the action with the highest Q-values for all discrete regions
    # Current shape of the q_value_dist is [1, n_envs, action_dim, num_atoms]
    # So we take the sum of all the individual atom q_values and then take argmax
    # along action dim to get the optimal action. Since batch_size is 1 for this
    # function, we squeeze the first dimension out.
    action = torch.argmax(q_value_dist.sum(-1), axis=-1).squeeze(0)
    return action
```

Experience Replay

Categorical DQN like other DQNs uses *Replay Buffer* like other off-policy algorithms. Whenever a transition (s_t, a_t, r_t, s_{t+1}) is encountered, it is stored into the replay buffer. Batches of these transitions are sampled while updating the network parameters. This helps in breaking the strong correlation between the updates that would have been present had the transitions been trained and discarded immediately after they are encountered and also helps to avoid the rapid forgetting of the possibly rare transitions that would be useful later on.

```python
    def log(self, timestep: int) -> None:
        """Helper function to log

        Sends useful parameters to the logger.

        Args:
            timestep (int): Current timestep of training
        """
        self.logger.write(
            {
                "timestep": timestep,
                "Episode": self.episodes,
                **self.agent.get_logging_params(),
                "Episode Reward": safe_mean(self.training_rewards),
```

Projected Bellman Update

The sample bellman update $\hat{\mathcal{T}}Z_\theta$ is projected onto the support of Z_θ for the update as shown in the figure below. The bellman update for each atom j can be calculated as

$$\hat{\mathcal{T}}\ddagger_{|} := r + \gamma\ddagger_{|}$$

and then it's probability $\surd_{|}(x', \pi x')$ is distributed to the neighbours of the update. Here, (x, a, r, x') is a sample transition. The i^{th} component of the projected update is calculated as

$$(\Phi\hat{\mathcal{T}}Z_\theta(x,a))_i = \sum_{j=0}^{N-1}\left[1 - \frac{|\left[\hat{\mathcal{T}}\ddagger_{|}\right]_{V_{MIN}}^{V_{MAX}} - \ddagger_{\rangle}|}{\Delta\ddagger}\right]_0^1 \surd_{|}(x', \pi(x'))$$

The loss is calculated using KL divergence (cross entropy loss). This is also known as the **Bernoulli algorithm**

$$D_{KL}(\Phi\hat{\mathcal{T}}Z_{\tilde{\theta}(x,a)||Z_\theta(x,a))}$$

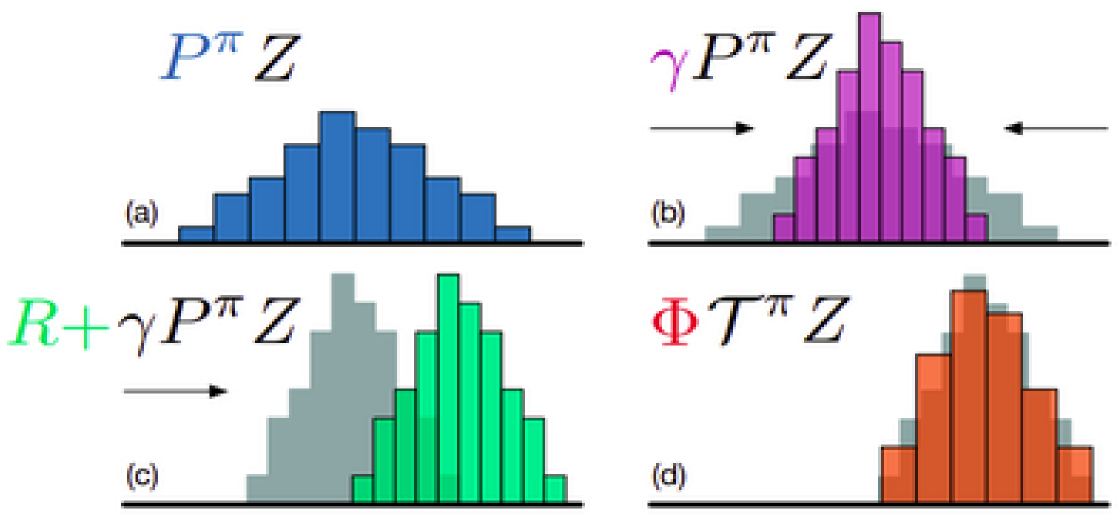

Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

```
def categorical_q_target(
    agent: DQN,
    next_states: torch.Tensor,
    rewards: torch.Tensor,
    dones: torch.Tensor,
):
```

(continues on next page)

(continued from previous page)

```
    """Projected Distribution of Q-values

    Helper function for Categorical/Distributional DQN

    Args:
        agent (:obj:`DQN`): The agent
        next_states (:obj:`torch.Tensor`): Next states being encountered by the agent
        rewards (:obj:`torch.Tensor`): Rewards received by the agent
        dones (:obj:`torch.Tensor`): Game over status of each environment

    Returns:
        target_q_values (object): Projected Q-value Distribution or Target Q Values
    """
    delta_z = float(agent.v_max - agent.v_min) / (agent.num_atoms - 1)
    support = torch.linspace(agent.v_min, agent.v_max, agent.num_atoms)

    next_q_value_dist = agent.target_model(next_states) * support
    next_actions = (
        torch.argmax(next_q_value_dist.sum(-1), axis=-1).unsqueeze(-1).unsqueeze(-1)
    )

    next_actions = next_actions.expand(
        agent.batch_size, agent.env.n_envs, 1, agent.num_atoms
    )
    next_q_values = next_q_value_dist.gather(2, next_actions).squeeze(2)

    rewards = rewards.unsqueeze(-1).expand_as(next_q_values)
    dones = dones.unsqueeze(-1).expand_as(next_q_values)

    # Refer to the paper in section 4 for notation
    Tz = rewards + (1 - dones) * 0.99 * support
    Tz = Tz.clamp(min=agent.v_min, max=agent.v_max)
    bz = (Tz - agent.v_min) / delta_z
    l = bz.floor().long()
    u = bz.ceil().long()

    offset = (
        torch.linspace(
            0,
            (agent.batch_size * agent.env.n_envs - 1) * agent.num_atoms,
            agent.batch_size * agent.env.n_envs,
        )
        .long()
        .view(agent.batch_size, agent.env.n_envs, 1)
        .expand(agent.batch_size, agent.env.n_envs, agent.num_atoms)
    )

    target_q_values = torch.zeros(next_q_values.size())
    target_q_values.view(-1).index_add_(
        0,
        (l + offset).view(-1),
        (next_q_values * (u.float() - bz)).view(-1),
    )
    target_q_values.view(-1).index_add_(
        0,
        (u + offset).view(-1),
        (next_q_values * (bz - l.float())).view(-1),
```

(continues on next page)

(continued from previous page)

```
    )
    return target_q_values

```

Training through the API

```
from genrl.agents import CategoricalDQN
from genrl.environments import VectorEnv
from genrl.trainers import OffPolicyTrainer

env = VectorEnv("CartPole-v0")
agent = CategoricalDQN("mlp", env)
trainer = OffPolicyTrainer(agent, env, max_timesteps=20000)
trainer.train()
trainer.evaluate()
```

2.3.4 Custom Policy Networks

GenRL provides default policies for images (CNNPolicy) and for other types of inputs(MlpPolicy). Sometimes, these default policies may be insuffiecient for your problem, or you may want more control over the policy definition, and hence require a custom policy.

The following code tutorial runs through the steps to use a custom policy depending on your problem.

Import the required libraries (eg. torch, torch.nn) and from GenRL, the algorithm (eg VPG), the trainer (eg. OnPolicyTrainer), the policy to be modified (eg. MlpPolicy)

```
# The necessary imports
import torch
import torch.nn as nn

from genrl.agents import VPG
from genrl.core.policies import MlpPolicy
from genrl.environments import VectorEnv
from genrl.trainers import OnPolicyTrainer
```

Then define a `custom_policy` class that derives from the policy to be modified (in this case, the `MlpPolicy`)

```
# Define a custom MLP Policy
class custom_policy(MlpPolicy):
    def __init__(self, state_dim, action_dim, hidden, **kwargs):
        super().__init__(state_dim, action_dim, hidden)
        self.action_dim = action_dim
        self.state_dim = state_dim
```

The above class modifies the MlpPolicy to have the desired number of hidden layers in the MLP Neural network that parametrizes the policy. This is done by passing the variable hidden explicitly (default`hidden = (32, 32)`). The `state_dim` and `action_dim` variables stand for the dimensions of the state_space and the action_space, and are required to construct the neural network with the proper input and output shapes for your policy, given the environment.

In some cases, you may also want to redefine the policy used completely and not just customize and existing policy. This can be done by creating a new custom policy class that inhierits the BasePolicy class. The BasePolicy class is a basic implementation of a general policy, with a `forward` and a `get_action` method. The forward method maps the input state to the action probabilities, and the `get_action` method selects an action from the given action probabilities (for both continuous and discrete action_spaces)

Say you want to parametrize your policy by a Neural Network containing LSTM layers followed my MLP layers. This can be done as follows:

```
# Define a custom LSTM policy from the BasePolicy class
class custom_policy(BasePolicy):
    def __init__(self, state_dim, action_dim, hidden,
                 discrete=True, layer_size=512, layers=1, **kwargs):
        super(custom_policy, self).__init__(state_dim,
                                            action_dim,
                                            hidden,
                                            discrete,
                                            **kwargs)
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.layer_size = layer_size
        self.lstm = nn.LSTM(self.state_dim, layer_size, layers)
        self.fc = mlp([layer_size] + list(hidden) + [action_dim],
                      sac=self.sac)  # the mlp layers

    def forward(self, state):
        state, h = self.lstm(state.unsqueeze(0))
        state = state.view(-1, self.layer_size)
        action = self.fc(state)
        return action
```

Finally, it's time to train the custom policy. Define the environment to be trained on (`CartPole-v0` in this case), and the `state_dim` and `action_dim` variables.

```
# Initialize an environment
env = VectorEnv("CartPole-v0", 1)

# Initialize the custom Policy
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
policy = custom_policy(state_dim=state_dim, action_dim=action_dim,
                        hidden = (64, 64))
```

Then the algorithm is initialised with the custom policy defined, and the OnPolicyTrainer trains in with logging for better inference.

```
algo = VPG(policy, env)

# Initialize the trainer and start training
trainer = OnPolicyTrainer(algo, env, log_mode=["csv"],
                          logdir="./logs", epochs=100)
trainer.train()
```

2.3.5 Using A2C

Using A2C on "CartPole-v0"

```
import gym
```

(continues on next page)

(continued from previous page)

```
from genrl.agents import A2C
from genrl.trainers import OnPolicyTrainer
from genrl.environments import VectorEnv

env = VectorEnv("CartPole-v0")
agent = A2C('mlp', env, gamma=0.9, lr_policy=0.01, lr_value=0.1, policy_layers=(32,
↪32), value_layers=(32, 32),rollout_size=2048)
trainer = OnPolicyTrainer(agent, env, log_mode=['stdout', 'tensorboard'], log_key=
↪"Episode")
trainer.train()
```

Using A2C on atari env - “Pong-v0”

```


env = VectorEnv("Pong-v0", env_type = "atari")
agent = A2C('cnn', env, gamma=0.99, lr_policy=0.01, lr_value=0.1,  policy_layers=(32,
↪32), value_layers=(32, 32), rollout_size=2048)
trainer = OnPolicyTrainer(agent, env, log_mode=['stdout', 'tensorboard'], log_key=
↪"timestep")
trainer.train()
```

More details can be found in the docs for A2C and OnPolicyTrainer.

2.3.6 Using Shared Parameters in Actor Critic Agents in GenRL

The Actor Critic Agents use two networks, an Actor network to select an action to be taken in the current state, and a critic network, to estimate the value of the state the agent is currently in. There are two common ways to implement this actor critic architecture.

The first method - Indpendent Actor and critic networks -

```
            state
           /     \
<actor network>   <critic network>
        /             \
    action          value
```

And the second method - Using a set of shared parameters to extract a feature vector from the state. The actor and the critic network act on this feature vector to select an action and estimate the value

```
             state
               |
           <decoder>
         /           \
<actor network>     <critic network>
      /                 \
  action               value
```

GenRL provides support to incorporte this decoder network in all of the actor critic agents through a `shared_layers` parameter. `shared_layers` takes the sizes of the mlp layers to be used, and `None` if no decoder network is to be used

As an example - in A2C -

```
# The imports
from genrl.agents import A2C
from genrl.environments import VectorEnv
from genrl.trainers import OnPolicyTrainer

# Initializing the environment
env = VectorEnv("CartPole-v0", 1)

# Initializing the agent to be used
algo = A2C(
        "mlp",
        env,
        policy_layers=(128,),
        value_layers=(128,),
        shared_layers=(32, 64),
        rollout_size=128,
    )

# Finally initializing the trainer and trainer
trainer = OnPolicyTrainer(algo, env, log_mode=["csv"], logdir="./logs", epochs=1)
trainer.train()
```

The above example uses and mlp of layer sizes (32, 64) as the decoder, and can be visualised as follows -

```
     state
       |
      <32>
       |
      <64>
     /    \
  <128>   <128>
   /        \
action     value
```

2.3.7 Vanilla Policy Gradient (VPG)

If you wanted to explore Policy Gradient algorithms in RL, there is a high chance you would've heard of PPO, DDPG, etc. but understanding them can be tricky if you're just starting.

VPG is arguably one of the easiest to understand policy gradient algorithms while still performing to a good enough level.

Let's understand policy gradient at a high level, unlike the classical algorithms like Q-Learning, Monte Carlo where you try to optimise the outputs of the action-value function of the agent which are then used to determine the optimal policy. In policy gradient, as one would like to say we go directly for the kill shot, basically we optimise the thing we want to use at the end, i.e. the Policy.

So that explains the "Policy" part of Policy Gradient, so what about "Gradient", so gradient comes from the fact that we try to optimise the policy by gradient ascent (unlike the popular gradient descent, here we want to increase the values, hence ascent). So that explains the name, but how does it even work.

For that, have a look at the following Psuedo Code (source: OpenAI)

For a more fundamental understanding this spinningup article is a good resource

Now that we have an understanding of how VPG works at a high level let's jump into the code to see it in actionThis is a very minimal way to run a VPG agent on **GenRL**

VPG agent on a Cartpole Environment

```
import gym  # OpenAI Gym

from genrl.agents import VPG
from genrl.trainers import OnPolicyTrainer
from genrl.environments import VectorEnv

env = VectorEnv("CartPole-v1")
agent = VPG('mlp', env)
trainer = OnPolicyTrainer(agent, env, epochs=200)
trainer.train()
```

This will run a VPG agent `agent` which will interact with the `CartPole-v1` gym environment Let's understand the output on running this (your individual values may differ),

```
timestep         Episode          loss             mean_reward
0                0                8.022            19.8835
20480            10               25.969           75.2941
40960            20               29.2478          144.2254
61440            30               25.5711          129.6203
81920            40               19.8718          96.6038
102400           50               19.2585          106.9452
122880           60               17.7781          99.9024
143360           70               23.6839          121.543
163840           80               24.4362          129.2114
184320           90               28.1183          156.3359
204800           100              26.6074          155.1515
225280           110              27.2012          178.8646
245760           120              26.4612          164.498
266240           130              22.8618          148.4058
286720           140              23.465           153.4082
307200           150              21.9764          151.1439
327680           160              22.445           151.1439
348160           170              22.9925          155.7414
368640           180              22.6605          165.1613
389120           190              23.4676          177.316
```

`timestep`: It is basically the units of time the agent has interacted with the environment since the start of trainingEpisode: It is one complete rollout of the agent, to put it simply it is one complete run until the agent ends up winning or losing`loss`: The loss encountered in that episode`mean_reward`: The mean reward accumulated in that episode

Now if you look closely the agent will not converge to the max reward even if you increase the epochs to say 5000, it is because that during training the agent is behaving according to a stochastic policy (Meaning when you try to pick from an action given a state from the policy it doesn't simply take the one with the maximum return, rather it samples an action from a probability distribution, so in other words, the policy isn't just like a lookup table, it's function which outputs a probability distribution over the actions which we sample from when using it to pick our optimal action).So even if the agent has figured out the optimal policy it is not taking the most optimal action at every step there is an inherent stochasticity to it.If we want the agent to make full use of the learnt policy we can add the following line of code at after the training

```
trainer.evaluate(render=True)
```

This will not only make the agent follow a deterministic policy and thus help you achieve the maximun reward possible reward attainable from the learnt policy but also allow you to see your agent perform by passing `render=True`

For more information on the VPG implementation and the various hyperparameters available have a look at the official **GenRL** docs here

Some more implementations

VPG agent on an Atari Environment

```
env = VectorEnv("Pong-v0", env_type = "atari")
agent = VPG('cnn', env)
trainer = OnPolicyTrainer(agent, env, epochs=200)
trainer.train()
```

2.3.8 Saving and Loading Weights and Hyperparameters with GenRL

We often want to checkpoint our training model in the RL setting, GenRL offers to save your hyperparameters and weights using TOML and pytorch state_dict respectively.

Following is a sample code to save checkpoints -

```
import gym
import shutil

from genrl.agents import VPG
from genrl.environments.suite import VectorEnv
from genrl.core import NormalActionNoise
from genrl.trainers import OnPolicyTrainer

env = VectorEnv("CartPole-v0", 2)
algo = VPG("mlp", env, batch_size=5, replay_size=100)

trainer = OnPolicyTrainer(
    algo,
    env,
    log_mode=["stdout"],
    logdir="./logs",
    save_interval=100,
    epochs=100,
    evaluate_episodes=2,
)
trainer.train()
trainer.evaluate()
shutil.rmtree("./logs")
```

Let's say you have a saved weights and hyperparameters file to load onto the model you can change your trainer as below to load it -

```
trainer = OnPolicyTrainer(
    algo,
    env,
    log_mode=["stdout"],
```

(continues on next page)

(continued from previous page)

```
    logdir="./logs",
    save_interval=100,
    epochs=100,
    evaluate_episodes=2,
    load_weights="./checkpoints/VPG_CartPole-v0/1-log-0.pt",
    load_hyperparams="./checkpoints/VPG_CartPole-v0/1-log-0.toml",
)
```

2.4 Agents

2.4.1 A2C

genrl.agents.deep.a2c.a2c module

class `genrl.agents.deep.a2c.a2c.`**`A2C`**(**args*, *noise: Any = None*, *noise_std: float = 0.1*, *value_coeff: float = 0.5*, *entropy_coeff: float = 0.01*, ***kwargs*)

Bases: `genrl.agents.deep.base.onpolicy.OnPolicyAgent`

Advantage Actor Critic algorithm (A2C)

The synchronous version of A3C Paper: https://arxiv.org/abs/1602.01783

`network`
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

`env`
The environment that the agent is supposed to act on

Type Environment

`create_model`
Whether the model of the algo should be created when initialised

Type bool

`batch_size`
Mini batch size for loading experiences

Type int

`gamma`
The discount factor for rewards

Type float

`layers`
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`shared_layers`
Sizes of shared layers in Actor Critic if using

Type `tuple` of `int`

`lr_policy`
Learning rate for the policy/actor

Type float

`lr_value`
Learning rate for the critic

Type float

`rollout_size`
Capacity of the Replay Buffer

Type int

`buffer_type`
Choose the type of Buffer: ["rollout"]

Type str

`noise`
Action Noise function added to aid in exploration

Type `ActionNoise`

`noise_std`
Standard deviation of the action noise distribution

Type float

`value_coeff`
Ratio of magnitude of value updates to policy updates

Type float

`entropy_coeff`
Ratio of magnitude of entropy updates to policy updates

Type float

`seed`
Seed for randomness

Type int

`render`
Should the env be rendered during training?

Type bool

`device`
Hardware being used for training. Options: ["cuda" -> GPU, "cpu" -> CPU]

Type str

`empty_logs`()
Empties logs

`evaluate_actions`(*states: torch.Tensor*, *actions: torch.Tensor*)
Evaluates actions taken by actor

Actions taken by actor and their respective states are analysed to get log probabilities and values from critics

Parameters

- **`states`** (`torch.Tensor`) – States encountered in rollout

- **actions** (`torch.Tensor`) – Actions taken in response to respective states

Returns Values of states encountered during the rollout log_probs (`torch.Tensor`): Log of action probabilities given a state

Return type values (`torch.Tensor`)

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

get_traj_loss(*values: torch.Tensor*, *dones: torch.Tensor*) → None
Get loss from trajectory traversed by agent during rollouts

Computes the returns and advantages needed for calculating loss

Parameters

- **values** (`torch.Tensor`) – Values of states encountered during the rollout
- **dones** (`list` of bool) – Game over statuses of each environment

select_action(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Select action given state

Action Selection for On Policy Agents with Actor Critic

Parameters

- **state** (`torch.Tensor`) – Current state of the environment
- **deterministic** (*bool*) – Should the policy be deterministic or stochastic

Returns Action taken by the agent value (`torch.Tensor`): Value of given state log_prob (`torch.Tensor`): Log probability of selected action

Return type action (`torch.Tensor`)

update_params() → None
Updates the the A2C network

Function to update the A2C actor-critic architecture

2.4.2 DDPG

genrl.agents.deep.ddpg.ddpg module

class genrl.agents.deep.ddpg.ddpg.**DDPG**(**args*, *noise: genrl.core.noise.ActionNoise = None*, *noise_std: float = 0.2*, ***kwargs*)
Bases: `genrl.agents.deep.base.offpolicy.OffPolicyAgentAC`

Deep Deterministic Policy Gradient Algorithm

Paper: https://arxiv.org/abs/1509.02971

`network`
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

`env`
The environment that the agent is supposed to act on

Type Environment

`create_model`
Whether the model of the algo should be created when initialised

Type bool

`batch_size`
Mini batch size for loading experiences

Type int

`gamma`
The discount factor for rewards

Type float

`layers`
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`shared_layers`
Sizes of shared layers in Actor Critic if using

Type `tuple` of `int`

`lr_policy`
Learning rate for the policy/actor

Type float

`lr_value`
Learning rate for the critic

Type float

`replay_size`
Capacity of the Replay Buffer

Type int

`buffer_type`
Choose the type of Buffer: [“push”, “prioritized”]

Type str

`polyak`
Target model update parameter (1 for hard update)

Type float

`noise`
Action Noise function added to aid in exploration

Type `ActionNoise`

`noise_std`
Standard deviation of the action noise distribution

Type float

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

empty_logs()
Empties logs

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural Network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

update_params(*update_interval: int*) → None
Update parameters of the model

Parameters **update_interval** (*int*) – Interval between successive updates of the target model

2.4.3 DQN

genrl.agents.deep.dqn.base module

class `genrl.agents.deep.dqn.base.`**DQN**(**args*, *max_epsilon: float = 1.0*, *min_epsilon: float = 0.01*, *epsilon_decay: int = 500*, ***kwargs*)
Bases: `genrl.agents.deep.base.offpolicy.OffPolicyAgent`

Base DQN Class

Paper: https://arxiv.org/abs/1312.5602

network
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

env
The environment that the agent is supposed to act on

Type Environment

create_model
Whether the model of the algo should be created when initialised

Type bool

`batch_size`
Mini batch size for loading experiences

Type int

`gamma`
The discount factor for rewards

Type float

`value_layers`
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`lr_value`
Learning rate for the Q-value function

Type float

`replay_size`
Capacity of the Replay Buffer

Type int

`buffer_type`
Choose the type of Buffer: [“push”, “prioritized”]

Type str

`max_epsilon`
Maximum epsilon for exploration

Type str

`min_epsilon`
Minimum epsilon for exploration

Type str

`epsilon_decay`
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

`seed`
Seed for randomness

Type int

`render`
Should the env be rendered during training?

Type bool

`device`
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

`calculate_epsilon_by_frame``()` → float
Helper function to calculate epsilon after every timestep

Exponentially decays exploration rate from max epsilon to min epsilon The greater the value of epsilon_decay, the slower the decrease in epsilon

empty_logs() → None
Empties logs

get_greedy_action(*state: torch.Tensor*) → torch.Tensor
Greedy action selection

Parameters **state** (`torch.Tensor`) – Current state of the environment

Returns Action taken by the agent

Return type action (`torch.Tensor`)

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

get_q_values(*states: torch.Tensor*, *actions: torch.Tensor*) → torch.Tensor
Get Q values corresponding to specific states and actions

Parameters

- **states** (`torch.Tensor`) – States for which Q-values need to be found
- **actions** (`torch.Tensor`) – Actions taken at respective states

Returns Q values for the given states and actions

Return type q_values (`torch.Tensor`)

get_target_q_values(*next_states: torch.Tensor, rewards: List[float], dones: List[bool]*) → torch.Tensor
Get target Q values for the DQN

Parameters

- **next_states** (`torch.Tensor`) – Next states for which target Q-values need to be found
- **rewards** (`list`) – Rewards at each timestep for each environment
- **dones** (`list`) – Game over status for each environment

Returns Target Q values for the DQN

Return type target_q_values (`torch.Tensor`)

load_weights(*weights*) → None
Load weights for the agent from pretrained model

Parameters **weights** (`torch.Tensor`) – neural net weights

select_action(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Select action given state

Epsilon-greedy action-selection

Parameters

- **state** (`torch.Tensor`) – Current state of the environment

- **`deterministic`** (*`bool`*) – Should the policy be deterministic or stochastic

Returns Action taken by the agent

Return type action (`torch.Tensor`)

`update_params` (*update_interval: int*) → None

Update parameters of the model

Parameters **`update_interval`** (*`int`*) – Interval between successive updates of the target model

`update_params_before_select_action` (*timestep: int*) → None

Update necessary parameters before selecting an action

This updates the epsilon (exploration rate) of the agent every timestep

Parameters **`timestep`** (*`int`*) – Timestep of training

`update_target_model`() → None

Function to update the target Q model

Updates the target model with the training model's weights when called

genrl.agents.deep.dqn.categorical module

class `genrl.agents.deep.dqn.categorical.`**`CategoricalDQN`**(**args, noisy_layers: Tuple = (32, 128), num_atoms: int = 51, v_min: int = -10, v_max: int = 10, **kwargs*)

Bases: *`genrl.agents.deep.dqn.base.DQN`*

Categorical DQN Algorithm

Paper: https://arxiv.org/pdf/1707.06887.pdf

`network`

The network type of the Q-value function. Supported types: ["cnn", "mlp"]

Type str

`env`

The environment that the agent is supposed to act on

Type Environment

`create_model`

Whether the model of the algo should be created when initialised

Type bool

`batch_size`

Mini batch size for loading experiences

Type int

`gamma`

The discount factor for rewards

Type float

`layers`

Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`lr_value`
Learning rate for the Q-value function

Type float

`replay_size`
Capacity of the Replay Buffer

Type int

`buffer_type`
Choose the type of Buffer: [“push”, “prioritized”]

Type str

`max_epsilon`
Maximum epsilon for exploration

Type str

`min_epsilon`
Minimum epsilon for exploration

Type str

`epsilon_decay`
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

`noisy_layers`
Noisy layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`num_atoms`
Number of atoms used in the discrete distribution

Type int

`v_min`
Lower bound of value distribution

Type int

`v_max`
Upper bound of value distribution

Type int

`seed`
Seed for randomness

Type int

`render`
Should the env be rendered during training?

Type bool

`device`
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

`get_greedy_action`(*state: torch.Tensor*) → torch.Tensor
Greedy action selection

Parameters **`state`** (`torch.Tensor`) – Current state of the environment

Returns Action taken by the agent

Return type action (`torch.Tensor`)

`get_q_loss` (*batch: collections.namedtuple*)
Categorical DQN loss function to calculate the loss of the Q-function

Parameters **`batch`** (`collections.namedtuple` of `torch.Tensor`) – Batch of experiences

Returns Calculateed loss of the Q-function

Return type loss (`torch.Tensor`)

`get_q_values` (*states: torch.Tensor*, *actions: torch.Tensor*)
Get Q values corresponding to specific states and actions

Parameters

- **`states`** (`torch.Tensor`) – States for which Q-values need to be found
- **`actions`** (`torch.Tensor`) – Actions taken at respective states

Returns Q values for the given states and actions

Return type q_values (`torch.Tensor`)

`get_target_q_values` (*next_states: torch.Tensor*, *rewards: torch.Tensor*, *dones: torch.Tensor*)
Projected Distribution of Q-values

Helper function for Categorical/Distributional DQN

Parameters

- **`next_states`** (`torch.Tensor`) – Next states being encountered by the agent
- **`rewards`** (`torch.Tensor`) – Rewards received by the agent
- **`dones`** (`torch.Tensor`) – Game over status of each environment

Returns Projected Q-value Distribution or Target Q Values

Return type target_q_values (object)

genrl.agents.deep.dqn.double module

`class` `genrl.agents.deep.dqn.double.`**`DoubleDQN`** (**args*, ***kwargs*)
Bases: *`genrl.agents.deep.dqn.base.DQN`*

Double DQN Class

Paper: https://arxiv.org/abs/1509.06461

`network`
The network type of the Q-value function. Supported types: ["cnn", "mlp"]

Type str

`env`
The environment that the agent is supposed to act on

Type Environment

`batch_size`
Mini batch size for loading experiences

Type int

gamma
The discount factor for rewards

Type float

layers
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

lr_value
Learning rate for the Q-value function

Type float

replay_size
Capacity of the Replay Buffer

Type int

buffer_type
Choose the type of Buffer: [“push”, “prioritized”]

Type str

max_epsilon
Maximum epsilon for exploration

Type str

min_epsilon
Minimum epsilon for exploration

Type str

epsilon_decay
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

get_target_q_values(*next_states: torch.Tensor*, *rewards: torch.Tensor*, *dones: torch.Tensor*) → torch.Tensor
Get target Q values for the DQN

Parameters

- **next_states** (`torch.Tensor`) – Next states for which target Q-values need to be found
- **rewards** (`list`) – Rewards at each timestep for each environment

- **dones** (`list`) – Game over status for each environment

Returns Target Q values for the DQN

Return type target_q_values (`torch.Tensor`)

genrl.agents.deep.dqn.dueling module

class `genrl.agents.deep.dqn.dueling.`**`DuelingDQN`**(**args*, ***kwargs*)

Bases: *`genrl.agents.deep.dqn.base.DQN`*

Dueling DQN class

Paper: https://arxiv.org/abs/1511.06581

`network`

The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

`env`

The environment that the agent is supposed to act on

Type Environment

`batch_size`

Mini batch size for loading experiences

Type int

`gamma`

The discount factor for rewards

Type float

`layers`

Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`lr_value`

Learning rate for the Q-value function

Type float

`replay_size`

Capacity of the Replay Buffer

Type int

`buffer_type`

Choose the type of Buffer: [“push”, “prioritized”]

Type str

`max_epsilon`

Maximum epsilon for exploration

Type str

`min_epsilon`

Minimum epsilon for exploration

Type str

epsilon_decay
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

genrl.agents.deep.dqn.noisy module

class `genrl.agents.deep.dqn.noisy.`**NoisyDQN**(**args*, *noisy_layers: Tuple = (128, 128)*, ***kwargs*)

Bases: `genrl.agents.deep.dqn.base.DQN`

Noisy DQN Algorithm

Paper: https://arxiv.org/abs/1706.10295

network
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

env
The environment that the agent is supposed to act on

Type Environment

batch_size
Mini batch size for loading experiences

Type int

gamma
The discount factor for rewards

Type float

layers
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

lr_value
Learning rate for the Q-value function

Type float

replay_size
Capacity of the Replay Buffer

Type int

buffer_type
Choose the type of Buffer: [“push”, “prioritized”]

Type str

max_epsilon
Maximum epsilon for exploration

Type str

min_epsilon
Minimum epsilon for exploration

Type str

epsilon_decay
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

noisy_layers
Noisy layers in the Neural Network of the Q-value function

Type `tuple` of `int`

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

genrl.agents.deep.dqn.prioritized module

class `genrl.agents.deep.dqn.prioritized.`**`PrioritizedReplayDQN`**(**args, alpha: float = 0.6, beta: float = 0.4, **kwargs*)

Bases: *`genrl.agents.deep.dqn.base.DQN`*

Prioritized Replay DQN Class

Paper: https://arxiv.org/abs/1511.05952

network
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

env
The environment that the agent is supposed to act on

Type Environment

batch_size
Mini batch size for loading experiences

Type int

gamma
The discount factor for rewards

Type float

layers
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

lr_value
Learning rate for the Q-value function

Type float

replay_size
Capacity of the Replay Buffer

Type int

buffer_type
Choose the type of Buffer: [“push”, “prioritized”]

Type str

max_epsilon
Maximum epsilon for exploration

Type str

min_epsilon
Minimum epsilon for exploration

Type str

epsilon_decay
Rate of decay of epsilon (in order to decrease exploration with time)

Type str

alpha
Prioritization constant

Type float

beta
Importance Sampling bias

Type float

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

get_q_loss(*batch: collections.namedtuple*) → torch.Tensor
Normal Function to calculate the loss of the Q-function

Parameters **batch** (`collections.namedtuple` of `torch.Tensor`) – Batch of experiences

Returns Calculateed loss of the Q-function

Return type loss (`torch.Tensor`)

genrl.agents.deep.dqn.utils module

`genrl.agents.deep.dqn.utils.`**`categorical_greedy_action`**(*agent: genrl.agents.deep.dqn.base.DQN*, *state: torch.Tensor*) → torch.Tensor

Greedy action selection for Categorical DQN

Parameters

- **agent** (`DQN`) – The agent
- **state** (`torch.Tensor`) – Current state of the environment

Returns Action taken by the agent

Return type action (`torch.Tensor`)

`genrl.agents.deep.dqn.utils.`**`categorical_q_loss`**(*agent: genrl.agents.deep.dqn.base.DQN*, *batch: collections.namedtuple*)

Categorical DQN loss function to calculate the loss of the Q-function

Parameters

- **agent** (`DQN`) – The agent
- **batch** (`collections.namedtuple` of `torch.Tensor`) – Batch of experiences

Returns Calculateed loss of the Q-function

Return type loss (`torch.Tensor`)

`genrl.agents.deep.dqn.utils.`**`categorical_q_target`**(*agent: genrl.agents.deep.dqn.base.DQN*, *next_states: torch.Tensor*, *rewards: torch.Tensor*, *dones: torch.Tensor*)

Projected Distribution of Q-values

Helper function for Categorical/Distributional DQN

Parameters

- **agent** (`DQN`) – The agent
- **next_states** (`torch.Tensor`) – Next states being encountered by the agent
- **rewards** (`torch.Tensor`) – Rewards received by the agent
- **dones** (`torch.Tensor`) – Game over status of each environment

Returns Projected Q-value Distribution or Target Q Values

Return type target_q_values (object)

`genrl.agents.deep.dqn.utils.`**`categorical_q_values`**(*agent: genrl.agents.deep.dqn.base.DQN*, *states: torch.Tensor*, *actions: torch.Tensor*)

Get Q values given state for a Categorical DQN

Parameters

- **agent** (DQN) – The agent
- **states** (torch.Tensor) – States being replayed
- **actions** (torch.Tensor) – Actions being replayed

Returns Q values for the given states and actions

Return type q_values (torch.Tensor)

genrl.agents.deep.dqn.utils.**ddqn_q_target**(*agent: genrl.agents.deep.dqn.base.DQN*, *next_states: torch.Tensor*, *rewards: torch.Tensor*, *dones: torch.Tensor*) → torch.Tensor

Double Q-learning target

Can be used to replace the *get_target_values* method of the Base DQN class in any DQN algorithm

Parameters

- **agent** (DQN) – The agent
- **next_states** (torch.Tensor) – Next states being encountered by the agent
- **rewards** (torch.Tensor) – Rewards received by the agent
- **dones** (torch.Tensor) – Game over status of each environment

Returns Target Q values using Double Q-learning

Return type target_q_values (torch.Tensor)

genrl.agents.deep.dqn.utils.**prioritized_q_loss**(*agent: genrl.agents.deep.dqn.base.DQN*, *batch: collections.namedtuple*)

Function to calculate the loss of the Q-function

Returns The agent loss (torch.Tensor): Calculateed loss of the Q-function

Return type agent (DQN)

2.4.4 PPO1

genrl.agents.deep.ppo1.ppo1 module

class genrl.agents.deep.ppo1.ppo1.**PPO1**(**args*, *clip_param: float = 0.2*, *value_coeff: float = 0.5*, *entropy_coeff: float = 0.01*, ***kwargs*)

Bases: genrl.agents.deep.base.onpolicy.OnPolicyAgent

Proximal Policy Optimization algorithm (Clipped policy).

Paper: https://arxiv.org/abs/1707.06347

network

The network type of the Q-value function. Supported types: ["cnn", "mlp"]

Type str

env

The environment that the agent is supposed to act on

Type Environment

create_model

Whether the model of the algo should be created when initialised

Type bool

`batch_size`
Mini batch size for loading experiences

Type int

`gamma`
The discount factor for rewards

Type float

`layers`
Layers in the Neural Network of the Q-value function

Type `tuple` of `int`

`shared_layers`
Sizes of shared layers in Actor Critic if using

Type `tuple` of `int`

`lr_policy`
Learning rate for the policy/actor

Type float

`lr_value`
Learning rate for the Q-value function

Type float

`rollout_size`
Capacity of the Rollout Buffer

Type int

`buffer_type`
Choose the type of Buffer: [“rollout”]

Type str

`clip_param`
Epsilon for clipping policy loss

Type float

`value_coeff`
Ratio of magnitude of value updates to policy updates

Type float

`entropy_coeff`
Ratio of magnitude of entropy updates to policy updates

Type float

`seed`
Seed for randomness

Type int

`render`
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: ["cuda" -> GPU, "cpu" -> CPU]

Type str

empty_logs()
Empties logs

evaluate_actions(*states: torch.Tensor*, *actions: torch.Tensor*)
Evaluates actions taken by actor

Actions taken by actor and their respective states are analysed to get log probabilities and values from critics

Parameters

- **states** (`torch.Tensor`) – States encountered in rollout
- **actions** (`torch.Tensor`) – Actions taken in response to respective states

Returns Values of states encountered during the rollout log_probs (`torch.Tensor`): Log of action probabilities given a state

Return type values (`torch.Tensor`)

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

get_traj_loss(*values*, *dones*)
Get loss from trajectory traversed by agent during rollouts

Computes the returns and advantages needed for calculating loss

Parameters

- **values** (`torch.Tensor`) – Values of states encountered during the rollout
- **dones** (`list` of bool) – Game over statuses of each environment

select_action(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Select action given state

Action Selection for On Policy Agents with Actor Critic

Parameters

- **state** (`np.ndarray`) – Current state of the environment
- **deterministic** (*`bool`*) – Should the policy be deterministic or stochastic

Returns Action taken by the agent value (`torch.Tensor`): Value of given state log_prob (`torch.Tensor`): Log probability of selected action

Return type action (`np.ndarray`)

update_params()
Updates the the A2C network

Function to update the A2C actor-critic architecture

2.4.5 VPG

genrl.agents.deep.vpg.vpg module

class genrl.agents.deep.vpg.vpg.**VPG**(**args*, ***kwargs*)
Bases: `genrl.agents.deep.base.onpolicy.OnPolicyAgent`

Vanilla Policy Gradient algorithm

Paper https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf

network (str): The network type of the Q-value function. Supported types: ["cnn", "mlp"]

env (Environment): The environment that the agent is supposed to act on create_model (bool): Whether the model of the algo should be created when initialised batch_size (int): Mini batch size for loading experiences gamma (float): The discount factor for rewards layers (`tuple` of `int`): Layers in the Neural Network

of the Q-value function

lr_policy (float): Learning rate for the policy/actor lr_value (float): Learning rate for the Q-value function rollout_size (int): Capacity of the Rollout Buffer buffer_type (str): Choose the type of Buffer: ["rollout"] seed (int): Seed for randomness render (bool): Should the env be rendered during training? device (str): Hardware being used for training. Options:

["cuda" -> GPU, "cpu" -> CPU]

empty_logs()
Empties logs

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_log_probs(*states: torch.Tensor*, *actions: torch.Tensor*)
Get log probabilities of action values

Actions taken by actor and their respective states are analysed to get log probabilities

Parameters

- **states** (`torch.Tensor`) – States encountered in rollout
- **actions** (`torch.Tensor`) – Actions taken in response to respective states

Returns Log of action probabilities given a state

Return type log_probs (`torch.Tensor`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

`get_traj_loss`(*values*, *dones*)
Get loss from trajectory traversed by agent during rollouts

Computes the returns and advantages needed for calculating loss

Parameters

- **`values`** (`torch.Tensor`) – Values of states encountered during the rollout
- **`dones`** (`list` of bool) – Game over statuses of each environment

`select_action`(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Select action given state

Action Selection for Vanilla Policy Gradient

Parameters

- **`state`** (`np.ndarray`) – Current state of the environment
- **`deterministic`** (*`bool`*) – Should the policy be deterministic or stochastic

Returns

Action taken by the agent value (`torch.Tensor`): Value of given state. In VPG, there is no critic

to find the value so we set this to a default 0 for convenience

log_prob (`torch.Tensor`): Log probability of selected action

Return type action (`np.ndarray`)

`update_params`() → None
Updates the the A2C network

Function to update the A2C actor-critic architecture

2.4.6 TD3

genrl.agents.deep.td3.td3 module

class `genrl.agents.deep.td3.td3.`**`TD3`**(**args*, *policy_frequency: int = 2*, *noise: genrl.core.noise.ActionNoise = None*, *noise_std: float = 0.2*, ***kwargs*)
Bases: `genrl.agents.deep.base.offpolicy.OffPolicyAgentAC`

Twin Delayed DDPG Algorithm

Paper: https://arxiv.org/abs/1509.02971

`network`
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

`env`
The environment that the agent is supposed to act on

Type Environment

`create_model`
Whether the model of the algo should be created when initialised

Type bool

batch_size
Mini batch size for loading experiences

Type int

gamma
The discount factor for rewards

Type float

policy_layers
Neural network layer dimensions for the policy

Type `tuple` of `int`

value_layers
Neural network layer dimensions for the critics

Type `tuple` of `int`

shared_layers
Sizes of shared layers in Actor Critic if using

Type `tuple` of `int`

lr_policy
Learning rate for the policy/actor

Type float

lr_value
Learning rate for the critic

Type float

replay_size
Capacity of the Replay Buffer

Type int

buffer_type
Choose the type of Buffer: [“push”, “prioritized”]

Type str

polyak
Target model update parameter (1 for hard update)

Type float

policy_frequency
Frequency of policy updates in comparison to critic updates

Type int

noise
Action Noise function added to aid in exploration

Type `ActionNoise`

noise_std
Standard deviation of the action noise distribution

Type float

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

empty_logs()
Empties logs

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

update_params(*update_interval: int*) → None
Update parameters of the model

Parameters **update_interval** (*int*) – Interval between successive updates of the target model

2.4.7 SAC

genrl.agents.deep.sac.sac module

class genrl.agents.deep.sac.sac.**SAC**(**args*, *alpha: float = 0.01*, *polyak: float = 0.995*, *entropy_tuning: bool = True*, ***kwargs*)
Bases: `genrl.agents.deep.base.offpolicy.OffPolicyAgentAC`

Soft Actor Critic algorithm (SAC)

Paper: https://arxiv.org/abs/1812.05905

network
The network type of the Q-value function. Supported types: [“cnn”, “mlp”]

Type str

env
The environment that the agent is supposed to act on

Type Environment

create_model
Whether the model of the algo should be created when initialised

Type bool

batch_size
Mini batch size for loading experiences

Type int

gamma
The discount factor for rewards

Type float

policy_layers
Neural network layer dimensions for the policy

Type `tuple` of `int`

value_layers
Neural network layer dimensions for the critics

Type `tuple` of `int`

shared_layers
Sizes of shared layers in Actor Critic if using

Type `tuple` of `int`

lr_policy
Learning rate for the policy/actor

Type float

lr_value
Learning rate for the critic

Type float

replay_size
Capacity of the Replay Buffer

Type int

buffer_type
Choose the type of Buffer: [“push”, “prioritized”]

Type str

alpha
Entropy factor

Type str

polyak
Target model update parameter (1 for hard update)

Type float

entropy_tuning
True if entropy tuning should be done, False otherwise

Type bool

seed
Seed for randomness

Type int

render
Should the env be rendered during training?

Type bool

device
Hardware being used for training. Options: [“cuda” -> GPU, “cpu” -> CPU]

Type str

empty_logs()
Empties logs

get_alpha_loss(*log_probs*)
Calculate Entropy Loss

Parameters **log_probs** (*float*) – Log probs

get_hyperparams() → Dict[str, Any]
Get relevant hyperparameters to save

Returns Hyperparameters to be saved weights (`torch.Tensor`): Neural network weights

Return type hyperparams (`dict`)

get_logging_params() → Dict[str, Any]
Gets relevant parameters for logging

Returns Logging parameters for monitoring training

Return type logs (`dict`)

get_p_loss(*states: torch.Tensor*) → torch.Tensor
Function to get the Policy loss

Parameters **states** (`torch.Tensor`) – States for which Q-values need to be found

Returns Calculated policy loss

Return type loss (`torch.Tensor`)

get_target_q_values(*next_states: torch.Tensor, rewards: List[float], dones: List[bool]*) → torch.Tensor
Get target Q values for the SAC

Parameters

- **next_states** (`torch.Tensor`) – Next states for which target Q-values need to be found
- **rewards** (`list`) – Rewards at each timestep for each environment
- **dones** (`list`) – Game over status for each environment

Returns Target Q values for the SAC

Return type target_q_values (`torch.Tensor`)

select_action(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor
Select action given state

Action Selection

Parameters

- **state** (`np.ndarray`) – Current state of the environment
- **deterministic** (*bool*) – Should the policy be deterministic or stochastic

Returns Action taken by the agent

Return type action (`np.ndarray`)

update_params (*update_interval: int*) → None
Update parameters of the model

Parameters **update_interval** (*int*) – Interval between successive updates of the target model

update_target_model () → None
Function to update the target Q model

Updates the target model with the training model's weights when called

2.4.8 Q-Learning

genrl.agents.classical.qlearning.qlearning module

class genrl.agents.classical.qlearning.qlearning.**QLearning** (*env: gym.core.Env, epsilon: float = 0.9, gamma: float = 0.95, lr: float = 0.01*)

Bases: `object`

Q-Learning Algorithm.

Paper- https://link.springer.com/article/10.1007/BF00992698

env
Environment with which agent interacts.

Type gym.Env

epsilon
exploration coefficient for epsilon-greedy exploration.

Type float, optional

gamma
discount factor.

Type float, optional

lr
learning rate for optimizer.

Type float, optional

get_action (*state: numpy.ndarray, explore: bool = True*) → numpy.ndarray
Epsilon greedy selection of epsilon in the explore phase.

Parameters

- **state** (*np.ndarray*) – Environment state.
- **explore** (*bool, optional*) – True if exploration is required. False if not.

Returns action.

Return type np.ndarray

get_hyperparams () → Dict[str, Any]

update (*transition: Tuple*) → None
Update the Q table.

Parameters **`transition`** (*`Tuple`*) – transition 4-tuple used to update Q-table. In the form (state, action, reward, next_state)

2.4.9 SARSA

genrl.agents.classical.sarsa.sarsa module

class `genrl.agents.classical.sarsa.sarsa.`**`SARSA`**(*env: gym.core.Env*, *epsilon: float = 0.9*, *lmbda: float = 0.9*, *gamma: float = 0.95*, *lr: float = 0.01*)

Bases: `object`

SARSA Algorithm.

Paper- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2539&rep=rep1&type=pdf

`env`

Environment with which agent interacts.

Type gym.Env

`epsilon`

exploration coefficient for epsilon-greedy exploration.

Type float, optional

`gamma`

discount factor.

Type float, optional

`lr`

learning rate for optimizer.

Type float, optional

`get_action`(*state: numpy.ndarray*, *explore: bool = True*) → numpy.ndarray

Epsilon greedy selection of epsilon in the explore phase.

Parameters

- **`state`** (*`np.ndarray`*) – Environment state.
- **`explore`** (*`bool, optional`*) – True if exploration is required. False if not.

Returns action.

Return type np.ndarray

`update`(*transition: Tuple*) → None

Update the Q table and e values

Parameters **`transition`** (*`Tuple`*) – transition 4-tuple used to update Q-table. In the form (state, action, reward, next_state)

2.4.10 Contextual Bandit

Base

class `genrl.agents.bandits.contextual.base.`**`DCBAgent`**(*bandit: genrl.core.bandit.Bandit*, *device: str = 'cpu'*, ***kwargs*)

Bases: `genrl.core.bandit.BanditAgent`

Base class for deep contextual bandit solving agents

Parameters

- **`bandit`** (*`gennav.deep.bandit.data_bandits.DataBasedBandit`*) – The bandit to solve
- **`device`** (*`str`*) – Device to use for tensor operations. "cpu" for cpu or "cuda" for cuda. Defaults to "cpu".

`bandit`

The bandit to solve

Type gennav.deep.bandit.data_bandits.DataBasedBandit

`device`

Device to use for tensor operations.

Type torch.device

`select_action`(*context: torch.Tensor*) → int

Select an action based on given context

Parameters **`context`** (*`torch.Tensor`*) – The context vector to select action for

Note: This method needs to be implemented in the specific agent.

Returns The action to take

Return type int

`update_parameters`(*action: Optional[int] = None*, *batch_size: Optional[int] = None*, *train_epochs: Optional[int] = None*) → None

Update parameters of the agent.

Parameters

- **`action`** (*`Optional[int], optional`*) – Action to update the parameters for. Defaults to None.
- **`batch_size`** (*`Optional[int], optional`*) – Size of batch to update parameters with. Defaults to None.
- **`train_epochs`** (*`Optional[int], optional`*) – Epochs to train neural network for. Defaults to None.

Note: This method needs to be implemented in the specific agent.

Bootstrap Neural

class genrl.agents.bandits.contextual.bootstrap_neural.**BootstrapNeuralAgent**(*bandit: genrl.utils.data_band* ***kwargs*)

Bases: *genrl.agents.bandits.contextual.base.DCBAgent*

Bootstraped ensemble agentfor deep contextual bandits.

Parameters

- **bandit** (*DataBasedBandit*) – The bandit to solve
- **init_pulls** (*int, optional*) – Number of times to select each action initially. Defaults to 3.
- **hidden_dims** (*List[int], optional*) – Dimensions of hidden layers of network. Defaults to [50, 50].
- **init_lr** (*float, optional*) – Initial learning rate. Defaults to 0.1.
- **lr_decay** (*float, optional*) – Decay rate for learning rate. Defaults to 0.5.
- **lr_reset** (*bool, optional*) – Whether to reset learning rate ever train interval. Defaults to True.
- **max_grad_norm** (*float, optional*) – Maximum norm of gradients for gradient clipping. Defaults to 0.5.
- **dropout_p** (*Optional[float], optional*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **eval_with_dropout** (*bool, optional*) – Whether or not to use dropout at inference. Defaults to False.
- **n** (*int, optional*) – Number of models in ensemble. Defaults to 10.
- **add_prob** (*float, optional*) – Probability of adding a transition to a database. Defaults to 0.95.
- **device** (*str*) – Device to use for tensor operations. “cpu” for cpu or “cuda” for cuda. Defaults to “cpu”.

select_action(*context: torch.Tensor*) → int

Select an action based on given context.

Selects an action by computing a forward pass through a randomly selected network from the ensemble.

Parameters **context** (*torch.Tensor*) – The context vector to select action for.

Returns The action to take.

Return type int

update_db(*context: torch.Tensor*, *action: int*, *reward: int*)

Updates transition database with given transition

The transition is added to each database with a certain probability.

Parameters

- **context** (*torch.Tensor*) – Context recieved
- **action** (*int*) – Action taken
- **reward** (*int*) – Reward recieved

update_params (*action: Optional[int] = None*, *batch_size: int = 512*, *train_epochs: int = 20*)

Update parameters of the agent.

Trains each neural network in the ensemble.

Parameters

- **action** (*Optional[int], optional*) – Action to update the parameters for. Not applicable in this agent. Defaults to None.
- **batch_size** (*int, optional*) – Size of batch to update parameters with. Defaults to 512
- **train_epochs** (*int, optional*) – Epochs to train neural network for. Defaults to 20

Fixed

class genrl.agents.bandits.contextual.fixed.**FixedAgent** (*bandit: genrl.utils.data_bandits.base.DataBasedBandit*, *p: List[float] = None*, *device: str = 'cpu'*)

Bases: *genrl.agents.bandits.contextual.base.DCBAgent*

select_action (*context: torch.Tensor*) → int

Select an action based on fixed probabilities.

Parameters **context** (*torch.Tensor*) – The context vector to select action for. In this agent, context vector is not considered.

Returns The action to take.

Return type int

update_db (**args*, ***kwargs*)

update_params (**args*, ***kwargs*)

Linear Posterior

class genrl.agents.bandits.contextual.linpos.**LinearPosteriorAgent** (*bandit: genrl.utils.data_bandits.base.DataB*
***kwargs*)

Bases: *genrl.agents.bandits.contextual.base.DCBAgent*

Deep contextual bandit agent using bayesian regression for posterior inference.

Parameters

- **bandit** (*DataBasedBandit*) – The bandit to solve
- **init_pulls** (*int, optional*) – Number of times to select each action initially. Defaults to 3.
- **lambda_prior** (*float, optional*) – Guassian prior for linear model. Defaults to 0.25.
- **a0** (*float, optional*) – Inverse gamma prior for noise. Defaults to 6.0.
- **b0** (*float, optional*) – Inverse gamma prior for noise. Defaults to 6.0.

- **device** (*str*) – Device to use for tensor operations. "cpu" for cpu or "cuda" for cuda. Defaults to "cpu".

select_action(*context: torch.Tensor*) → int

Select an action based on given context.

Selecting action with highest predicted reward computed through betas sampled from posterior.

Parameters **context** (*torch.Tensor*) – The context vector to select action for.

Returns The action to take.

Return type int

update_db(*context: torch.Tensor*, *action: int*, *reward: int*)

Updates transition database with given transition

Parameters

- **context** (*torch.Tensor*) – Context recieved
- **action** (*int*) – Action taken
- **reward** (*int*) – Reward recieved

update_params(*action: int*, *batch_size: int = 512*, *train_epochs: Optional[int] = None*)

Update parameters of the agent.

Updated the posterior over beta though bayesian regression.

Parameters

- **action** (*int*) – Action to update the parameters for.
- **batch_size** (*int, optional*) – Size of batch to update parameters with. Defaults to 512
- **train_epochs** (*Optional[int], optional*) – Epochs to train neural network for. Not applicable in this agent. Defaults to None

Neural Greedy

class genrl.agents.bandits.contextual.neural_greedy.**NeuralGreedyAgent**(*bandit: genrl.utils.data_bandits.base.I*
***kwargs*)

Bases: *genrl.agents.bandits.contextual.base.DCBAgent*

Deep contextual bandit agent using epsilon greedy with a neural network.

Parameters

- **bandit** (*DataBasedBandit*) – The bandit to solve
- **init_pulls** (*int, optional*) – Number of times to select each action initially. Defaults to 3.
- **hidden_dims** (*List[int], optional*) – Dimensions of hidden layers of network. Defaults to [50, 50].
- **init_lr** (*float, optional*) – Initial learning rate. Defaults to 0.1.
- **lr_decay** (*float, optional*) – Decay rate for learning rate. Defaults to 0.5.
- **lr_reset** (*bool, optional*) – Whether to reset learning rate ever train interval. Defaults to True.

- **`max_grad_norm`** (*`float, optional`*) – Maximum norm of gradients for gradient clipping. Defaults to 0.5.
- **`dropout_p`** (*`Optional[float], optional`*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **`eval_with_dropout`** (*`bool, optional`*) – Whether or not to use dropout at inference. Defaults to False.
- **`epsilon`** (*`float, optional`*) – Probability of selecting a random action. Defaults to 0.0.
- **`device`** (*`str`*) – Device to use for tensor operations. "cpu" for cpu or "cuda" for cuda. Defaults to "cpu".

`select_action`(*context: torch.Tensor*) → int

Select an action based on given context.

Selects an action by computing a forward pass through network with an epsillon probability of selecting a random action.

Parameters **`context`** (*`torch.Tensor`*) – The context vector to select action for.

Returns The action to take.

Return type int

`update_db`(*context: torch.Tensor*, *action: int*, *reward: int*)

Updates transition database with given transition

Parameters

- **`context`** (*`torch.Tensor`*) – Context recieved
- **`action`** (*`int`*) – Action taken
- **`reward`** (*`int`*) – Reward recieved

`update_params`(*action: Optional[int] = None*, *batch_size: int = 512*, *train_epochs: int = 20*)

Update parameters of the agent.

Trains neural network.

Parameters

- **`action`** (*`Optional[int], optional`*) – Action to update the parameters for. Not applicable in this agent. Defaults to None.
- **`batch_size`** (*`int, optional`*) – Size of batch to update parameters with. Defaults tp 512
- **`train_epochs`** (*`int, optional`*) – Epochs to train neural network for. Defaults to 20

Neural Linear Posterior

class `genrl.agents.bandits.contextual.neural_linpos.`**`NeuralLinearPosteriorAgent`**(*bandit: genrl.utils.data_l*
***kwargs*)

Bases: *`genrl.agents.bandits.contextual.base.DCBAgent`*

Deep contextual bandit agent using bayesian regression on for posterior inference

A neural network is used to transform context vector to a latent represntation on which bayesian regression is performed.

Parameters

- **`bandit`** (*`DataBasedBandit`*) – The bandit to solve
- **`init_pulls`** (*`int, optional`*) – Number of times to select each action initially. Defaults to 3.
- **`hidden_dims`** (*`List[int], optional`*) – Dimensions of hidden layers of network. Defaults to [50, 50].
- **`init_lr`** (*`float, optional`*) – Initial learning rate. Defaults to 0.1.
- **`lr_decay`** (*`float, optional`*) – Decay rate for learning rate. Defaults to 0.5.
- **`lr_reset`** (*`bool, optional`*) – Whether to reset learning rate ever train interval. Defaults to True.
- **`max_grad_norm`** (*`float, optional`*) – Maximum norm of gradients for gradient clipping. Defaults to 0.5.
- **`dropout_p`** (*`Optional[float], optional`*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **`eval_with_dropout`** (*`bool, optional`*) – Whether or not to use dropout at inference. Defaults to False.
- **`nn_update_ratio`** (*`int, optional`*) – . Defaults to 2.
- **`lambda_prior`** (*`float, optional`*) – Guassian prior for linear model. Defaults to 0.25.
- **`a0`** (*`float, optional`*) – Inverse gamma prior for noise. Defaults to 3.0.
- **`b0`** (*`float, optional`*) – Inverse gamma prior for noise. Defaults to 3.0.
- **`device`** (*`str`*) – Device to use for tensor operations. “cpu” for cpu or “cuda” for cuda. Defaults to “cpu”.

`select_action`(*context: torch.Tensor*) → int

Select an action based on given context.

Selects an action by computing a forward pass through network to output a representation of the context on which bayesian linear regression is performed to select an action.

Parameters **`context`** (*`torch.Tensor`*) – The context vector to select action for.

Returns The action to take.

Return type int

`update_db`(*context: torch.Tensor*, *action: int*, *reward: int*)

Updates transition database with given transition

Updates latent context and predicted rewards seperately.

Parameters

- **`context`** (*`torch.Tensor`*) – Context recieved
- **`action`** (*`int`*) – Action taken
- **`reward`** (*`int`*) – Reward recieved

update_params (*action: int*, *batch_size: int = 512*, *train_epochs: int = 20*)

Update parameters of the agent.

Trains neural network and updates bayesian regression parameters.

Parameters

- **action** (*int*) – Action to update the parameters for.
- **batch_size** (*int, optional*) – Size of batch to update parameters with. Defaults to 512
- **train_epochs** (*int, optional*) – Epochs to train neural network for. Defaults to 20

Neural Noise Sampling

class genrl.agents.bandits.contextual.neural_noise_sampling.**NeuralNoiseSamplingAgent** (*bandit: genrl.ut* ***kwarg*

Bases: *genrl.agents.bandits.contextual.base.DCBAgent*

Deep contextual bandit agent with noise sampling for neural network parameters.

Parameters

- **bandit** (*DataBasedBandit*) – The bandit to solve
- **init_pulls** (*int, optional*) – Number of times to select each action initially. Defaults to 3.
- **hidden_dims** (*List[int], optional*) – Dimensions of hidden layers of network. Defaults to [50, 50].
- **init_lr** (*float, optional*) – Initial learning rate. Defaults to 0.1.
- **lr_decay** (*float, optional*) – Decay rate for learning rate. Defaults to 0.5.
- **lr_reset** (*bool, optional*) – Whether to reset learning rate ever train interval. Defaults to True.
- **max_grad_norm** (*float, optional*) – Maximum norm of gradients for gradient clipping. Defaults to 0.5.
- **dropout_p** (*Optional[float], optional*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **eval_with_dropout** (*bool, optional*) – Whether or not to use dropout at inference. Defaults to False.
- **noise_std_dev** (*float, optional*) – Standard deviation of sampled noise. Defaults to 0.05.
- **eps** (*float, optional*) – Small constant for bounding KL divergece of noise. Defaults to 0.1.
- **noise_update_batch_size** (*int, optional*) – Batch size for updating noise parameters. Defaults to 256.
- **device** (*str*) – Device to use for tensor operations. “cpu” for cpu or “cuda” for cuda. Defaults to “cpu”.

`select_action`(*context: torch.Tensor*) → int
Select an action based on given context.

Selects an action by adding noise to neural network paramters and the computing forward with the context vector as input.

Parameters **`context`** (*`torch.Tensor`*) – The context vector to select action for.

Returns The action to take

Return type int

`update_db`(*context: torch.Tensor*, *action: int*, *reward: int*)
Updates transition database with given transition

Parameters

- **`context`** (*`torch.Tensor`*) – Context recieved
- **`action`** (*`int`*) – Action taken
- **`reward`** (*`int`*) – Reward recieved

`update_params`(*action: Optional[int] = None*, *batch_size: int = 512*, *train_epochs: int = 20*)
Update parameters of the agent.

Trains each neural network in the ensemble.

Parameters

- **`action`** (*`Optional[int]`, `optional`*) – Action to update the parameters for. Not applicable in this agent. Defaults to None.
- **`batch_size`** (*`int`, `optional`*) – Size of batch to update parameters with. Defaults to 512
- **`train_epochs`** (*`int`, `optional`*) – Epochs to train neural network for. Defaults to 20

Variational

class `genrl.agents.bandits.contextual.variational.`**`VariationalAgent`**(*bandit: genrl.utils.data_bandits.base.Data*
***kwargs*)

Bases: *`genrl.agents.bandits.contextual.base.DCBAgent`*

Deep contextual bandit agent using variation inference.

Parameters

- **`bandit`** (*`DataBasedBandit`*) – The bandit to solve
- **`init_pulls`** (*`int`, `optional`*) – Number of times to select each action initially. Defaults to 3.
- **`hidden_dims`** (*`List[int]`, `optional`*) – Dimensions of hidden layers of network. Defaults to [50, 50].
- **`init_lr`** (*`float`, `optional`*) – Initial learning rate. Defaults to 0.1.
- **`lr_decay`** (*`float`, `optional`*) – Decay rate for learning rate. Defaults to 0.5.
- **`lr_reset`** (*`bool`, `optional`*) – Whether to reset learning rate ever train interval. Defaults to True.

- **max_grad_norm** (*float, optional*) – Maximum norm of gradients for gradient clipping. Defaults to 0.5.
- **dropout_p** (*Optional[float], optional*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **eval_with_dropout** (*bool, optional*) – Whether or not to use dropout at inference. Defaults to False.
- **noise_std** (*float, optional*) – Standard deviation of noise in bayesian neural network. Defaults to 0.1.
- **device** (*str*) – Device to use for tensor operations. "cpu" for cpu or "cuda" for cuda. Defaults to "cpu".

select_action (*context: torch.Tensor*) → int

Select an action based on given context.

Selects an action by computing a forward pass through the bayesian neural network.

Parameters **context** (*torch.Tensor*) – The context vector to select action for.

Returns The action to take.

Return type int

update_db (*context: torch.Tensor, action: int, reward: int*)

Updates transition database with given transition

Parameters

- **context** (*torch.Tensor*) – Context recieved
- **action** (*int*) – Action taken
- **reward** (*int*) – Reward recieved

update_params (*action: int, batch_size: int = 512, train_epochs: int = 20*)

Update parameters of the agent.

Trains each neural network in the ensemble.

Parameters

- **action** (*Optional[int], optional*) – Action to update the parameters for. Not applicable in this agent. Defaults to None.
- **batch_size** (*int, optional*) – Size of batch to update parameters with. Defaults to 512
- **train_epochs** (*int, optional*) – Epochs to train neural network for. Defaults to 20

2.4.11 Multi-Armed Bandit

Base

class genrl.agents.bandits.multiarmed.base.**MABAgent** (*bandit: genrl.core.bandit.MultiArmedBandit*)

Bases: genrl.core.bandit.BanditAgent

Base Class for Contextual Bandit solving Policy

Parameters

- **bandit** (*MultiArmedlBandit type object*) – The Bandit to solve
- **requires_init_run** – Indicated if initialisation of Q values is required

action_hist
Get the history of actions taken for contexts

Returns List of context, actions pairs

Return type list

counts
Get the number of times each action has been taken

Returns Numpy array with count for each action

Return type numpy.ndarray

regret
Get the current regret

Returns The current regret

Return type float

regret_hist
Get the history of regrets incurred for each step

Returns List of rewards

Return type list

reward_hist
Get the history of rewards received for each step

Returns List of rewards

Return type list

select_action(*context: int*) → int
Select an action

This method needs to be implemented in the specific policy.

Parameters **context** (*int*) – the context to select action for

Returns Selected action

Return type int

update_params(*context: int, action: int, reward: Union[int, float]*) → None
Update parmeters for the policy

This method needs to be implemented in the specific policy.

Parameters

- **context** (*int*) – context for which action is taken
- **action** (*int*) – action taken for the step
- **reward** (*int or float*) – reward obtained for the step

Bayesian Bandit

class genrl.agents.bandits.multiarmed.bayesian.**BayesianUCBMABAgent**(*bandit: genrl.core.bandit.MultiArmedBand*, *alpha: float = 1.0*, *beta: float = 1.0*, *confidence: float = 3.0*)

Bases: `genrl.agents.bandits.multiarmed.base.MABAgent`

Multi-Armed Bandit Solver with Bayesian Upper Confidence Bound based Action Selection Strategy.

Refer to Section 2.7 of Reinforcement Learning: An Introduction.

Parameters

- **bandit** (*MultiArmedlBandit type object*) – The Bandit to solve
- **alpha** (*float*) – alpha value for beta distribution
- **beta** (*float*) – beta values for beta distibution
- **c** (*float*) – Confidence level which controls degree of exploration

a

alpha parameter of beta distribution associated with the policy

Type numpy.ndarray

b

beta parameter of beta distribution associated with the policy

Type numpy.ndarray

confidence

Confidence level which weights the exploration term

Type float

quality

Q values for all the actions for alpha, beta and c

Type numpy.ndarray

select_action(*context: int*) → int

Select an action according to bayesian upper confidence bound

Take action that maximises a weighted sum of the Q values and a beta distribution paramerterized by alpha and beta and weighted by c for each action

Parameters

- **context** (*int*) – the context to select action for
- **t** (*int*) – timestep to choose action for

Returns Selected action

Return type int

update_params(*context: int*, *action: int*, *reward: float*) → None
Update parmeters for the policy

Updates the regret as the difference between max Q value and that of the action. Updates the Q values according to the reward recieved in this step

Parameters

- **context** (*int*) – context for which action is taken
- **action** (*int*) – action taken for the step
- **reward** (*float*) – reward obtained for the step

Bernoulli Bandit

class genrl.agents.bandits.multiarmed.bernoulli_mab.**BernoulliMAB**(*bandits: int = 1*, *arms: int = 5*, *reward_probs: numpy.ndarray = None*, *context_type: str = 'tensor'*)

Bases: genrl.core.bandit.MultiArmedBandit

Contextual Bandit with categorial context and bernoulli reward distribution

Parameters

- **bandits** (*int*) – Number of bandits
- **arms** (*int*) – Number of arms in each bandit
- **reward_probs** (*numpy.ndarray*) – Probabilities of getting rewards

Espilon Greedy

class genrl.agents.bandits.multiarmed.epsgreedy.**EpsGreedyMABAgent**(*bandit: genrl.core.bandit.MultiArmedBandit*, *eps: float = 0.05*)

Bases: *genrl.agents.bandits.multiarmed.base.MABAgent*

Contextual Bandit Policy with Epsilon Greedy Action Selection Strategy.

Refer to Section 2.3 of Reinforcement Learning: An Introduction.

Parameters

- **bandit** (*MultiArmedlBandit type object*) – The Bandit to solve
- **eps** (*float*) – Probability with which a random action is to be selected.

eps
Exploration constant

Type float

quality
Q values assigned by the policy to all actions

Type numpy.ndarray

select_action(*context: int*) → int
Select an action according to epsilon greedy startegy

A random action is selected with espilon probability over the optimal action according to the current Q values to encourage exploration of the policy.

Parameters **context** (*int*) – the context to select action for

Returns Selected action

Return type int

update_params(*context: int*, *action: int*, *reward: float*) → None
Update parmeters for the policy

Updates the regret as the difference between max Q value and that of the action. Updates the Q values according to the reward recieved in this step.

Parameters

- **context** (*int*) – context for which action is taken
- **action** (*int*) – action taken for the step
- **reward** (*float*) – reward obtained for the step

Gaussian

class genrl.agents.bandits.multiarmed.gaussian_mab.**GaussianMAB**(*bandits: int = 10*, *arms: int = 5*, *reward_means: numpy.ndarray = None*, *context_type: str = 'tensor'*)

Bases: genrl.core.bandit.MultiArmedBandit

Contextual Bandit with categorial context and gaussian reward distribution

Parameters

- **bandits** (*int*) – Number of bandits
- **arms** (*int*) – Number of arms in each bandit
- **reward_means** (*numpy.ndarray*) – Mean of gaussian distribution for each reward

Gradient

class genrl.agents.bandits.multiarmed.gradient.**GradientMABAgent**(*bandit: genrl.core.bandit.MultiArmedBandit*, *alpha: float = 0.1*, *temp: float = 0.01*)

Bases: *genrl.agents.bandits.multiarmed.base.MABAgent*

Multi-Armed Bandit Solver with Softmax Action Selection Strategy.

Refer to Section 2.8 of Reinforcement Learning: An Introduction.

Parameters

- **`bandit`** (*`MultiArmedlBandit type object`*) – The Bandit to solve
- **`alpha`** (*`float`*) – The step size parameter for gradient based update
- **`temp`** (*`float`*) – Temperature for softmax distribution over Q values of actions

`alpha`
Step size parameter for gradient based update of policy

Type float

`probability_hist`
History of probabilty values assigned to each action for each timestep

Type numpy.ndarray

`quality`
Q values assigned by the policy to all actions

Type numpy.ndarray

`select_action`(*context: int*) → int
Select an action according by softmax action selection strategy

Action is sampled from softmax distribution computed over the Q values for all actions

Parameters **`context`** (*`int`*) – the context to select action for

Returns Selected action

Return type int

`temp`
Temperature for softmax distribution over Q values of actions

Type float

`update_params`(*context: int*, *action: int*, *reward: float*) → None
Update parmeters for the policy

Updates the regret as the difference between max Q value and that of the action. Updates the Q values through a gradient ascent step

Parameters

- **`context`** (*`int`*) – context for which action is taken
- **`action`** (*`int`*) – action taken for the step
- **`reward`** (*`float`*) – reward obtained for the step

Thmopson Sampling

class genrl.agents.bandits.multiarmed.thompson.**ThompsonSamplingMABAgent**(*bandit: genrl.core.bandit.MultiArm al- pha: float = 1.0*, *beta: float = 1.0*)

Bases: *genrl.agents.bandits.multiarmed.base.MABAgent*

Multi-Armed Bandit Solver with Bayesian Upper Confidence Bound based Action Selection Strategy.

Parameters

- **bandit** (*MultiArmedlBandit type object*) – The Bandit to solve
- **a** (*float*) – alpha value for beta distribution
- **b** (*float*) – beta values for beta distibution

a

alpha parameter of beta distribution associated with the policy

Type numpy.ndarray

b

beta parameter of beta distribution associated with the policy

Type numpy.ndarray

quality

Q values for all the actions for alpha, beta and c

Type numpy.ndarray

select_action(*context: int*) → int

Select an action according to Thompson Sampling

Samples are taken from beta distribution parameterized by alpha and beta for each action. The action with the highest sample is selected.

Parameters **context** (*int*) – the context to select action for

Returns Selected action

Return type int

update_params(*context: int*, *action: int*, *reward: float*) → None

Update parmeters for the policy

Updates the regret as the difference between max Q value and that of the action. Updates the alpha value of beta distribution by adding the reward while the beta value is updated by adding 1 - reward. Update the counts the action taken.

Parameters

- **context** (*int*) – context for which action is taken
- **action** (*int*) – action taken for the step

- **reward** (*float*) – reward obtained for the step

Upper Confidence Bound

class genrl.agents.bandits.multiarmed.ucb.**UCBMABAgent**(*bandit: genrl.core.bandit.MultiArmedBandit*, *confidence: float = 1.0*)

Bases: *genrl.agents.bandits.multiarmed.base.MABAgent*

Multi-Armed Bandit Solver with Upper Confidence Bound based Action Selection Strategy.

Refer to Section 2.7 of Reinforcement Learning: An Introduction.

Parameters

- **bandit** (*MultiArmedlBandit type object*) – The Bandit to solve
- **c** (*float*) – Confidence level which controls degree of exploration

confidence

Confidence level which weights the exploration term

Type float

quality

q values assigned by the policy to all actions

Type numpy.ndarray

select_action(*context: int*) → int

Select an action according to upper confidence bound action selction

Take action that maximises a weighted sum of the Q values for the action and an exploration encouragement term controlled by c.

Parameters **context** (*int*) – the context to select action for

Returns Selected action

Return type int

update_params(*context: int*, *action: int*, *reward: float*) → None

Update parmeters for the policy

Updates the regret as the difference between max Q value and that of the action. Updates the Q values according to the reward recieved in this step.

Parameters

- **context** (*int*) – context for which action is taken
- **action** (*int*) – action taken for the step
- **reward** (*float*) – reward obtained for the step

2.5 Environments

2.5.1 Environments

Subpackages

Vectorized Envrionments

Submodules

genrl.environments.vec_env.monitor module

class `genrl.environments.vec_env.monitor.`**`VecMonitor`**(*venv: genrl.environments.vec_env.vector_envs.VecEnv, history_length: int = 0, info_keys: Tuple = ()*)

Bases: *`genrl.environments.vec_env.wrappers.VecEnvWrapper`*

Monitor class for VecEnvs. Saves important variables into the info dictionary

Parameters

- **`venv`** (*`object`*) – Vectorized Environment
- **`history_length`** (*`int`*) – Length of history for episode rewards and episode lengths
- **`info_keys`** (*`tuple or list`*) – Important variables to save

`reset`() → numpy.ndarray

Resets Vectorized Environment

Returns Initial observations

Return type Numpy Array

`step`(*actions: numpy.ndarray*) → Tuple

Steps through all the environments and records important information

Parameters **`actions`** (*`Numpy Array`*) – Actions to be taken for the Vectorized Environment

Returns States, rewards, dones, infos

genrl.environments.vec_env.normalize module

class `genrl.environments.vec_env.normalize.`**`VecNormalize`**(*venv: genrl.environments.vec_env.vector_envs.VecEnv, norm_obs: bool = True, norm_reward: bool = True, clip_reward: float = 20.0*)

Bases: *`genrl.environments.vec_env.wrappers.VecEnvWrapper`*

Wrapper to implement Normalization of observations and rewards for VecEnvs

Parameters

- **`venv`** (*`Vectorized Environment`*) – The Vectorized environment
- **`n_envs`** (*`int`*) – Number of environments in VecEnv
- **`norm_obs`** (*`bool`*) – True if observations should be normalized, else False
- **`norm_reward`** (*`bool`*) – True if rewards should be normalized, else False
- **`clip_reward`** (*`float`*) – Maximum absolute value for rewards

`close`()

Close all individual environments in the Vectorized Environment

reset() → numpy.ndarray
Resets Vectorized Environment

Returns Initial observations

Return type Numpy Array

step(*actions: numpy.ndarray*) → Tuple
Steps through all the environments and normalizes the observations and rewards (if enabled)

Parameters **actions** (`Numpy Array`) – Actions to be taken for the Vectorized Environment

Returns States, rewards, dones, infos

genrl.environments.vec_env.utils module

class genrl.environments.vec_env.utils.**RunningMeanStd**(*epsilon: float = 0.0001, shape: Tuple = ()*)
Bases: `object`

Utility Function to compute a running mean and variance calculator

Parameters

- **epsilon** (`float`) – Small number to prevent division by zero for calculations
- **shape** (`Tuple`) – Shape of the RMS object

update(*batch: torch.Tensor*)

genrl.environments.vec_env.vector_envs module

class genrl.environments.vec_env.vector_envs.**SerialVecEnv**(**args, **kwargs*)
Bases: `genrl.environments.vec_env.vector_envs.VecEnv`

Constructs a wrapper for serial execution through envs.

close()
Closes all envs

get_spaces()

images() → List[T]
Returns an array of images from each env render

render(*mode='human'*)
Renders all envs in a tiles format similar to baselines

Parameters **mode** (`string`) – (Can either be 'human' or 'rgb_array'. Displays tiled images in 'human' and returns tiled images in 'rgb_array')

reset() → torch.Tensor
Resets all envs

reset_single_env(*i: int*) → torch.Tensor
Resets single environment

step(*actions: torch.Tensor*) → Tuple
Steps through all envs serially

Parameters **actions** (`Iterable of ints/floats`) – Actions from the model

class genrl.environments.vec_env.vector_envs.**SubProcessVecEnv**(**args*, ***kwargs*)
Bases: *genrl.environments.vec_env.vector_envs.VecEnv*

Constructs a wrapper for parallel execution through envs.

close()
Closes all environments and processes

get_spaces() → Tuple
Returns state and action spaces of environments

reset() → torch.Tensor
Resets environments

Returns States after environment reset

seed(*seed: int = None*)
Sets seed for reproducability

step(*actions: torch.Tensor*) → Tuple
Steps through environments serially

Parameters **actions** (*Iterable of ints/floats*) – Actions from the model

class genrl.environments.vec_env.vector_envs.**VecEnv**(*envs: List[T], n_envs: int = 2*)
Bases: abc.ABC

Base class for multiple environments.

Parameters

- **env** (*Gym Environment*) – Gym environment to be vectorised
- **n_envs** (*int*) – Number of environments

action_shape

action_spaces

close()

n_envs

obs_shape

observation_spaces

reset()

sample() → List[T]
Return samples of actions from each environment

seed(*seed: int*)
Set seed for reproducibility in all environments

step(*actions*)

genrl.environments.vec_env.vector_envs.**worker**(*parent_conn: multiprocessing.context.BaseContext.Pipe, child_conn: multiprocessing.context.BaseContext.Pipe, env: gym.core.Env*)
Worker class to facilitate multiprocessing

Parameters

- **parent_conn** (*Multiprocessing Pipe Connection*) – Parent connection of Pipe
- **child_conn** (*Multiprocessing Pipe Connection*) – Child connection of Pipe
- **env** (*Gym Environment*) – Gym environment we need multiprocessing for

genrl.environments.vec_env.wrappers module

class genrl.environments.vec_env.wrappers.**VecEnvWrapper**(*venv*)

Bases: *genrl.environments.vec_env.vector_envs.VecEnv*

close()

render(*mode='human'*)

reset()

step(*actions*)

Module contents

Submodules

genrl.environments.action_wrappers module

class genrl.environments.action_wrappers.**ClipAction**(*env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv]*)

Bases: gym.core.ActionWrapper

Action Wrapper to clip actions

Parameters **env** (*object*) – The environment whose actions need to be clipped

action(*action: numpy.ndarray*) → numpy.ndarray

class genrl.environments.action_wrappers.**RescaleAction**(*env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv], low: int, high: int*)

Bases: gym.core.ActionWrapper

Action Wrapper to rescale actions

Parameters

- **env** (*object*) – The environment whose actions need to be rescaled
- **low** (*int*) – Lower limit of action
- **high** (*int*) – Upper limit of action

action(*action: numpy.ndarray*) → numpy.ndarray

genrl.environments.atari_preprocessing module

class `genrl.environments.atari_preprocessing.`**`AtariPreprocessing`**(*env: gym.core.Env, frameskip: Union[Tuple, int] = (2, 5), grayscale: bool = True, screen_size: int = 84*)

Bases: `gym.core.Wrapper`

Implementation for Image preprocessing for Gym Atari environments. Implements: 1) Frameskip 2) Grayscale 3) Downsampling to square image

param env Atari environment

param frameskip Number of steps between actions. E.g. frameskip=4 will mean 1 action will be taken for every 4 frames. It'll be a tuple

if non-deterministic and a random number will be chosen from (2, 5)

param grayscale Whether or not the output should be converted to grayscale

param screen_size Size of the output screen (square output)

type env Gym Environment

type frameskip tuple or int

type grayscale boolean

type screen_size int

`reset`() → numpy.ndarray

Resets state of environment

Returns Initial state

Return type NumPy array

`step`(*action: numpy.ndarray*) → numpy.ndarray

Step through Atari environment for given action

Parameters **`action`** (*`NumPy array`*) – Action taken by agent

Returns Current state, reward(for frameskip number of actions), done, info

genrl.environments.atari_wrappers module

class `genrl.environments.atari_wrappers.`**`FireReset`**(*env: gym.core.Env*)

Bases: `gym.core.Wrapper`

Some Atari environments do not actually do anything until a specific action (the fire action) is taken, so we make it take the action before starting the training process

Parameters **`env`** (*`Gym Environment`*) – Atari environment

`reset`() → numpy.ndarray

Resets state of environment. Performs the noop action a random number of times to introduce stochasticity

Returns Initial state

Return type NumPy array

class `genrl.environments.atari_wrappers.`**`NoopReset`**(*env: gym.core.Env*, *max_noops: int = 30*)

Bases: `gym.core.Wrapper`

Some Atari environments always reset to the same state. So we take a random number of some empty (noop) action to introduce some stochasticity.

Parameters

- **`env`** (*`Gym Environment`*) – Atari environment
- **`max_noops`** (*`int`*) – Maximum number of Noops to be taken

`reset`() → numpy.ndarray

Resets state of environment. Performs the noop action a random number of times to introduce stochasticity

Returns Initial state

Return type NumPy array

`step`(*action: numpy.ndarray*) → numpy.ndarray

Step through underlying Atari environment for given action

Parameters **`action`** (*`NumPy array`*) – Action taken by agent

Returns Current state, reward(for frameskip number of actions), done, info

genrl.environments.base_wrapper module

class `genrl.environments.base_wrapper.`**`BaseWrapper`**(*env: Any*, *batch_size: int = None*)

Bases: `abc.ABC`

Base class for all wrappers

`batch_size`

The number of batches trained per update

`close`() → None

Closes environment and performs any other cleanup

Must be overridden by subclasses

`render`() → None

Render the environment

`reset`() → None

Resets state of environment

Must be overriden by subclasses

Returns Initial state

`seed`(*seed: int = None*) → None

Set seed for environment

`step`(*action: numpy.ndarray*) → None

Step through the environment

Must be overriden by subclasses

genrl.environments.frame_stack module

class genrl.environments.frame_stack.**FrameStack**(*env: gym.core.Env*, *framestack: int = 4*, *compress: bool = True*)

Bases: `gym.core.Wrapper`

Wrapper to stack the last few(4 by default) observations of agent efficiently

Parameters

- **env** (*Gym Environment*) – Environment to be wrapped
- **framestack** (*int*) – Number of frames to be stacked
- **compress** (*bool*) – True if we want to use LZ4 compression to conserve memory usage

reset() → numpy.ndarray

Resets environment

Returns Initial state of environment

Return type NumPy Array

step(*action: numpy.ndarray*) → numpy.ndarray

Steps through environment

Parameters **action** (*NumPy Array*) – Action taken by agent

Returns Next state, reward, done, info

Return type NumPy Array, float, boolean, dict

class genrl.environments.frame_stack.**LazyFrames**(*frames: List[T]*, *compress: bool = False*)

Bases: `object`

Efficient data structure to save each frame only once. Can use LZ4 compression to optimizer memory usage.

Parameters

- **frames** (*collections.deque*) – List of frames that needs to converted to a LazyFrames data structure
- **compress** (*boolean*) – True if we want to use LZ4 compression to conserve memory usage

shape

Returns dimensions of other object

genrl.environments.gym_wrapper module

class genrl.environments.gym_wrapper.**GymWrapper**(*env: gym.core.Env*)

Bases: `gym.core.Wrapper`

Wrapper class for all Gym Environments

Parameters

- **env** (*string*) – Gym environment name
- **n_envs** (*None, int*) – Number of environments. None if not vectorised
- **parallel** (*boolean*) – If vectorised, should environments be run through serially or parallelly

action_shape

close() → None
Closes environment

obs_shape

render(*mode: str = 'human'*) → None
Renders all envs in a tiles format similar to baselines.

Parameters **mode** (*string*) – Can either be 'human' or 'rgb_array'. Displays tiled images in 'human' and returns tiled images in 'rgb_array'

reset() → numpy.ndarray
Resets environment

Returns Initial state

sample() → numpy.ndarray
Shortcut method to directly sample from environment's action space

Returns Random action from action space

Return type NumPy Array

seed(*seed: int = None*) → None
Set environment seed

Parameters **seed** (*int*) – Value of seed

step(*action: numpy.ndarray*) → numpy.ndarray
Steps the env through given action

Parameters **action** (*NumPy array*) – Action taken by agent

Returns Next observation, reward, game status and debugging info

genrl.environments.suite module

genrl.environments.suite.**AtariEnv**(*env_id: str, wrapper_list: List[T] = [<class 'genrl.environments.atari_preprocessing.AtariPreprocessing'>, <class 'genrl.environments.atari_wrappers.NoopReset'>, <class 'genrl.environments.atari_wrappers.FireReset'>, <class 'genrl.environments.time_limit.AtariTimeLimit'>, <class 'genrl.environments.frame_stack.FrameStack'>]*) → gym.core.Env
Function to apply wrappers for all Atari envs by Trainer class

Parameters

- **env** (*string*) – Environment Name
- **wrapper_list** (*list or tuple*) – List of wrappers to use

Returns Gym Atari Environment

Return type object

genrl.environments.suite.**GymEnv**(*env_id: str*) → gym.core.Env
Function to apply wrappers for all regular Gym envs by Trainer class

Parameters **env** (*string*) – Environment Name

Returns Gym Environment

Return type object

`genrl.environments.suite.`**`VectorEnv`**(*env_id: str*, *n_envs: int = 2*, *parallel: int = False*, *env_type: str = 'gym'*) → genrl.environments.vec_env.vector_envs.VecEnv

Chooses the kind of Vector Environment that is required

param env_id Gym environment to be vectorised

param n_envs Number of environments

param parallel True if we want environments to run parallely and (

subprocesses, False if we want environments to run serially one after the other)

param env_type Type of environment. Currently, we support ["gym", "atari"]

type env_id string

type n_envs int

type parallel False

type env_type string

returns Vector Environment

rtype object

genrl.environments.time_limit module

`class` `genrl.environments.time_limit.`**`AtariTimeLimit`**(*env*, *max_episode_len=None*)

Bases: `gym.core.Wrapper`

`reset`(***kwargs*)

Resets the state of the environment and returns an initial observation.

Returns the initial observation.

Return type observation (object)

`step`(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **`action`** (`object`) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

`class` `genrl.environments.time_limit.`**`TimeLimit`**(*env*, *max_episode_len=None*)

Bases: `gym.core.Wrapper`

`reset`(***kwargs*)

Resets the state of the environment and returns an initial observation.

Returns the initial observation.

Return type observation (object)

step (*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

Module contents

2.6 Core

2.6.1 ActorCritic

class genrl.core.actor_critic.**CNNActorCritic** (*framestack: int*, *action_dim: gym.spaces.space.Space*, *policy_layers: Tuple = (256,)*, *value_layers: Tuple = (256,)*, *val_type: str = 'V'*, *discrete: bool = True*, **args*, ***kwargs*)

Bases: *genrl.core.base.BaseActorCritic*

CNN Actor Critic

param framestack Number of previous frames to stack together

param action_dim Action dimensions of the environment

param fc_layers Sizes of hidden layers

param val_type Specifies type of value function: (

"V" for V(s), "Qs" for Q(s), "Qsa" for Q(s,a))

param discrete True if action space is discrete, else False

param framestack Number of previous frames to stack together

type action_dim int

type fc_layers tuple or list

type val_type str

type discrete bool

get_action (*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor

Get action from the Actor based on input

param state The state being passed as input to the Actor

param deterministic (True if the action space is deterministic,

else False)

type state Tensor

type deterministic boolean

returns action

`get_params`()

`get_value`(*inp: torch.Tensor*) → torch.Tensor
Get value from the Critic based on input

Parameters **`inp`** (*`Tensor`*) – Input to the Critic

Returns value

class `genrl.core.actor_critic.`**`MlpActorCritic`**(*state_dim: gym.spaces.space.Space, action_dim: gym.spaces.space.Space, shared_layers: None, policy_layers: Tuple = (32, 32), value_layers: Tuple = (32, 32), val_type: str = 'V', discrete: bool = True, **kwargs*)

Bases: *`genrl.core.base.BaseActorCritic`*

MLP Actor Critic

`state_dim`
State dimensions of the environment

Type int

`action_dim`
Action space dimensions of the environment

Type int

`policy_layers`
Hidden layers in the policy MLP

Type `list` or `tuple`

`value_layers`
Hidden layers in the value MLP

Type `list` or `tuple`

`val_type`
Value type of the critic network

Type str

`discrete`
True if the action space is discrete, else False

Type bool

`sac`
True if a SAC-like network is needed, else False

Type bool

`activation`
Activation function to be used. Can be either "tanh" or "relu"

Type str

`get_params`()

class genrl.core.actor_critic.**MlpSharedActorCritic**(*state_dim: gym.spaces.space.Space*, *action_dim: gym.spaces.space.Space*, *shared_layers: Tuple = (32, 32)*, *policy_layers: Tuple = (32, 32)*, *value_layers: Tuple = (32, 32)*, *val_type: str = 'V'*, *discrete: bool = True*, ***kwargs*)

Bases: `genrl.core.base.BaseActorCritic`

MLP Shared Actor Critic

state_dim
State dimensions of the environment

Type int

action_dim
Action space dimensions of the environment

Type int

shared_layers
Hidden layers in the shared MLP

Type `list` or `tuple`

policy_layers
Hidden layers in the policy MLP

Type `list` or `tuple`

value_layers
Hidden layers in the value MLP

Type `list` or `tuple`

val_type
Value type of the critic network

Type str

discrete
True if the action space is discrete, else False

Type bool

sac
True if a SAC-like network is needed, else False

Type bool

activation
Activation function to be used. Can be either "tanh" or "relu"

Type str

get_action(*state: torch.Tensor*, *deterministic: bool = False*)
Get Actions from the actor

Arg: state (`torch.Tensor`): The state(s) being passed to the critics deterministic (bool): True if the action space is deterministic, else False

Returns

List of actions as estimated by the critic distribution (): The distribution from which the action was sampled

(None if determinist

Return type action (`list`)

`get_features`(*state: torch.Tensor*)

Extract features from the state, which is then an input to get_action and get_value

Parameters **`state`** (`torch.Tensor`) – The state(s) being passed

Returns The feature(s) extracted from the state

Return type features (`torch.Tensor`)

`get_params`()

`get_value`(*state: torch.Tensor*)

Get Values from the Critic

Arg: state (`torch.Tensor`): The state(s) being passed to the critics

Returns List of values as estimated by the critic

Return type values (`list`)

class `genrl.core.actor_critic.`**`MlpSharedSingleActorTwoCritic`**(*state_dim: gym.spaces.space.Space, action_dim: gym.spaces.space.Space, shared_layers: Tuple = (32, 32), policy_layers: Tuple = (32, 32), value_layers: Tuple = (32, 32), val_type: str = 'Qsa', discrete: bool = True, num_critics: int = 2, **kwargs*)

Bases: *`genrl.core.actor_critic.MlpSingleActorTwoCritic`*

MLP Actor Critic

`state_dim`

State dimensions of the environment

Type int

`action_dim`

Action space dimensions of the environment

Type int

`shared_layers`

Hidden layers in the shared MLP

Type `list` or `tuple`

`policy_layers`
Hidden layers in the policy MLP

Type `list` or `tuple`

`value_layers`
Hidden layers in the value MLP

Type `list` or `tuple`

`val_type`
Value type of the critic network

Type str

`discrete`
True if the action space is discrete, else False

Type bool

`num_critics`
Number of critics in the architecture

Type int

`sac`
True if a SAC-like network is needed, else False

Type bool

`activation`
Activation function to be used. Can be either "tanh" or "relu"

Type str

`get_action`(*state: torch.Tensor*, *deterministic: bool = False*)
Get Actions from the actor

Arg: state (`torch.Tensor`): The state(s) being passed to the critics deterministic (bool): True if the action space is deterministic, else False

Returns

List of actions as estimated by the critic distribution (): The distribution from which the action was sampled

(None if deterministic)

Return type action (`list`)

`get_features`(*state: torch.Tensor*)
Extract features from the state, which is then an input to get_action and get_value

Parameters **`state`** (`torch.Tensor`) – The state(s) being passed

Returns The feature(s) extracted from the state

Return type features (`torch.Tensor`)

`get_params`()

`get_value`(*state: torch.Tensor*, *mode='first'*)
Get Values from both the Critic

Arg: state (`torch.Tensor`): The state(s) being passed to the critics mode (str): What values should be returned. Types:

"both" –> Both values will be returned "min" –> The minimum of both values will be returned "first" –> The value from the first critic only will be returned

Returns List of values as estimated by each individual critic

Return type values (`list`)

class `genrl.core.actor_critic.`**`MlpSingleActorTwoCritic`**(*state_dim: gym.spaces.space.Space, action_dim: gym.spaces.space.Space, policy_layers: Tuple = (32, 32), value_layers: Tuple = (32, 32), val_type: str = 'V', discrete: bool = True, num_critics: int = 2, **kwargs*)

Bases: *`genrl.core.base.BaseActorCritic`*

MLP Actor Critic

`state_dim`
State dimensions of the environment

Type int

`action_dim`
Action space dimensions of the environment

Type int

`policy_layers`
Hidden layers in the policy MLP

Type `list` or `tuple`

`value_layers`
Hidden layers in the value MLP

Type `list` or `tuple`

`val_type`
Value type of the critic network

Type str

`discrete`
True if the action space is discrete, else False

Type bool

`num_critics`
Number of critics in the architecture

Type int

`sac`
True if a SAC-like network is needed, else False

Type bool

`activation`
Activation function to be used. Can be either "tanh" or "relu"

Type str

`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.

`get_action`(*state: torch.Tensor*, *deterministic: bool = False*)
Get Actions from the actor

Arg: state (`torch.Tensor`): The state(s) being passed to the critics deterministic (bool): True if the action space is deterministic, else False

Returns

List of actions as estimated by the critic distribution (): The distribution from which the action was sampled

(None if determinist

Return type action (`list`)

`get_params`()

`get_value`(*state: torch.Tensor*, *mode='first'*) → torch.Tensor
Get Values from the Critic

Arg: state (`torch.Tensor`): The state(s) being passed to the critics mode (str): What values should be returned. Types:

"both" –> Both values will be returned "min" –> The minimum of both values will be returned "first" –> The value from the first critic only will be returned

Returns List of values as estimated by each individual critic

Return type values (`list`)

`genrl.core.actor_critic.`**`get_actor_critic_from_name`**(*name_: str*)
Returns Actor Critic given the type of the Actor Critic

Parameters **`ac_name`** (*`str`*) – Name of the policy needed

Returns Actor Critic class to be used

2.6.2 Base

class `genrl.core.base.`**`BaseActorCritic`**
Bases: `torch.nn.modules.module.Module`

Basic implementation of a general Actor Critic

`get_action`(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor

Get action from the Actor based on input

param state The state being passed as input to the Actor

param deterministic (True if the action space is deterministic,

else False)

type state Tensor

type deterministic boolean

returns action

`get_value`(*state: torch.Tensor*) → torch.Tensor
Get value from the Critic based on input

Parameters **`state`** (*`Tensor`*) – Input to the Critic

Returns value

`class` `genrl.core.base.`**`BasePolicy`**(*state_dim: int*, *action_dim: int*, *hidden: Tuple*, *discrete: bool*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`

Basic implementation of a general Policy

Parameters

- **`state_dim`** (*`int`*) – State dimensions of the environment
- **`action_dim`** (*`int`*) – Action dimensions of the environment
- **`hidden`** (*`tuple or list`*) – Sizes of hidden layers
- **`discrete`** (*`bool`*) – True if action space is discrete, else False

`forward`(*state: torch.Tensor*) → Tuple[torch.Tensor, Optional[torch.Tensor]]
Defines the computation performed at every call.

Parameters **`state`** (*`Tensor`*) – The state being passed as input to the policy

`get_action`(*state: torch.Tensor*, *deterministic: bool = False*) → torch.Tensor

Get action from policy based on input

param state The state being passed as input to the policy

param deterministic (True if the action space is deterministic,

else False)

type state Tensor

type deterministic boolean

returns action

`class` `genrl.core.base.`**`BaseValue`**(*state_dim: int*, *action_dim: int*)
Bases: `torch.nn.modules.module.Module`

Basic implementation of a general Value function

`forward`(*state: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Parameters **`state`** (*`Tensor`*) – Input to value function

`get_value`(*state: torch.Tensor*) → torch.Tensor
Get value from value function based on input

Parameters **`state`** (*`Tensor`*) – Input to value function

Returns Value

2.6.3 Buffers

class `genrl.core.buffers.`**`PrioritizedBuffer`**(*capacity: int*, *alpha: float = 0.6*, *beta: float = 0.4*)

Bases: `object`

Implements the Prioritized Experience Replay Mechanism

Parameters

- **`capacity`** (*`int`*) – Size of the replay buffer
- **`alpha`** (*`int`*) – Level of prioritization

`pos`

`push`(*inp: Tuple*) → None

Adds new experience to buffer

param inp (Tuple containing *state*, *action*, *reward*,

next_state **and** ***done*****)**

type inp tuple

returns None

`sample`(*batch_size: int*, *beta: float = None*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

(Returns randomly sampled memories from replay memory along with their

respective indices and weights)

param batch_size Number of samples per batch

param beta (Bias exponent used to correct

Importance Sampling (IS) weights)

type batch_size int

type beta float

returns (Tuple containing *states*, *actions*, *next_states*,

rewards, *dones*, *indices* and *weights*)

`update_priorities`(*batch_indices: Tuple*, *batch_priorities: Tuple*) → None

Updates list of priorities with new order of priorities

param batch_indices List of indices of batch

param batch_priorities (List of priorities of the batch at the

specific indices)

type batch_indices list or tuple

type batch_priorities list or tuple

class genrl.core.buffers.**PrioritizedReplayBufferSamples**(*states*, *actions*, *rewards*, *next_states*, *dones*, *indices*, *weights*)

Bases: `tuple`

actions
Alias for field number 1

dones
Alias for field number 4

indices
Alias for field number 5

next_states
Alias for field number 3

rewards
Alias for field number 2

states
Alias for field number 0

weights
Alias for field number 6

class genrl.core.buffers.**ReplayBuffer**(*capacity: int*)

Bases: `object`

Implements the basic Experience Replay Mechanism

Parameters **capacity** (*int*) – Size of the replay buffer

push(*inp: Tuple*) → None
Adds new experience to buffer

Parameters **inp** (*tuple*) – Tuple containing state, action, reward, next_state and done

Returns None

sample(*batch_size: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Returns randomly sampled experiences from replay memory

param batch_size Number of samples per batch

type batch_size int

returns (Tuple composing of *state*, *action*, *reward*,

next_state and *done*)

class genrl.core.buffers.**ReplayBufferSamples**(*states*, *actions*, *rewards*, *next_states*, *dones*)

Bases: `tuple`

actions
Alias for field number 1

dones
Alias for field number 4

next_states
Alias for field number 3

rewards
Alias for field number 2

states
Alias for field number 0

2.6.4 Noise

class `genrl.core.noise.`**ActionNoise**(*mean: float*, *std: float*)
Bases: `abc.ABC`

Base class for Action Noise

Parameters

- **mean** (*float*) – Mean of noise distribution
- **std** (*float*) – Standard deviation of noise distribution

mean
Returns mean of noise distribution

std
Returns standard deviation of noise distribution

class `genrl.core.noise.`**NoisyLinear**(*in_features: int*, *out_features: int*, *std_init: float = 0.4*)
Bases: `torch.nn.modules.module.Module`

Noisy Linear Layer Class

Class to represent a Noisy Linear class (noisy version of nn.Linear)

in_features
Input dimensions

Type int

out_features
Output dimensions

Type int

std_init
Weight initialisation constant

Type float

forward(*state: torch.Tensor*) → torch.Tensor
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.

reset_noise() → None
Reset noise components of layer

reset_parameters() → None
Reset parameters of layer

class genrl.core.noise.**NormalActionNoise**(*mean: float*, *std: float*)

Bases: `genrl.core.noise.ActionNoise`

Normal implementation of Action Noise

Parameters

- **mean** (*float*) – Mean of noise distribution
- **std** (*float*) – Standard deviation of noise distribution

reset() → None

class genrl.core.noise.**OrnsteinUhlenbeckActionNoise**(*mean: float*, *std: float*, *theta: float = 0.15*, *dt: float = 0.01*, *initial_noise: torch.Tensor = None*)

Bases: `genrl.core.noise.ActionNoise`

Ornstein Uhlenbeck implementation of Action Noise

Parameters

- **mean** (*float*) – Mean of noise distribution
- **std** (*float*) – Standard deviation of noise distribution
- **theta** (*float*) – Parameter used to solve the Ornstein Uhlenbeck process
- **dt** (*float*) – Small parameter used to solve the Ornstein Uhlenbeck process
- **initial_noise** (*torch.Tensor*) – Initial noise distribution

reset() → None

Reset the initial noise value for the noise distribution sampling

2.6.5 Policies

class genrl.core.policies.**CNNPolicy**(*framestack: int*, *action_dim: int*, *hidden: Tuple = (32, 32)*, *discrete: bool = True*, **args*, ***kwargs*)

Bases: `genrl.core.base.BasePolicy`

CNN Policy

Parameters

- **framestack** (*int*) – Number of previous frames to stack together
- **action_dim** (*int*) – Action dimensions of the environment
- **fc_layers** (*tuple or list*) – Sizes of hidden layers
- **discrete** (*bool*) – True if action space is discrete, else False
- **channels** (*list or tuple*) – Channel sizes for cnn layers

forward(*state: numpy.ndarray*) → numpy.ndarray

Defines the computation performed at every call.

Parameters **state** (*Tensor*) – The state being passed as input to the policy

class genrl.core.policies.**MlpPolicy**(*state_dim: int*, *action_dim: int*, *hidden: Tuple = (32, 32)*, *discrete: bool = True*, **args*, ***kwargs*)

Bases: `genrl.core.base.BasePolicy`

MLP Policy

Parameters

- **state_dim** (*int*) – State dimensions of the environment
- **action_dim** (*int*) – Action dimensions of the environment
- **hidden** (*tuple or list*) – Sizes of hidden layers
- **discrete** (*bool*) – True if action space is discrete, else False

genrl.core.policies.**get_policy_from_name**(*name_: str*)

Returns policy given the name of the policy

Parameters **name** (*str*) – Name of the policy needed

Returns Policy Function to be used

2.6.6 RolloutStorage

class genrl.core.rollout_storage.**BaseBuffer**(*buffer_size: int, env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv], device: Union[torch.device, str] = 'cpu'*)

Bases: object

Base class that represent a buffer (rollout or replay) :param buffer_size: (int) Max number of element in the buffer :param env: (Environment) The environment being trained on :param device: (Union[torch.device, str]) PyTorch device

to which the values will be converted

Parameters **n_envs** – (int) Number of parallel environments

add(**args*, ***kwargs*) → None

Add elements to the buffer.

extend(**args*, ***kwargs*) → None

Add a new batch of transitions to the buffer

reset() → None

Reset the buffer.

sample(*batch_size: int*)

Parameters **batch_size** – (int) Number of element to sample

Returns (Union[RolloutBufferSamples, ReplayBufferSamples])

size() → int

Returns (int) The current size of the buffer

static swap_and_flatten(*arr: numpy.ndarray*) → numpy.ndarray

Swap and then flatten axes 0 (buffer_size) and 1 (n_envs) to convert shape from [n_steps, n_envs, ...] (when ... is the shape of the features) to [n_steps * n_envs, ...] (which maintain the order) :param arr: (np.ndarray) :return: (np.ndarray)

to_torch(*array: numpy.ndarray*, *copy: bool = True*) → torch.Tensor

Convert a numpy array to a PyTorch tensor. Note: it copies the data by default :param array: (np.ndarray) :param copy: (bool) Whether to copy or not the data

(may be useful to avoid changing things be reference)

Returns (torch.Tensor)

class genrl.core.rollout_storage.**ReplayBufferSamples**(*observations*, *actions*, *next_observations*, *dones*, *rewards*)

Bases: `tuple`

actions
Alias for field number 1

dones
Alias for field number 3

next_observations
Alias for field number 2

observations
Alias for field number 0

rewards
Alias for field number 4

class genrl.core.rollout_storage.**RolloutBuffer**(*buffer_size: int*, *env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv]*, *device: Union[torch.device, str] = 'cpu'*, *gae_lambda: float = 1*, *gamma: float = 0.99*)

Bases: *genrl.core.rollout_storage.BaseBuffer*

Rollout buffer used in on-policy algorithms like A2C/PPO. :param buffer_size: (int) Max number of element in the buffer :param env: (Environment) The environment being trained on :param device: (torch.device) :param gae_lambda: (float) Factor for trade-off of bias vs variance for Generalized Advantage Estimator

Equivalent to classic advantage when set to 1.

Parameters

- **gamma** – (float) Discount factor
- **n_envs** – (int) Number of parallel environments

add(*obs: None._VariableFunctions.zeros*, *action: None._VariableFunctions.zeros*, *reward: None._VariableFunctions.zeros*, *done: None._VariableFunctions.zeros*, *value: torch.Tensor*, *log_prob: torch.Tensor*) → None

Parameters

- **obs** – (torch.zeros) Observation
- **action** – (torch.zeros) Action
- **reward** – (torch.zeros)
- **done** – (torch.zeros) End of episode signal.
- **value** – (torch.Tensor) estimated value of the current state following the current policy.
- **log_prob** – (torch.Tensor) log probability of the action following the current policy.

get(*batch_size: Optional[int] = None*) → Generator[genrl.core.rollout_storage.RolloutBufferSamples, None, None]

reset() → None
Reset the buffer.

class `genrl.core.rollout_storage.`**`RolloutBufferSamples`**(*observations*, *actions*, *old_values*, *old_log_prob*, *advantages*, *returns*)

Bases: `tuple`

`actions`
Alias for field number 1

`advantages`
Alias for field number 4

`observations`
Alias for field number 0

`old_log_prob`
Alias for field number 3

`old_values`
Alias for field number 2

`returns`
Alias for field number 5

class `genrl.core.rollout_storage.`**`RolloutReturn`**(*episode_reward*, *episode_timesteps*, *n_episodes*, *continue_training*)

Bases: `tuple`

`continue_training`
Alias for field number 3

`episode_reward`
Alias for field number 0

`episode_timesteps`
Alias for field number 1

`n_episodes`
Alias for field number 2

2.6.7 Values

class `genrl.core.values.`**`CnnCategoricalValue`**(**args*, ***kwargs*)

Bases: *`genrl.core.values.CnnNoisyValue`*

Class for Categorical DQN's CNN Q-Value function

`framestack`
No. of frames being passed into the Q-value function

Type int

`action_dim`
Action space dimensions

Type int

`fc_layers`
Fully connected layer dimensions

Type `tuple`

`noisy_layers`
Noisy layer dimensions

Type `tuple`

`num_atoms`
Number of atoms used to discretise the Categorical DQN value distribution

Type int

`forward`(*state: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Parameters **`state`** (*`Tensor`*) – Input to value function

class `genrl.core.values.`**`CnnDuelingValue`**(**args*, ***kwargs*)
Bases: *`genrl.core.values.CnnValue`*

Class for Dueling DQN's MLP Q-Value function

`framestack`
No. of frames being passed into the Q-value function

Type int

`action_dim`
Action space dimensions

Type int

`fc_layers`
Hidden layer dimensions

Type `tuple`

`forward`(*inp: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Parameters **`state`** (*`Tensor`*) – Input to value function

class `genrl.core.values.`**`CnnNoisyValue`**(**args*, ***kwargs*)
Bases: *`genrl.core.values.CnnValue`*, *`genrl.core.values.MlpNoisyValue`*

Class for Noisy DQN's CNN Q-Value function

`state_dim`
Number of previous frames to stack together

Type int

`action_dim`
Action space dimensions

Type int

`fc_layers`
Fully connected layer dimensions

Type `tuple`

`noisy_layers`
Noisy layer dimensions

Type `tuple`

`num_atoms`
Number of atoms used to discretise the Categorical DQN value distribution

Type int

forward(*state: numpy.ndarray*) → numpy.ndarray
Defines the computation performed at every call.

Parameters **state** (*Tensor*) – Input to value function

class genrl.core.values.**CnnValue**(**args*, ***kwargs*)
Bases: `genrl.core.values.MlpValue`

CNN Value Function class

param framestack Number of previous frames to stack together

param action_dim Action dimension of environment

param val_type Specifies type of value function: (

"V" for V(s), "Qs" for Q(s), "Qsa" for Q(s,a))

param fc_layers Sizes of hidden layers

type framestack int

type action_dim int

type val_type string

type fc_layers tuple or list

forward(*state: numpy.ndarray*) → numpy.ndarray
Defines the computation performed at every call.

Parameters **state** (*Tensor*) – Input to value function

class genrl.core.values.**MlpCategoricalValue**(**args*, ***kwargs*)
Bases: `genrl.core.values.MlpNoisyValue`

Class for Categorical DQN's MLP Q-Value function

state_dim
Observation space dimensions

Type int

action_dim
Action space dimensions

Type int

fc_layers
Fully connected layer dimensions

Type `tuple`

noisy_layers
Noisy layer dimensions

Type `tuple`

num_atoms
Number of atoms used to discretise the Categorical DQN value distribution

Type int

forward(*state: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Parameters **state** (*Tensor*) – Input to value function

class genrl.core.values.**MlpDuelingValue**(**args*, ***kwargs*)
Bases: `genrl.core.values.MlpValue`

Class for Dueling DQN's MLP Q-Value function

state_dim
Observation space dimensions

Type int

action_dim
Action space dimensions

Type int

hidden
Hidden layer dimensions

Type `tuple`

forward(*state: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Parameters **state** (*Tensor*) – Input to value function

class genrl.core.values.**MlpNoisyValue**(**args*, *noisy_layers: Tuple = (128, 512)*, ***kwargs*)
Bases: `genrl.core.values.MlpValue`

reset_noise() → None
Resets noise for any Noisy layers in Value function

class genrl.core.values.**MlpValue**(*state_dim: int*, *action_dim: int = None*, *val_type: str = 'V'*, *fc_layers: Tuple = (32, 32)*, ***kwargs*)
Bases: `genrl.core.base.BaseValue`

MLP Value Function class

param state_dim State dimensions of environment

param action_dim Action dimensions of environment

param val_type Specifies type of value function: (

"V" for V(s), "Qs" for Q(s), "Qsa" for Q(s,a))

param hidden Sizes of hidden layers

type state_dim int

type action_dim int

type val_type string

type hidden tuple or list

genrl.core.values.**get_value_from_name**(*name_: str*) → Union[Type[genrl.core.values.MlpValue], Type[genrl.core.values.CnnValue]]
Gets the value function given the name of the value function

Parameters **name** (*string*) – Name of the value function needed

Returns Value function

2.7 Utilities

2.7.1 Logger

class `genrl.utils.logger.`**`CSVLogger`**(*logdir: str*)

Bases: `object`

CSV Logging class

Parameters **`logdir`** (*`string`*) – Directory to save log at

`close`() → None

Close the logger

`write`(*kvs: Dict[str, Any], log_key*) → None

Add entry to logger

Parameters **`kvs`** (*`dict`*) – Entries to be logged

class `genrl.utils.logger.`**`HumanOutputFormat`**(*logdir: str*)

Bases: `object`

Output from a log file in a human readable format

Parameters **`logdir`** (*`string`*) – Directory at which log is present

`close`() → None

`max_key_len`(*kvs: Dict[str, Any]*) → None

Finds max key length

Parameters **`kvs`** (*`dict`*) – Entries to be logged

`round`(*num: float*) → float

Returns a rounded float value depending on self.maxlen

Parameters **`num`** (*`float`*) – Value to round

`write`(*kvs: Dict[str, Any], log_key*) → None

Log the entry out in human readable format

Parameters **`kvs`** (*`dict`*) – Entries to be logged

`write_to_file`(*kvs: Dict[str, Any], file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → None

Log the entry out in human readable format

Parameters

- **`kvs`** (*`dict`*) – Entries to be logged
- **`file`** (*`io.TextIOWrapper`*) – Name of file to write logs to

class `genrl.utils.logger.`**`Logger`**(*logdir: str = None, formats: List[str] = ['csv']*)

Bases: `object`

Logger class to log important information

Parameters

- **`logdir`** (*`string`*) – Directory to save log at
- **`formats`** (*`list`*) – Formatting of each log ['csv', 'stdout', 'tensorboard']

`close`() → None

Close the logger

formats
Return save format(s)

logdir
Return log directory

write(*kvs: Dict[str, Any], log_key: str = 'timestep'*) → None
Add entry to logger

Parameters

- **kvs** (*dict*) – Entry to be logged
- **log_key** (*str*) – Key plotted on log_key

class genrl.utils.logger.**TensorboardLogger**(*logdir: str*)
Bases: object

Tensorboard Logging class

Parameters **logdir** (*string*) – Directory to save log at

close() → None
Close the logger

write(*kvs: Dict[str, Any], log_key: str = 'timestep'*) → None
Add entry to logger

Parameters

- **kvs** (*dict*) – Entries to be logged
- **log_key** (*str*) – Key plotted on x_axis

genrl.utils.logger.**get_logger_by_name**(*name: str*)
Gets the logger given the type of logger

Parameters **name** (*string*) – Name of the value function needed

Returns Logger

2.7.2 Utilities

genrl.utils.utils.**cnn**(*channels: Tuple = (4, 16, 32), kernel_sizes: Tuple = (8, 4), strides: Tuple = (4, 2), **kwargs*) → Tuple

(Generates a CNN model given input dimensions, channels, kernel_sizes and

strides)

param channels Input output channels before and after each convolution

param kernel_sizes Kernel sizes for each convolution

param strides Strides for each convolution

param in_size Input dimensions (assuming square input)

type channels tuple

type kernel_sizes tuple

type strides tuple

type in_size int

returns (Convolutional Neural Network with convolutional layers and activation layers)

`genrl.utils.utils.`**`get_env_properties`**(*env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv], network: Union[str, Any] = 'mlp'*) → Tuple[int]

Finds important properties of environment

param env Environment that the agent is interacting with

type env Gym Environment

param network Type of network architecture, eg. "mlp", "cnn"

type network str

returns (State space dimensions, Action space dimensions,

discreteness of action space and action limit (highest action value)

rtype int, float, . . . ; int, float, . . . ; bool; int, float, . . .

`genrl.utils.utils.`**`get_model`**(*type_: str*, *name_: str*) → Union

Utility to get the class of required function

param type_ "ac" for Actor Critic, "v" for Value, "p" for Policy

param name_ Name of the specific structure of model. (

Eg. "mlp" or "cnn")

type type_ string

returns Required class. Eg. MlpActorCritic

`genrl.utils.utils.`**`mlp`**(*sizes: Tuple*, *activation: str = 'relu'*, *sac: bool = False*)

Generates an MLP model given sizes of each layer

param sizes Sizes of hidden layers

param sac True if Soft Actor Critic is being used, else False

type sizes tuple or list

type sac bool

returns (Neural Network with fully-connected linear layers and

activation layers)

`genrl.utils.utils.`**`noisy_mlp`**(*fc_layers: List[int], noisy_layers: List[int], activation='relu'*)

Noisy MLP generating helper function

Parameters

- **`fc_layers`** (`list` of `int`) – List of fully connected layers
- **`noisy_layers`** (`list` of `int`) – :ist of noisy layers
- **`activation`** (*`str`*) – Activation function to be used. ["tanh", "relu"]

Returns Noisy MLP model

`genrl.utils.utils.`**`safe_mean`**(*log: Union[torch.Tensor, List[int]]*)

Returns 0 if there are no elements in logs `genrl.utils.utils.`**`set_seeds`**(*seed: int*, *env: Union[gym.core.Env, genrl.environments.vec_env.vector_envs.VecEnv] = None*) → None

Sets seeds for reproducibility

Parameters

- **`seed`** (*`int`*) – Seed Value
- **`env`** (*`Gym Environment`*) – Optionally pass gym environment to set its seed

2.7.3 Models

`class` `genrl.utils.models.`**`TabularModel`**(*s_dim: int*, *a_dim: int*)

Bases: `object`

Sample-based tabular model class for deterministic, discrete environments

Parameters

- **`s_dim`** (*`int`*) – environment state dimension
- **`a_dim`** (*`int`*) – environment action dimension

`add`(*state: numpy.ndarray*, *action: numpy.ndarray*, *reward: float*, *next_state: numpy.ndarray*) → None

add transition to model :param state: state :param action: action :param reward: reward :param next_state: next state :type state: float array :type action: int :type reward: int :type next_state: float array

`is_empty`() → bool

Check if the model has been updated or not

Returns True if model not updated yet

Return type bool

`sample`() → Tuple

sample state action pair from model

Returns state and action

Return type int, float, .. ; int, float, ..

`step`(*state: numpy.ndarray*, *action: numpy.ndarray*) → Tuple

return consequence of action at state

Returns reward and next state

Return type int; int, float, ..

`genrl.utils.models.`**`get_model_from_name`**(*name_: str*)

get model object from name

Parameters **`name`** (*`str`*) – name of the model [‘tabular’]

Returns the model

2.8 Trainers

2.8.1 On-Policy Trainer

On Policy Trainer Class

Trainer class for all the On Policy Agents: A2C, PPO1 and VPG

`genrl.trainers.OnPolicyTrainer.`**`agent`**
Agent algorithm object

Type object

`genrl.trainers.OnPolicyTrainer.`**`env`**
Environment

Type object

`genrl.trainers.OnPolicyTrainer.`**`log_mode`**
List of different kinds of logging. Supported: ["csv", "stdout", "tensorboard"]

Type `list` of str

`genrl.trainers.OnPolicyTrainer.`**`log_key`**
Key plotted on x_axis. Supported: ["timestep", "episode"]

Type str

`genrl.trainers.OnPolicyTrainer.`**`log_interval`**
Timesteps between successive logging of parameters onto the console

Type int

`genrl.trainers.OnPolicyTrainer.`**`logdir`**
Directory where log files should be saved.

Type str

`genrl.trainers.OnPolicyTrainer.`**`epochs`**
Total number of epochs to train for

Type int

`genrl.trainers.OnPolicyTrainer.`**`max_timesteps`**
Maximum limit of timesteps to train for

Type int

`genrl.trainers.OnPolicyTrainer.`**`off_policy`**
True if the agent is an off policy agent, False if it is on policy

Type bool

`genrl.trainers.OnPolicyTrainer.`**`save_interval`**
Timesteps between successive saves of the agent's important hyperparameters

Type int

`genrl.trainers.OnPolicyTrainer.`**`save_model`**
Directory where the checkpoints of agent parameters should be saved

Type str

`genrl.trainers.OnPolicyTrainer.`**`run_num`**
A run number allotted to the save of parameters

Type int

`genrl.trainers.OnPolicyTrainer.`**`load_model`**
File to load saved parameter checkpoint from

Type str

`genrl.trainers.OnPolicyTrainer.`**`render`**
True if environment is to be rendered during training, else False

Type bool

`genrl.trainers.OnPolicyTrainer.`**`evaluate_episodes`**
Number of episodes to evaluate for

Type int

`genrl.trainers.OnPolicyTrainer.`**`seed`**
Set seed for reproducibility

Type int

`genrl.trainers.OnPolicyTrainer.`**`n_envs`**
Number of environments

2.8.2 Off-Policy Trainer

Off Policy Trainer Class

Trainer class for all the Off Policy Agents: DQN (all variants), DDPG, TD3 and SAC

`genrl.trainers.OffPolicyTrainer.`**`agent`**
Agent algorithm object

Type object

`genrl.trainers.OffPolicyTrainer.`**`env`**
Environment

Type object

`genrl.trainers.OffPolicyTrainer.`**`buffer`**
Replay Buffer object

Type object

`genrl.trainers.OffPolicyTrainer.`**`max_ep_len`**
Maximum Episode length for training

Type int

`genrl.trainers.OffPolicyTrainer.`**`max_timesteps`**
Maximum limit of timesteps to train for

Type int

`genrl.trainers.OffPolicyTrainer.`**`warmup_steps`**
Number of warmup steps. (random actions are taken to add randomness to training)

Type int

`genrl.trainers.OffPolicyTrainer.`**`start_update`**
Timesteps after which the agent networks should start updating

Type int

`genrl.trainers.OffPolicyTrainer.`**`update_interval`**
Timesteps between target network updates

Type int

`genrl.trainers.OffPolicyTrainer.`**`log_mode`**
List of different kinds of logging. Supported: ["csv", "stdout", "tensorboard"]

Type `list` of str

`genrl.trainers.OffPolicyTrainer.`**`log_key`**
Key plotted on x_axis. Supported: ["timestep", "episode"]

Type str

`genrl.trainers.OffPolicyTrainer.`**`log_interval`**
Timesteps between successive logging of parameters onto the console

Type int

`genrl.trainers.OffPolicyTrainer.`**`logdir`**
Directory where log files should be saved.

Type str

`genrl.trainers.OffPolicyTrainer.`**`epochs`**
Total number of epochs to train for

Type int

`genrl.trainers.OffPolicyTrainer.`**`off_policy`**
True if the agent is an off policy agent, False if it is on policy

Type bool

`genrl.trainers.OffPolicyTrainer.`**`save_interval`**
Timesteps between successive saves of the agent's important hyperparameters

Type int

`genrl.trainers.OffPolicyTrainer.`**`save_model`**
Directory where the checkpoints of agent parameters should be saved

Type str

`genrl.trainers.OffPolicyTrainer.`**`run_num`**
A run number allotted to the save of parameters

Type int

`genrl.trainers.OffPolicyTrainer.`**`load_model`**
File to load saved parameter checkpoint from

Type str

`genrl.trainers.OffPolicyTrainer.`**`render`**
True if environment is to be rendered during training, else False

Type bool

`genrl.trainers.OffPolicyTrainer.`**`evaluate_episodes`**
Number of episodes to evaluate for

Type int

`genrl.trainers.OffPolicyTrainer.`**`seed`**
Set seed for reproducibility

Type int

`genrl.trainers.OffPolicyTrainer.`**`n_envs`**
Number of environments

2.8.3 Classical Trainer

Global trainer class for classical RL algorithms

param agent Algorithm object to train

param env standard gym environment to train on

param mode mode of value function update [‘learn’, ‘plan’, ‘dyna’]

param model model to use for planning [‘tabular’]

param n_episodes number of training episodes

param plan_n_steps number of planning step per environment interaction

param start_steps number of initial exploration timesteps

param seed seed for random number generator

param render render gym environment

type agent object

type env Gym environment

type mode str

type model str

type n_episodes int

type plan_n_steps int

type start_steps int

type seed int

type render bool

2.8.4 Deep Contextual Bandit Trainer

Bandit Trainer Class

param agent Agent to train.

type agent genrl.deep.bandit.dcb_agents.DCBAgent

param bandit Bandit to train agent on.

type bandit genrl.deep.bandit.data_bandits.DataBasedBandit

param logdir Path to directory to store logs in.

type logdir str

param log_mode List of modes for logging.

type log_mode List[str]

2.8.5 Multi Armed Bandit Trainer

Bandit Trainer Class

param agent Agent to train.

type agent genrl.deep.bandit.dcb_agents.DCBAgent

param bandit Bandit to train agent on.

type bandit genrl.deep.bandit.data_bandits.DataBasedBandit

param logdir Path to directory to store logs in.

type logdir str

param log_mode List of modes for logging.

type log_mode List[str]

2.8.6 Base Trainer

Base Trainer Class

To be inherited specific use-cases

`genrl.trainers.Trainer.`**`agent`**
Agent algorithm object

Type object

`genrl.trainers.Trainer.`**`env`**
Environment

Type object

`genrl.trainers.Trainer.`**`log_mode`**
List of different kinds of logging. Supported: ["csv", "stdout", "tensorboard"]

Type `list` of str

`genrl.trainers.Trainer.`**`log_key`**
Key plotted on x_axis. Supported: ["timestep", "episode"]

Type str

`genrl.trainers.Trainer.`**`log_interval`**
Timesteps between successive logging of parameters onto the console

Type int

`genrl.trainers.Trainer.`**`logdir`**
Directory where log files should be saved.

Type str

`genrl.trainers.Trainer.`**`epochs`**
Total number of epochs to train for

Type int

`genrl.trainers.Trainer.`**`max_timesteps`**
Maximum limit of timesteps to train for

Type int

`genrl.trainers.Trainer.`**`off_policy`**
True if the agent is an off policy agent, False if it is on policy

Type bool

genrl.trainers.Trainer.**save_interval**
Timesteps between successive saves of the agent's important hyperparameters

Type int

genrl.trainers.Trainer.**save_model**
Directory where the checkpoints of agent parameters should be saved

Type str

genrl.trainers.Trainer.**run_num**
A run number allotted to the save of parameters

Type int

genrl.trainers.Trainer.**load_weights**
Weights file

Type str

genrl.trainers.Trainer.**load_hyperparams**
File to load hyperparameters

Type str

genrl.trainers.Trainer.**render**
True if environment is to be rendered during training, else False

Type bool

genrl.trainers.Trainer.**evaluate_episodes**
Number of episodes to evaluate for

Type int

genrl.trainers.Trainer.**seed**
Set seed for reproducibility

Type int

genrl.trainers.Trainer.**n_envs**
Number of environments

2.9 Common

2.9.1 Classical Common

genrl.classical.common.models

genrl.classical.common.trainer

genrl.classical.common.values

2.9.2 Bandit Common

genrl.bandit.core

genrl.bandit.trainer

genrl.bandit.agents.cb_agents.common.base_model

class `genrl.agents.bandits.contextual.common.base_model.`**`Model`**(*layer*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`, `abc.ABC`

Bayesian Neural Network used in Deep Contextual Bandit Models.

Parameters

- **`context_dim`** (*`int`*) – Length of context vector.
- **`hidden_dims`** (*`List[int], optional`*) – Dimensions of hidden layers of network.
- **`n_actions`** (*`int`*) – Number of actions that can be selected. Taken as length of output vector for network to predict.
- **`init_lr`** (*`float, optional`*) – Initial learning rate.
- **`max_grad_norm`** (*`float, optional`*) – Maximum norm of gradients for gradient clipping.
- **`lr_decay`** (*`float, optional`*) – Decay rate for learning rate.
- **`lr_reset`** (*`bool, optional`*) – Whether to reset learning rate ever train interval. Defaults to False.
- **`dropout_p`** (*`Optional[float], optional`*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **`noise_std`** (*`float`*) – Standard deviation of noise used in the network. Defaults to 0.1

`use_dropout`
Indicated whether or not dropout should be used in forward pass.

Type int

`forward`(*context: torch.Tensor*, ***kwargs*) → Dict[str, torch.Tensor]
Computes forward pass through the network.

Parameters **`context`** (*`torch.Tensor`*) – The context vector to perform forward pass on.

Returns Dictionary of outputs

Return type Dict[str, torch.Tensor]

`train_model`(*db: genrl.agents.bandits.contextual.common.transition.TransitionDB*, *epochs: int*, *batch_size: int*)
Trains the network on a given database for given epochs and batch_size.

Parameters

- **`db`** (`TransitionDB`) – The database of transitions to train on.
- **`epochs`** (*`int`*) – Number of gradient steps to take.
- **`batch_size`** (*`int`*) – The size of each batch to perform gradient descent on.

genrl.bandit.agents.cb_agents.common.bayesian

class genrl.agents.bandits.contextual.common.bayesian.**BayesianLinear**(*in_features: int*, *out_features: int*, *bias: bool = True*)

Bases: `torch.nn.modules.module.Module`

Linear Layer for Bayesian Neural Networks.

Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool, optional*) – Whether to use an additive bias. Defaults to True.

forward(*x: torch.Tensor*, *kl: bool = True*, *frozen: bool = False*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Apply linear transormation to input.

The weight and bias is sampled for each forward pass from a normal distribution. The KL divergence of the sampled weigth and bias can also be computed if specified.

Parameters

- **x** (*torch.Tensor*) – Input to be transformed
- **kl** (*bool, optional*) – Whether to compute the KL divergence. Defaults to True.
- **frozen** (*bool, optional*) – Whether to freeze current parameters. Defaults to False.

Returns

The transformed input and optionally the computed KL divergence value.

Return type Tuple[torch.Tensor, Optional[torch.Tensor]]

reset_parameters() → None

Resets weight and bias parameters of the layer.

class genrl.agents.bandits.contextual.common.bayesian.**BayesianNNBanditModel**(***kwargs*)

Bases: *genrl.agents.bandits.contextual.common.base_model.Model*

Bayesian Neural Network used in Deep Contextual Bandit Models.

Parameters

- **context_dim** (*int*) – Length of context vector.
- **hidden_dims** (*List[int], optional*) – Dimensions of hidden layers of network.
- **n_actions** (*int*) – Number of actions that can be selected. Taken as length of output vector for network to predict.
- **init_lr** (*float, optional*) – Initial learning rate.
- **max_grad_norm** (*float, optional*) – Maximum norm of gradients for gradient clipping.
- **lr_decay** (*float, optional*) – Decay rate for learning rate.

- **lr_reset** (*bool, optional*) – Whether to reset learning rate ever train interval. Defaults to False.
- **dropout_p** (*Optional[float], optional*) – Probability for dropout. Defaults to None which implies dropout is not to be used.
- **noise_std** (*float*) – Standard deviation of noise used in the network. Defaults to 0.1

use_dropout
Indicated whether or not dropout should be used in forward pass.

Type int

forward(*context: torch.Tensor, kl: bool = True*) → Dict[str, torch.Tensor]
Computes forward pass through the network.

Parameters **context** (*torch.Tensor*) – The context vector to perform forward pass on.

Returns Dictionary of outputs

Return type Dict[str, torch.Tensor]

genrl.bandit.agents.cb_agents.common.neural

class genrl.agents.bandits.contextual.common.neural.**NeuralBanditModel**(***kwargs*)
Bases: *genrl.agents.bandits.contextual.common.base_model.Model*

Neural Network used in Deep Contextual Bandit Models.

Parameters

- **context_dim** (*int*) – Length of context vector.
- **hidden_dims** (*List[int], optional*) – Dimensions of hidden layers of network.
- **n_actions** (*int*) – Number of actions that can be selected. Taken as length of output vector for network to predict.
- **init_lr** (*float, optional*) – Initial learning rate.
- **max_grad_norm** (*float, optional*) – Maximum norm of gradients for gradient clipping.
- **lr_decay** (*float, optional*) – Decay rate for learning rate.
- **lr_reset** (*bool, optional*) – Whether to reset learning rate ever train interval. Defaults to False.
- **dropout_p** (*Optional[float], optional*) – Probability for dropout. Defaults to None which implies dropout is not to be used.

use_dropout
Indicated whether or not dropout should be used in forward pass.

Type bool

forward(*context: torch.Tensor*) → Dict[str, torch.Tensor]
Computes forward pass through the network.

Parameters **context** (*torch.Tensor*) – The context vector to perform forward pass on.

Returns Dictionary of outputs

Return type Dict[str, torch.Tensor]

genrl.bandit.agents.cb_agents.common.transition

class genrl.agents.bandits.contextual.common.transition.**TransitionDB**(*device: Union[str, torch.device] = 'cpu'*)

Bases: `object`

Database for storing (context, action, reward) transitions.

Parameters **device** (*str*) – Device to use for tensor operations. "cpu" for cpu or "cuda" for cuda. Defaults to "cpu".

db

Dictionary containing list of transitions.

Type dict

db_size

Number of transitions stored in database.

Type int

device

Device to use for tensor operations.

Type torch.device

add(*context: torch.Tensor*, *action: int*, *reward: int*)

Add (context, action, reward) transition to database

Parameters

- **context** (*torch.Tensor*) – Context recieved
- **action** (*int*) – Action taken
- **reward** (*int*) – Reward recieved

get_data(*batch_size: Optional[int] = None*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get a batch of transition from database

Parameters **batch_size** (*Union[int, None], optional*) – Size of batch required. Defaults to None which implies all transitions in the database are to be included in batch.

Returns

Tuple of stacked contexts, actions, rewards tensors.

Return type Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

get_data_for_action(*action: int*, *batch_size: Optional[int] = None*) → Tuple[torch.Tensor, torch.Tensor]

Get a batch of transition from database for a given action.

Parameters

- **action** (*int*) – The action to sample transitions for.
- **batch_size** (*Union[int, None], optional*) – Size of batch required. Defaults to None which implies all transitions in the database are to be included in batch.

Returns

Tuple of stacked contexts and rewards tensors.

Return type Tuple[torch.Tensor, torch.Tensor]

Python Module Index

Index