RL Anonymity (with Python)

Release v0.0.10-alpha

Alexandros Giavaras

Jun 04, 2022

CONTENTS

An experimental effort to use reinforcement learning techniques for data anonymization. The project repository is at RL anonymity (with Python).

CHAPTER

ONE

CONTENTS

1.1 Conceptual overview

The term data anonymization refers to techiniques that can be applied on a given dataset, D, such that it makes it difficult for a third party to identify or infer the existence of specific individuals in D. Anonymization techniques, typically result into some sort of distortion of the original dataset. This means that in order to maintain some utility of the transformed dataset, the transofrmations applied should be constrained in some sense. In the end, it can be argued, that data anonymization is an optimization problem meaning striking the right balance between data utility and privacy.

Reinforcement learning is a learning framework based on accumulated experience. In this paradigm, an agent is learning by iteracting with an environment without (to a large extent) any supervision. The following image describes, schematically, the reinforcement learning framework .

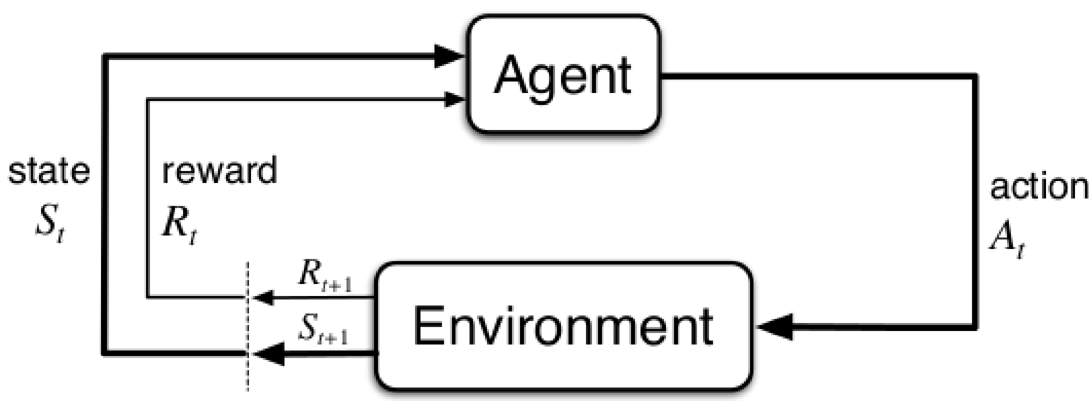


Fig. 1: Reinforcement learning paradigm.

The agent chooses an action, $A_t \in \mathbb{A}$, to perform out of predefined set of actions $\mathbb{A}$. The chosen action is executed by the environment instance and returns to the agent a reward signal, R_{t+1}, as well as the new state, S_{t+1}, that the enviroment is in. The overall goal of the agent is to maximize the expected total reward i.e.

$$maxE\left[R\right]$$

The framework has successfully been used to many recent advances in control, robotics, games and elsewhere.

In this work we are intersted in applying reinforcment learning techniques, in order to train agents to optimally anonymize a given data set. In particular, we want to consider the following two scenarios

- A tabular data set is to be publicly released
- A data set is behind a restrictive API that allows users to perform certain queries on the hidden data set.

For the first scenario, let's assume that we have in our disposal two numbers $DIST_{min}$ and $DIST_{max}$. The former indicates the minimum total data set distortion that it should be applied in order to satisfy some minimum safety criteria. The latter indicates the maximum total data set distortion that it should be applied in order to satisfy some utility criteria. Note that the same idea can be applied to enforce constraints on how much a column should be distorted. Furtheremore, let's assume the most common transformations applied for data anonymization

- Generalization
- Suppresion
- Permutation
- Pertubation
- Anatomization

We can conceive the above transformations as our action set $\mathbb{A}$. We can now cast the data anonymity problem into a form suitable for reinforcement learning. Specifically, our goal, and the agent's goal in that matter, is to obtain a policy pi of transformations such that by following pi, the data set total distortion will be into the interval $[DIST_{min}, DIST_{max}]$. This is done by choosing actions/transformations from $\mathbb{A}$. This is shown schematically in the figure below

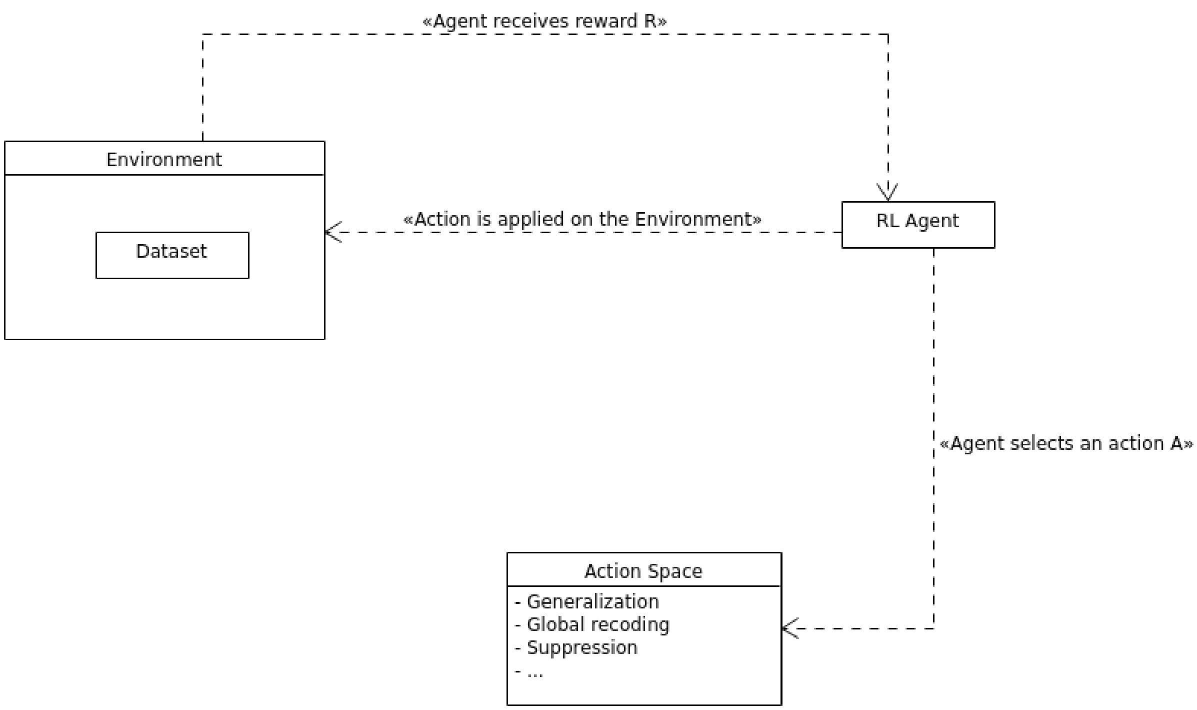


Fig. 2: Data anonymization using reinforcement learning.

Thus the environment is our case is an entity that encapsulates the original data set and controls the actions applied on it as well as the reward signal R_{t+1} and the next state S_{t+1} to be presented to the agent.

Nevertheless, there are some caveats that we need to take into account. We summarize these below.

First, we need a reward policy. The way we assign rewards implicitly specifies the degree of supervision we allow. For instance we could allow for a reward to be assigned every time a transformation is applied. This strategy allows for faster learning but it leaves little room for the agent to come up with novel strategies. In contrast, returning a reward at the end of the episode, although it increases the training time, it allows the agent to explore novel strategies. Related to the reward assignement is also the follwing issue. We need to reward the agent in a way that it is convinced that it should explore transformations. This is important as we don't want to the agent to simply exploit around the zero distortion point. The second thing we need to take into account is that the metric we use to measure the data set distortion plays an important role. Thirdly, we need to hold into memory two copies of the data set. One copy that no distortion is applied and one copy that we distort somehow during an episode. We need this setting so that we are able to compute the column distortions. Fourthly, we need to establish the episode termination criteria i.e. when do we consider that an episode is complete. Finally, as we assume that a data set may contain strings, floating point numbers as well as integers, then computed distortions are normalized. This is needed in order to avoid having large column distortions, e.g. consider a salary column being distorted, and also being able to sum all the column distortions in a meanigful way.

1.2 Installation

The following packages are required:

- NumPy
- Sphinx
- Python Pandas
- PyTorch
- Coverage.py

You can install there as usual with `pip`.

```
pip install -r requirements.txt
```

Installation of the package is done via `setuptools`

```
python setup.py
```

1.2.1 Run tests

The is a series of tests to verify the implementation. You can executed these by running the script `execute_tests_with_coverage.sh`.

1.2.2 Generate documentation

You will need Sphinx in order to generate the API documentation. Assuming that Sphinx is already installed on your machine execute the following commands (see also Sphinx tutorial).

```
sphinx-quickstart docs
sphinx-build -b html docs/source/ docs/build/html
```

1.3 Examples

Some examples can be found below

1.3.1 Q-learning on a three columns dataset

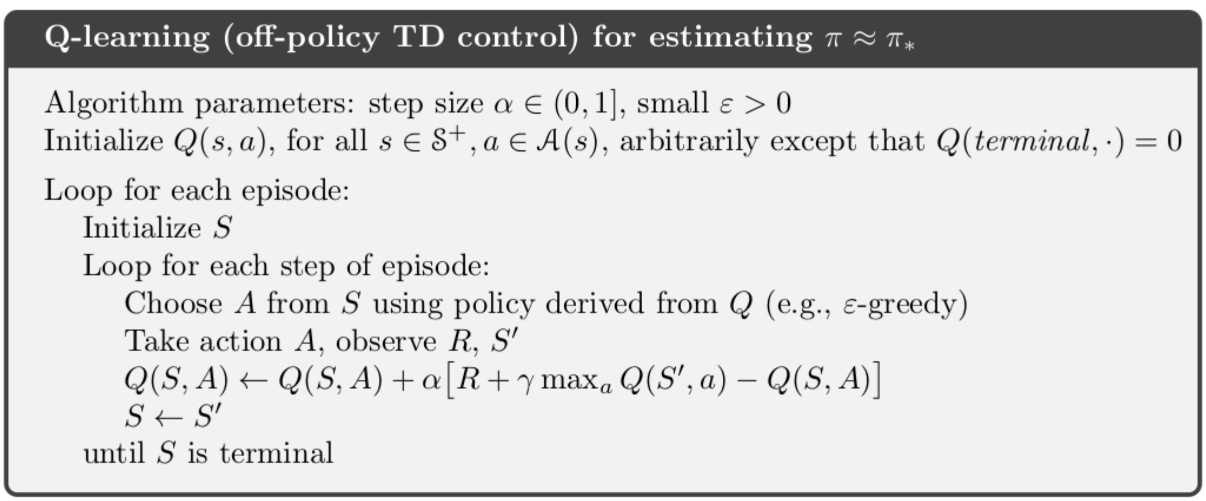
Overview

In this example, we use a tabular Q-learning algorithm to anonymize a data set with three columns. In particular, we discretize the total dataset distortion into bins. Another approach could be to discretize the distortion of each column into bins and create tuples of indeces representing a state. We follow the latter approach in another example.

Q-learning

Q-learning is one of the early breakthroughs in the field of reinforcement learning [1]. It was first introduced in [2]. Q-learning is an off-policy algorithm where the learned state-action value function $Q(s, \alpha)$ directly approximates the optimal state-action value function Q^*. This is done independently of the policy π being followed [1].

The Q-learning algorithm is an iterative algorithm where we iterate over a number of episodes. At each episode the algorithm steps over the environment for a user-specified number steps it executes an action which results in a new state. This is shown collectively in the image below



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Fig. 3: Q-learning algorithm. Image from [1].

At each episode step, the algorithm updates $Q(s, \alpha)$ according to:

$$Q(s_t, \alpha_t) = Q(s_t, \alpha_t) + \alpha \left[r_{t+1} + \gamma max_{\alpha} Q(s_{t+1}, \alpha) - Q(s_t, \alpha_t)\right]$$

where α is a user-defined learning factor and γ is the user-defined discount factor. The algorithm requires the following user-defined input

- Number of episodes
- Number of steps per episode
- γ

- α
- An external policy function to decide which action to take (e.g. ϵ-greedy)

Although with Q-learning $Q(s, \alpha)$ directly approximates Q^* independently of the policy π being followed, the policy still has an effect in that it determines which state-action pairs and visited updated. However, for correct convergence all that is required is that all pairs continue to be updated [1]. In fact, any method guaranteed to find optimal behavior in the general case must require it [1].

The algorithm above, stores the expected value estimate for each state-action pair in a table. This means we cannot use it when we have continuous states or actions, which would lead to an array of infinite length. Given that the total dataset distortion is assumed to be in the range $[0, 1]$ of the real numbers; where the edge points mean no distortion and full distortion of the data set/column respectively. We discretize this range into bins and for each entailed value of the distortion we use the corresponding bin as a state index. Alternatively, we could discretize the distortion of each column into bins and create tuples of indeces representing a state.

We preprocess the data set by normalizing the numeric columns. We will use the cosine normalized distance to measure the distortion of columns with string data. Similarly, we use the following L_2-based norm for calculating the distortion of numeric columns

$$dist(\mathbf{v}_1, \mathbf{v}_2) = \sqrt{\frac{||\mathbf{v}_1 - \mathbf{v}_2||_{L_2}}{N}}$$

where N is the size of the vector. This way the resulting distance, due to the normalization of numeric columns, will be in the range $[0, 1]$.

Code

The necessary imports

```
import numpy as np
import random


from src.trainers.trainer import Trainer, TrainerConfig
from src.algorithms.q_learning import QLearning, QLearnConfig
from src.spaces.action_space import ActionSpace
from src.spaces.actions import ActionIdentity, ActionStringGeneralize,␣
↪ActionNumericBinGeneralize
from src.policies.epsilon_greedy_policy import EpsilonGreedyPolicy, EpsilonDecayOption
from src.utils.iteration_control import IterationControl
from src.examples.helpers.plot_utils import plot_running_avg
from src.datasets import ColumnType
from src.examples.helpers.load_three_columns_mock_dataset import load_discrete_env, \
    get_ethinicity_hierarchy, get_salary_bins, load_mock_subjects
from src.spaces.env_type import DiscreteEnvType
from src.utils import INFO
```

Next establish a set of configuration parameters

```
# configuration params
EPS = 1.0
EPSILON_DECAY_OPTION = EpsilonDecayOption.CONSTANT_RATE  # .INVERSE_STEP
EPSILON_DECAY_FACTOR = 0.01
GAMMA = 0.99
ALPHA = 0.1
```

(continues on next page)

(continued from previous page)

```
N_EPISODES = 1001
N_ITRS_PER_EPISODE = 30
N_STATES = 10
# fix the rewards. Assume that any average distortion in
# (0.3, 0.7) suits us
MAX_DISTORTION = 0.7
MIN_DISTORTION = 0.3
OUT_OF_MAX_BOUND_REWARD = -1.0
OUT_OF_MIN_BOUND_REWARD = -1.0
IN_BOUNDS_REWARD = 5.0
OUTPUT_MSG_FREQUENCY = 100
N_ROUNDS_BELOW_MIN_DISTORTION = 10
SAVE_DISTORTED_SETS_DIR = "q_learning_three_columns_results/distorted_set"
PUNISH_FACTOR = 2.0
```

The dirver code brings all the elements together

```
if __name__ == '__main__':

    # set the seed for random engine
    random.seed(42)

    # set the seed for random engine
    random.seed(42)

    column_types = {"ethnicity": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "salary": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "diagnosis": ColumnType.INSENSITIVE_ATTRIBUTE}

    action_space = ActionSpace(n=5)
    # all the columns that are SENSITIVE_ATTRIBUTE will be kept as they are
    # because currently we have no model
    # also INSENSITIVE_ATTRIBUTE will be kept as is
    action_space.add_many(ActionIdentity(column_name="salary"),
                          ActionIdentity(column_name="diagnosis"),
                          ActionIdentity(column_name="ethnicity"),
                          ActionStringGeneralize(column_name="ethnicity",
                                                 generalization_table=get_ethinicity_
↪hierarchy()),
                          ActionNumericBinGeneralize(column_name="salary",
                                                     generalization_table=get_salary_
↪bins(ds=load_mock_subjects(),
                                                                                 ␣
↪ n_states=N_STATES)))

    env = load_discrete_env(env_type=DiscreteEnvType.TOTAL_DISTORTION_STATE, n_states=N_
↪STATES,
                            action_space=action_space,
                            min_distortion=MIN_DISTORTION, max_distortion=MIN_DISTORTION,
                            total_min_distortion=MIN_DISTORTION, total_max_
↪distortion=MAX_DISTORTION,
                            punish_factor=PUNISH_FACTOR, column_types=column_types,
```

(continues on next page)

(continued from previous page)

```
                              save_distoreted_sets_dir=SAVE_DISTORTED_SETS_DIR,
                              use_identifying_column_dist_in_total_dist=False,
                              use_identifying_column_dist_factor=-100,
                              gamma=GAMMA,
                              in_bounds_reward=IN_BOUNDS_REWARD,
                              out_of_min_bound_reward=OUT_OF_MIN_BOUND_REWARD,
                              out_of_max_bound_reward=OUT_OF_MAX_BOUND_REWARD,
                              n_rounds_below_min_distortion=N_ROUNDS_BELOW_MIN_DISTORTION)

    # save the data before distortion so that we can
    # later load it on ARX
    env.save_current_dataset(episode_index=-1, save_index=False)

    # configuration for the Q-learner
    algo_config = QLearnConfig(gamma=GAMMA, alpha=ALPHA,
                               n_itrs_per_episode=N_ITRS_PER_EPISODE,
                               policy=EpsilonGreedyPolicy(eps=EPS, n_actions=env.n_
↪actions,
                                                          decay_op=EPSILON_DECAY_OPTION,
                                                          epsilon_decay_factor=EPSILON_
↪DECAY_FACTOR))

    agent = QLearning(algo_config=algo_config)

    trainer_config = TrainerConfig(n_episodes=N_EPISODES, output_msg_frequency=OUTPUT_
↪MSG_FREQUENCY)
    trainer = Trainer(env=env, agent=agent, configuration=trainer_config)
    trainer.train()

    # avg_rewards = trainer.avg_rewards()
    avg_rewards = trainer.total_rewards
    plot_running_avg(avg_rewards, steps=100,
                     xlabel="Episodes", ylabel="Reward",
                     title="Running reward average over 100 episodes")

    avg_episode_dist = np.array(trainer.total_distortions)
    print("{0} Max/Min distortion {1}/{2}".format(INFO, np.max(avg_episode_dist), np.
↪min(avg_episode_dist)))

    plot_running_avg(avg_episode_dist, steps=100,
                     xlabel="Episodes", ylabel="Distortion",
                     title="Running distortion average over 100 episodes")

    print("=============================================")
    print("{0} Generating distorted dataset".format(INFO))
    # Let's play
    env.reset()

    stop_criterion = IterationControl(n_itrs=10, min_dist=MIN_DISTORTION, max_dist=MAX_
↪DISTORTION)
    agent.play(env=env, stop_criterion=stop_criterion)
    env.save_current_dataset(episode_index=-2, save_index=False)
```

(continues on next page)

(continued from previous page)

```
    print("{0} Done....".format(INFO))
    print("=============================================")
```

Results

The following images show the performance of the learning process

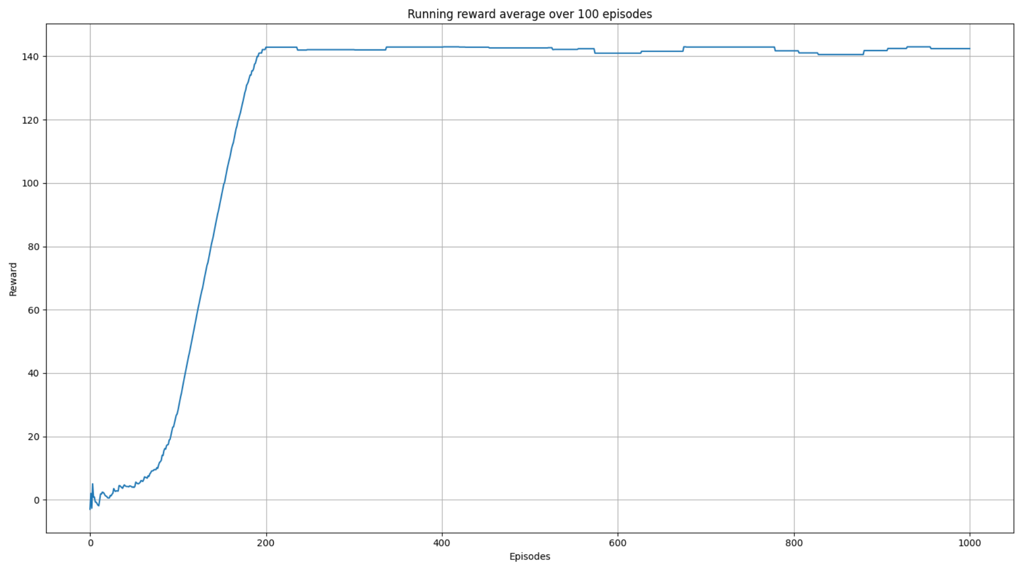


Fig. 4: Running average reward.

Although there is evidence of learning, it should be noted that this depends heavily on the applied transformations on the columns and the metrics used. So typically, some experimentation should be employed in order to determine the right options.

The following is snapshot of the distorted dataset produced by the agent

```
ethnicity,salary,diagnosis
British,0.3333333333333333,1
British,0.1111111111111111,0
British,0.5555555555555556,3
British,0.5555555555555556,3
British,0.1111111111111111,0
British,0.1111111111111111,1
British,0.1111111111111111,4
British,0.3333333333333333,3
British,0.1111111111111111,4
British,0.3333333333333333,0
Asian,0.1111111111111111,0
British,0.1111111111111111,0
British,0.1111111111111111,3
White,0.1111111111111111,0
```

(continues on next page)

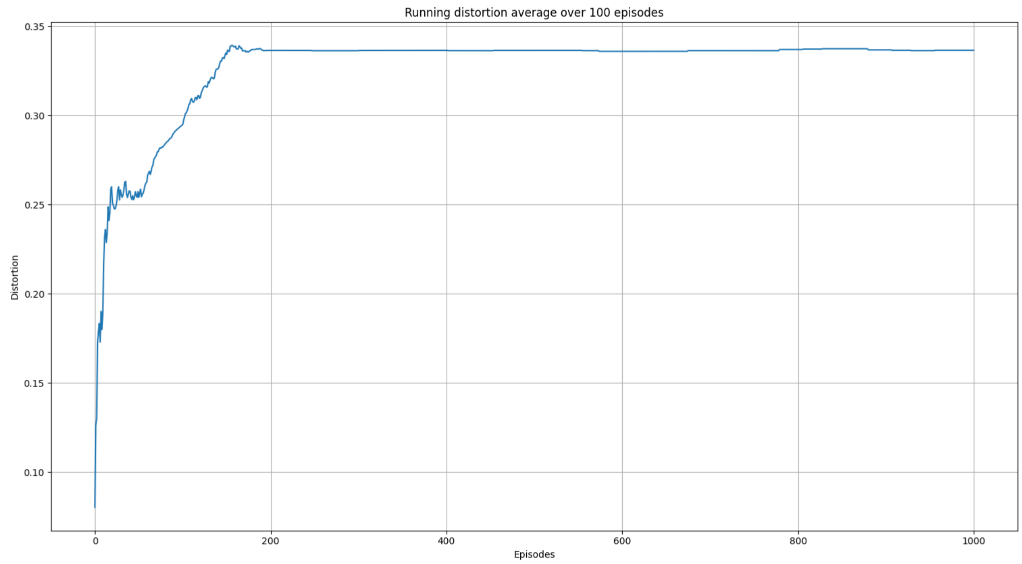


Fig. 5: Running average total distortion.

(continued from previous page)

```
British,0.1111111111111111,3
British,0.3333333333333333,4
Mixed,0.3333333333333333,4
British,0.7777777777777777,1
```

whilst the following is a snapshot of the distorted dataset by using ARX K-anonymity algorithm

```
NHSno,given_name,surname,gender,dob,ethnicity,education,salary,mutation_status,
↪preventative_treatment,diagnosis
*,*,*,*,*,White British,*,0.3333333333333333,*,*,1
*,*,*,*,*,White British,*,0.1111111111111111,*,*,0
*,*,*,*,*,White British,*,0.1111111111111111,*,*,1
*,*,*,*,*,White British,*,0.3333333333333333,*,*,3
*,*,*,*,*,White British,*,0.1111111111111111,*,*,4
*,*,*,*,*,White British,*,0.3333333333333333,*,*,0
*,*,*,*,*,Bangladeshi,*,0.1111111111111111,*,*,0
*,*,*,*,*,White British,*,0.1111111111111111,*,*,0
*,*,*,*,*,White other,*,0.1111111111111111,*,*,0
*,*,*,*,*,White British,*,0.3333333333333333,*,*,4
*,*,*,*,*,White British,*,0.7777777777777777,*,*,1
*,*,*,*,*,White British,*,0.1111111111111111,*,*,2
*,*,*,*,*,White British,*,0.1111111111111111,*,*,2
*,*,*,*,*,White other,*,0.1111111111111111,*,*,2
*,*,*,*,*,White British,*,0.5555555555555556,*,*,0
*,*,*,*,*,White British,*,0.5555555555555556,*,*,4
*,*,*,*,*,White British,*,0.5555555555555556,*,*,0
*,*,*,*,*,White British,*,0.3333333333333333,*,*,0
```

Note that the K-anonymity algorithm removes some rows during the anonymization process, so there is no one-to-one correspondence to the two outpus. Nonetheless, it shows qualitatively what the two algorithms produce.

References

1. Richard S. Sutton and Andrw G. Barto, Reinforcement Learning. An Introduction 2nd Edition, MIT Press.
2. C. J. C.

   D. Watkins, Learning from delayed rewards, King's College, Cambridge, Ph.D. thesis, 1989.

1.3.2 Q-learning algorithm on mock data set

Overview

In the previous example, we applied Q-learning on a dataset consisting of three columns. Moreover, we used a one dimensional state space; we discretized the range $[0, 1]$ into bins and used the resulting bin index as the state index. In this example, we will simply allow for more columns in the data set. Other than that, this example is the same as the previous one.

Code

The necessary imports

```
import random
import numpy as np

from src.examples.helpers.load_full_mock_dataset import load_discrete_env, get_
↪ethinicity_hierarchy, \
    get_gender_hierarchy, get_salary_bins, load_mock_subjects
from src.datasets import ColumnType
from src.spaces.env_type import DiscreteEnvType
from src.spaces.action_space import ActionSpace
from src.spaces.actions import ActionIdentity, ActionStringGeneralize,␣
↪ActionNumericBinGeneralize
from src.algorithms.q_learning import QLearnConfig, QLearning
from src.policies.epsilon_greedy_policy import EpsilonGreedyPolicy, EpsilonDecayOption
from src.trainers.trainer import Trainer, TrainerConfig
from src.examples.helpers.plot_utils import plot_running_avg
from src.utils import INFO
```

Next establish a set of configuration parameters

```
# configuration params
N_STATES = 10
GAMMA = 0.99
ALPHA = 0.1
PUNISH_FACTOR = 2.0
MAX_DISTORTION = 0.7
MIN_DISTORTION = 0.4
SAVE_DISTORTED_SETS_DIR = "/home/alex/qi3/drl_anonymity/src/examples/q_learning_all_cols_
↪results/distorted_set"
EPS = 1.0
```

(continues on next page)

(continued from previous page)

```
EPSILON_DECAY_OPTION = EpsilonDecayOption.CONSTANT_RATE  # .INVERSE_STEP
EPSILON_DECAY_FACTOR = 0.01
USE_IDENTIFYING_COLUMNS_DIST = True
IDENTIFY_COLUMN_DIST_FACTOR = 0.1
N_EPISODES = 1001
N_ITRS_PER_EPISODE = 30
OUT_OF_MAX_BOUND_REWARD = -1.0
OUT_OF_MIN_BOUND_REWARD = -1.0
IN_BOUNDS_REWARD = 5.0
OUTPUT_MSG_FREQUENCY = 100
N_ROUNDS_BELOW_MIN_DISTORTION = 10
```

The dirver code brings all the elements together

```
if __name__ == '__main__':

    # set the seed for random engine
    random.seed(42)

    # specify the column types. An identifying column
    # will me removed from the anonymized data set
    # An  INSENSITIVE_ATTRIBUTE remains intact.
    # A QUASI_IDENTIFYING_ATTRIBUTE is used in the anonymization
    # A SENSITIVE_ATTRIBUTE currently remains intact
    column_types = {"NHSno": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "given_name": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "surname": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "gender": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "dob": ColumnType.SENSITIVE_ATTRIBUTE,
                    "ethnicity": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "education": ColumnType.SENSITIVE_ATTRIBUTE,
                    "salary": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "mutation_status": ColumnType.SENSITIVE_ATTRIBUTE,
                    "preventative_treatment": ColumnType.SENSITIVE_ATTRIBUTE,
                    "diagnosis": ColumnType.INSENSITIVE_ATTRIBUTE}

    # define the action space
    action_space = ActionSpace(n=10)

    # all the columns that are SENSITIVE_ATTRIBUTE will be kept as they are
    # because currently we have no model
    # also INSENSITIVE_ATTRIBUTE will be kept as is
    # in order to declare this we use an ActionIdentity
    action_space.add_many(ActionIdentity(column_name="dob"),
                          ActionIdentity(column_name="education"),
                          ActionIdentity(column_name="salary"),
                          ActionIdentity(column_name="diagnosis"),
                          ActionIdentity(column_name="mutation_status"),
                          ActionIdentity(column_name="preventative_treatment"),
                          ActionIdentity(column_name="ethnicity"),
                          ActionStringGeneralize(column_name="ethnicity",
                                                 generalization_table=get_ethinicity_
↪hierarchy()),
```

(continues on next page)

(continued from previous page)

```
                              ActionStringGeneralize(column_name="gender",
                                                     generalization_table=get_gender_
↪hierarchy()),
                              ActionNumericBinGeneralize(column_name="salary",
                                                         generalization_table=get_salary_
↪bins(ds=load_mock_subjects(),
                                                                                          ␣
↪ n_states=N_STATES))
                              )
    action_space.shuffle()

    env = load_discrete_env(env_type=DiscreteEnvType.TOTAL_DISTORTION_STATE,
                            n_states=N_STATES,
                            min_distortion=MIN_DISTORTION, max_distortion=MAX_DISTORTION,
                            total_min_distortion=MIN_DISTORTION, total_max_
↪distortion=MAX_DISTORTION,
                            out_of_max_bound_reward=OUT_OF_MAX_BOUND_REWARD,
                            out_of_min_bound_reward=OUT_OF_MIN_BOUND_REWARD,
                            in_bounds_reward=IN_BOUNDS_REWARD,
                            punish_factor=PUNISH_FACTOR,
                            column_types=column_types,
                            action_space=action_space,
                            save_distoreted_sets_dir=SAVE_DISTORTED_SETS_DIR,
                            use_identifying_column_dist_in_total_dist=USE_IDENTIFYING_
↪COLUMNS_DIST,
                            use_identifying_column_dist_factor=IDENTIFY_COLUMN_DIST_
↪FACTOR,
                            gamma=GAMMA,
                            n_rounds_below_min_distortion=N_ROUNDS_BELOW_MIN_DISTORTION)

    agent_config = QLearnConfig(n_itrs_per_episode=N_ITRS_PER_EPISODE, gamma=GAMMA,
                                alpha=ALPHA,
                                policy=EpsilonGreedyPolicy(eps=EPS, n_actions=env.n_
↪actions,
                                                           decay_op=EPSILON_DECAY_OPTION,
                                                           epsilon_decay_factor=EPSILON_
↪DECAY_FACTOR))

    agent = QLearning(algo_config=agent_config)

    trainer_config = TrainerConfig(n_episodes=N_EPISODES, output_msg_frequency=OUTPUT_
↪MSG_FREQUENCY)
    trainer = Trainer(env=env, agent=agent, configuration=trainer_config)
    trainer.train()

    avg_rewards = trainer.total_rewards
    plot_running_avg(avg_rewards, steps=100,
                     xlabel="Episodes", ylabel="Reward",
                     title="Running reward average over 100 episodes")

    avg_episode_dist = np.array(trainer.total_distortions)
    print("{0} Max/Min distortion {1}/{2}".format(INFO, np.max(avg_episode_dist), np.
↪min(avg_episode_dist)))
```

(continues on next page)

(continued from previous page)

```
    plot_running_avg(avg_episode_dist, steps=100,
                     xlabel="Episodes", ylabel="Distortion",
                     title="Running distortion average over 100 episodes")
```

Results

The following images show the performance of the learning process

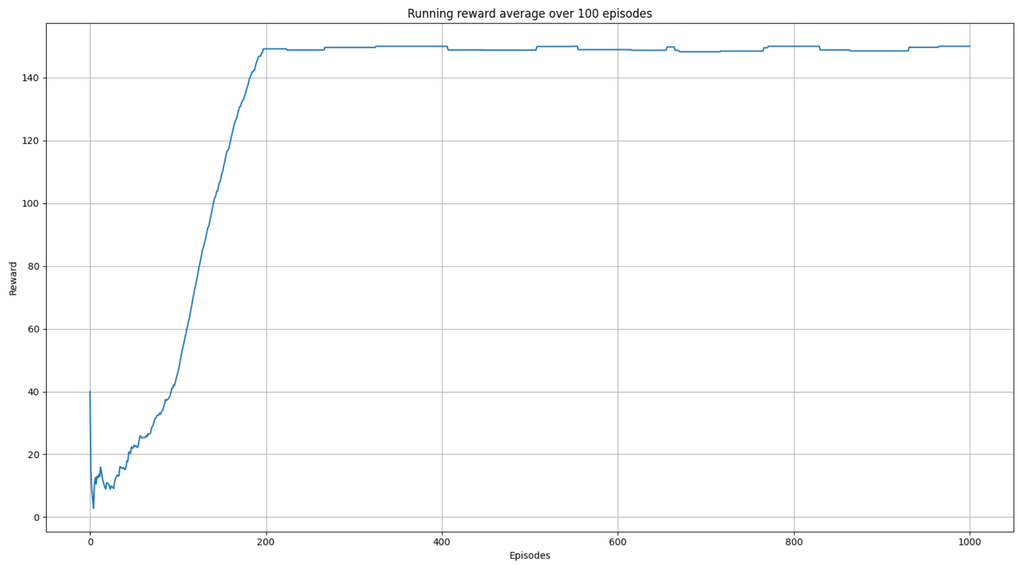


Fig. 6: Running average reward.

References

1. Richard S. Sutton and Andrw G. Barto, Reinforcement Learning. An Introduction 2nd Edition, MIT Press.

1.3.3 Semi-gradient SARSA algorithm on mock data set

Overview

In this example, we use the episodic semi-gradient SARSA algorithm to anonymize a data set.

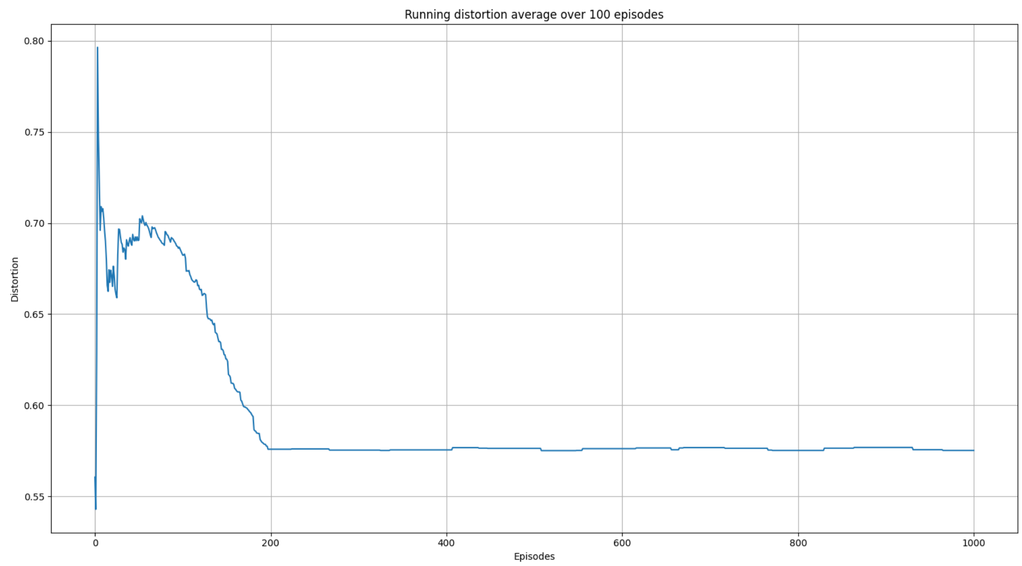


Fig. 7: Running average total distortion.

Semi-gradient SARSA algorithm

One of the major disadvantages of Qlearning we saw in the previous examples, is that we need to use a tabular representation of the state-action space. This poses limitations on how large the state space can be on current machines; for a data set with, say, 5 columns when each is discretized using 10 bins, this creates a state space of the the order $O(10^5)$. Although we won't address this here, we want to introduce the idea of weighting the columns. This idea comes from the fact that possibly not all columns carry the same information regarding anonimity and data set utility. Implicitly we decode this belief by categorizing the columns as

```
ColumnType.IDENTIFYING_ATTRIBUTE
ColumnType.QUASI_IDENTIFYING_ATTRIBUTE
ColumnType.SENSITIVE_ATTRIBUTE
ColumnType.INSENSITIVE_ATTRIBUTE
```

Thus, in this example, instead to representing the state-action function q_{π} using a table as we did in Q-learning on a three columns dataset, we will assume a functional form for it. Specifically, we assume that the state-action function can be approximated by $\hat{q} \approx q_{\pi}$ given by

$$\hat{q}(s, a) = \mathbf{w}^T \mathbf{x}(s, a) = \sum_{i}^{d} w_i, x_i(s, a)$$

where $\mathbf{w}$ is the weights vector and $\mathbf{x}(s, a)$ is called the feature vector representing state s when taking action a [1]. We will use *Tile coding* to construct $\mathbf{x}(s, \alpha)$. Our goal now is to find the components of the weight vector. We can use stochastic gradient descent (or SGD) for this [1]. In this case, the update rule is [1]

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \gamma \hat{q}(s_t, a_t, \mathbf{w}_t)\right] \nabla_{\mathbf{w}} \hat{q}(s_t, a_t, \mathbf{w}_t)$$

where α is the learning rate and U_t, for one-step SARSA, is given by [1]:

$$U_t = R_t + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}_t)$$

Since, $\hat{q}(s, a)$ is a linear function with respect to the weights, its gradient is given by

$$\nabla_{\mathbf{w}} \hat{q}(s, a) = \mathbf{x}(s, a)$$

The semi-gradient SARSA algorithm is shown below

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        If S' is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$

Fig. 8: Episodic semi-gradient SARSA algorithm. Image from [1].

Tile coding

Since we consider all the columns distortions in the data set, means that we deal with a multi-dimensional continuous spaces. In this case, we can use tile coding to construct $\mathbf{x}(s, \alpha)$ [1].

Tile coding is a form of coarse coding for multi-dimensional continuous spaces [1]. In this method, the features are grouped into partitions of the state space. Each partition is called a tiling, and each element of the partition is called a tile [1]. The following figure shows the a 2D state space partitioned in a uniform grid (left). If we only use this tiling, we would not have coarse coding but just a case of state aggregation.

In order to apply coarse coding, we use overlapping tiling partitions. In this case, each tiling is offset by a fraction of a tile width [1]. A simple case with four tilings is shown on the right side of following figure.

One practical advantage of tile coding is that the overall number of features that are active at a given instance is the same for any state [1]. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings [1]. This allows the learning parameter η, to be set according to

$$\eta = \frac{1}{n}$$

where n is the number of tilings.

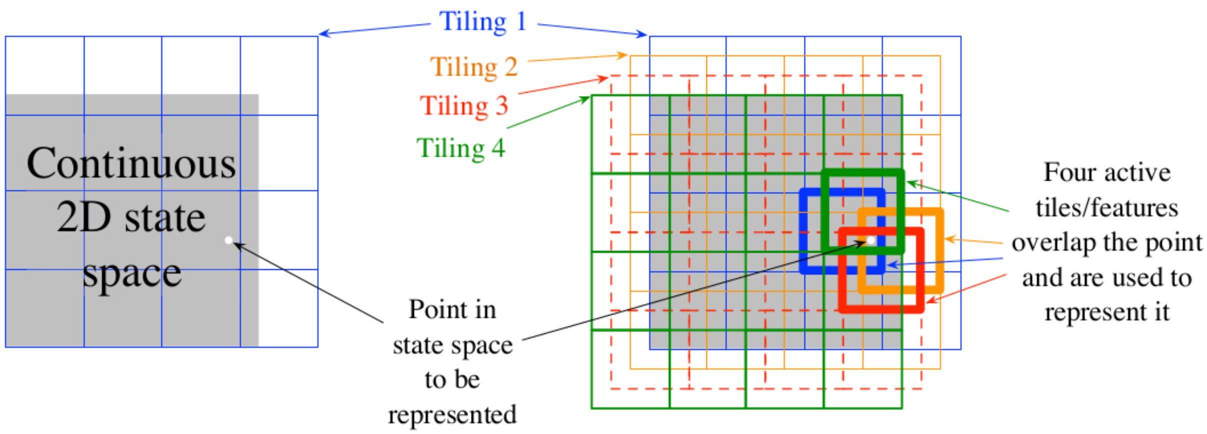


Fig. 9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension. Image from [1].

Code

The necessary imports

```
import random
import numpy as np

from src.algorithms.semi_gradient_sarsa import SemiGradSARSAConfig, SemiGradSARSA
from src.spaces.tiled_environment import TiledEnv, TiledEnvConfig, Layer

from src.spaces.action_space import ActionSpace
from src.spaces.actions import ActionIdentity, ActionStringGeneralize,␣
↪ActionNumericBinGeneralize
from src.trainers.trainer import Trainer, TrainerConfig
from src.policies.epsilon_greedy_policy import EpsilonDecayOption
from src.algorithms.epsilon_greedy_q_estimator import EpsilonGreedyQEstimatorConfig,␣
↪EpsilonGreedyQEstimator
from src.datasets import ColumnType
from src.spaces.env_type import DiscreteEnvType
from src.examples.helpers.load_full_mock_dataset import load_discrete_env, get_
↪ethinicity_hierarchy, \
    get_gender_hierarchy, get_salary_bins, load_mock_subjects
from src.examples.helpers.plot_utils import plot_running_avg
from src.utils import INFO
```

Next we set some constants

```
N_STATES = 10
N_LAYERS = 5
N_BINS = 10
N_EPISODES = 10001
OUTPUT_MSG_FREQUENCY = 100
GAMMA = 0.99
ALPHA = 0.1
```

(continues on next page)

(continued from previous page)

```
N_ITRS_PER_EPISODE = 30
EPS = 1.0
EPSILON_DECAY_OPTION = EpsilonDecayOption.CONSTANT_RATE
EPSILON_DECAY_FACTOR = 0.01
MAX_DISTORTION = 0.7
MIN_DISTORTION = 0.4
OUT_OF_MAX_BOUND_REWARD = -1.0
OUT_OF_MIN_BOUND_REWARD = -1.0
IN_BOUNDS_REWARD = 5.0
N_ROUNDS_BELOW_MIN_DISTORTION = 10
SAVE_DISTORTED_SETS_DIR = "semi_grad_sarsa_all_columns/distorted_set"
PUNISH_FACTOR = 2.0
USE_IDENTIFYING_COLUMNS_DIST = True
IDENTIFY_COLUMN_DIST_FACTOR = 0.1
```

The driver code brings all elements together

```
if __name__ == '__main__':

    # set the seed for random engine
    random.seed(42)

    # specify the column types. An identifying column
    # will me removed from the anonymized data set
    # An  INSENSITIVE_ATTRIBUTE remains intact.
    # A QUASI_IDENTIFYING_ATTRIBUTE is used in the anonymization
    # A SENSITIVE_ATTRIBUTE currently remains intact
    column_types = {"NHSno": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "given_name": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "surname": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "gender": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "dob": ColumnType.SENSITIVE_ATTRIBUTE,
                    "ethnicity": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "education": ColumnType.SENSITIVE_ATTRIBUTE,
                    "salary": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "mutation_status": ColumnType.SENSITIVE_ATTRIBUTE,
                    "preventative_treatment": ColumnType.SENSITIVE_ATTRIBUTE,
                    "diagnosis": ColumnType.INSENSITIVE_ATTRIBUTE}

    # define the action space
    action_space = ActionSpace(n=10)

    # all the columns that are SENSITIVE_ATTRIBUTE will be kept as they are
    # because currently we have no model
    # also INSENSITIVE_ATTRIBUTE will be kept as is
    # in order to declare this we use an ActionIdentity
    action_space.add_many(ActionIdentity(column_name="dob"),
                          ActionIdentity(column_name="education"),
                          ActionIdentity(column_name="salary"),
                          ActionIdentity(column_name="diagnosis"),
                          ActionIdentity(column_name="mutation_status"),
                          ActionIdentity(column_name="preventative_treatment"),
```

(continues on next page)

(continued from previous page)

```
                                ActionIdentity(column_name="ethnicity"),
                                ActionStringGeneralize(column_name="ethnicity",
                                                       generalization_table=get_ethinicity_
↪hierarchy()),
                                ActionStringGeneralize(column_name="gender",
                                                       generalization_table=get_gender_
↪hierarchy()),
                                ActionNumericBinGeneralize(column_name="salary",
                                                           generalization_table=get_salary_
↪bins(ds=load_mock_subjects(),
                                                                                          ␣
↪ n_states=N_STATES)))

    action_space.shuffle()

    # load the discrete environment
    env = load_discrete_env(env_type=DiscreteEnvType.MULTI_COLUMN_STATE, n_states=N_
↪STATES,
                            min_distortion={"ethnicity": 0.133, "salary": 0.133, "gender
↪": 0.133,
                                            "dob": 0.0, "education": 0.0, "diagnosis": 0.
↪0,
                                            "mutation_status": 0.0, "preventative_
↪treatment": 0.0,
                                            "NHSno": 0.0, "given_name": 0.0, "surname":␣
↪0.0},
                            max_distortion={"ethnicity": 0.133, "salary": 0.133, "gender
↪": 0.133,
                                            "dob": 0.0, "education": 0.0, "diagnosis": 0.
↪0,
                                            "mutation_status": 0.0, "preventative_
↪treatment": 0.0,
                                            "NHSno": 0.1, "given_name": 0.1, "surname":␣
↪0.1},
                            total_min_distortion=MIN_DISTORTION, total_max_
↪distortion=MAX_DISTORTION,
                            out_of_max_bound_reward=OUT_OF_MAX_BOUND_REWARD,
                            out_of_min_bound_reward=OUT_OF_MIN_BOUND_REWARD,
                            in_bounds_reward=IN_BOUNDS_REWARD,
                            punish_factor=PUNISH_FACTOR,
                            column_types=column_types,
                            action_space=action_space,
                            save_distoreted_sets_dir=SAVE_DISTORTED_SETS_DIR,
                            use_identifying_column_dist_in_total_dist=USE_IDENTIFYING_
↪COLUMNS_DIST,
                            use_identifying_column_dist_factor=IDENTIFY_COLUMN_DIST_
↪FACTOR,
                            gamma=GAMMA,
                            n_rounds_below_min_distortion=N_ROUNDS_BELOW_MIN_DISTORTION)

    # the configuration for the Tiled environment
    tiled_env_config = TiledEnvConfig(n_layers=N_LAYERS, n_bins=N_BINS,
```

(continues on next page)

(continued from previous page)

```
                                        env=env,
                                        column_ranges={"gender": [0.0, 1.0],
                                                       "ethnicity": [0.0, 1.0],
                                                       "salary": [0.0, 1.0]})

    # create the Tiled environment
    tiled_env = TiledEnv(tiled_env_config)
    tiled_env.create_tiles()

    # agent configuration
    agent_config = SemiGradSARSAConfig(gamma=GAMMA, alpha=ALPHA, n_itrs_per_episode=N_
↪ITRS_PER_EPISODE,
                                                                                   ␣
↪policy=EpsilonGreedyQEstimator(EpsilonGreedyQEstimatorConfig(eps=EPS, n_actions=tiled_
↪env.n_actions,
                                                                                   ␣
↪           decay_op=EPSILON_DECAY_OPTION,
                                                                                   ␣
↪           epsilon_decay_factor=EPSILON_DECAY_FACTOR,
                                                                                   ␣
↪           env=tiled_env,
                                                                                   ␣
↪           gamma=GAMMA,
                                                                                   ␣
↪           alpha=ALPHA)))
    # create the agent
    agent = SemiGradSARSA(agent_config)

    # create a trainer to train the SemiGradSARSA agent
    trainer_config = TrainerConfig(n_episodes=N_EPISODES, output_msg_frequency=OUTPUT_
↪MSG_FREQUENCY)
    trainer = Trainer(env=tiled_env, agent=agent, configuration=trainer_config)

    # train the agent
    trainer.train()

    # avg_rewards = trainer.avg_rewards()
    avg_rewards = trainer.total_rewards
    plot_running_avg(avg_rewards, steps=100,
                     xlabel="Episodes", ylabel="Reward",
                     title="Running reward average over 100 episodes")

    avg_episode_dist = np.array(trainer.total_distortions)
    print("{0} Max/Min distortion {1}/{2}".format(INFO, np.max(avg_episode_dist), np.
↪min(avg_episode_dist)))

    plot_running_avg(avg_episode_dist, steps=100,
                     xlabel="Episodes", ylabel="Distortion",
                     title="Running distortion average over 100 episodes")
```

The images above illustrate that there is clear evidence of learning as it was when using Qlearning. Furthermore, the training time is a lot more than the simple Qlearning algorithm. Thus, with the current implementation of semi0gradient SARSA we do not have any clear advantage. Instead, it could be argued that we maintain the constraints related with

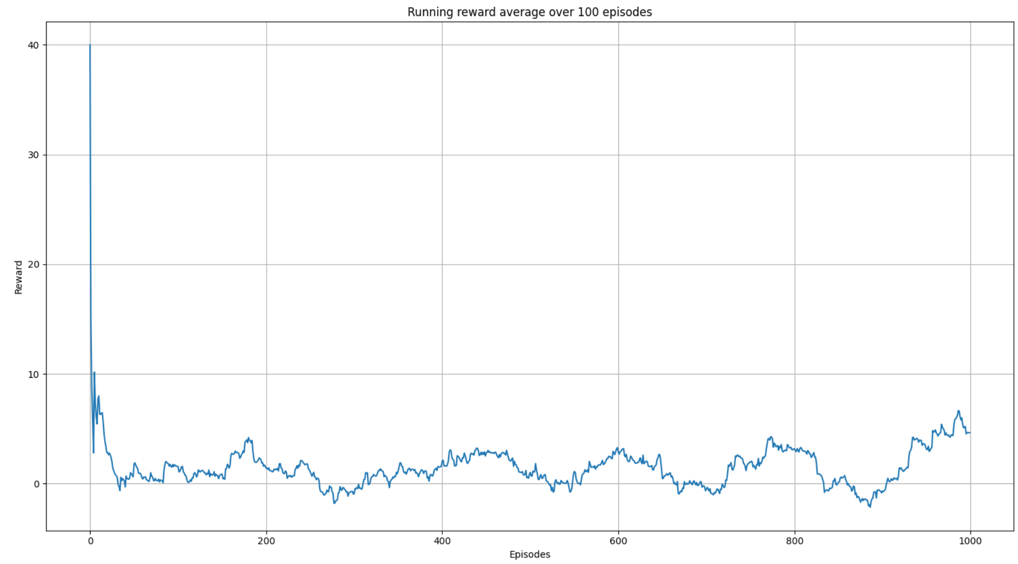


Fig. 10: Running average reward.

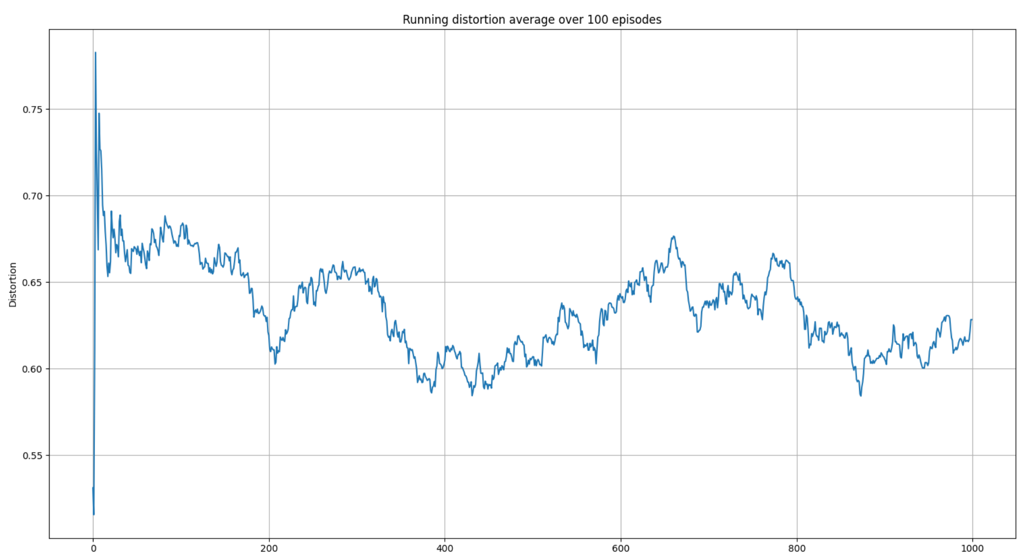


Fig. 11: Running average total distortion.

Qlearning (this comes form the tiling approach we used) without and clear advantage.

References

1. Richard S. Sutton and Andrw G. Barto, Reinforcement Learning. An Introduction 2nd Edition, MIT Press.

1.3.4 A2C algorithm on mock data set

Overview

Both the Q-learning algorithm we used in Q-learning on a three columns dataset and the SARSA algorithm in Semi-gradient SARSA on a three columns data set are value-based methods; that is they estimate directly value functions. Specifically the state-action function Q. By knowing Q we can construct a policy to follow for example to choose the action that at the given state maximizes the state-action function i.e. $argmax_{\alpha}Q(s_t, \alpha)$ i.e. a greedy policy. These methods are called off-policy methods.

However, the true objective of reinforcement learning is to directly learn a policy π. One class of algorithms towards this directions are policy gradient algorithms like REINFORCE and Advantage Actor-Critic or A2C algorithms. A review of A2C methods can be found in [1].

A2C algorithm

Typically with policy gradient methods and A2C in particular, we approximate directly the policy by a parameterized model. Thereafter, we train the model i.e. learn its paramters by taking samples from the environment. The main advantage of learning a parameterized policy is that it can be any learnable function e.g. a linear model or a deep neural network.

The A2C algorithm is a the synchronous version of A3C [2]. Both algorithms, fall under the umbrella of actor-critic methods. In these methods, we estimate a parameterized policy; the actor and a parameterized value function; the critic. The role of the policy or actor network is to indicate which action to take on a given state. In our implementation below, the policy network returns a probability distribution over the action space. Specifically, a tensor of probabilities. The role of the critic model is to evaluate how good is the action that is selected.

In our implementation we use a shared-weights model and use a single agent that interacts with multiple instances of the environment. In other words, we create a number of workers where each worker loads its own instance of the data set to anonymize.

The objective of the agent is to maximize the expected discounted return [2]:

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

where τ is the trajectory the agent observes with probability distribution ρ_{θ}, γ is the discount factor and $R(s_t, \alpha_t)$ represents some unknown to the agent reward function. We can rewrite the expression above as

$$J(\pi_{\theta}) = E_{\tau \sim \rho_{\theta}}\left[\sum_{t=0}^{T}\gamma^t R(s_t, a_t)\right] = \int \rho_{\theta}(\tau)\sum_{t=0}^{T}\gamma^t R(s_t, a_t)d\tau$$

Let's condense the involved notation by using $G(\tau)$ to denote the sum in the expression above i.e.

$$G(\tau) = \sum_{t=0}^{T}\gamma^t R(s_t, a_t)$$

The probability distribution ρ_{θ} should be a function of the followed policy π_{θ} as this dictates what action is followed. Indeed we can write [2],

$$\rho_{\theta} = p(s_0)\Pi_{t=0}^{\infty}\pi_{\theta}(a_t, s_t)P(s_{t+1}|s_t, a_t)$$

where $P(s_{t+1}|s_t, a_t)$ denotes the state transition probabilities. Policy gradient methods use the gardient of $J(\pi_{\theta})$ in order to make progress. It turns out, see for example [2, 3] that we can write

$$\nabla_{\theta}J(\pi_{\theta}) = \int \rho_{\theta}\nabla_{\theta}log(\rho_{\theta})G(\tau)d\tau$$

This equation above forms the essence of the policy gradient methods. However, we cannot fully evaluate the integral above as we don't know the transition probabilities. We can eliminate the term that involves the gradient $\nabla_{\theta}\rho_{\theta}$ by using the expression for ρ_{θ}

$$\nabla_{\theta}log(\rho_{\theta}) = \nabla_{\theta}log\left[p(s_0)\Pi_{t=0}^{\infty}\pi_{\theta}(a_t, s_t)P(s_{t+1}|s_t, a_t)\right]$$

From the expression above only the term $\pi_{\theta}(a_t, s_t)$ involves θ. Thus,

$$\nabla_{\theta}log(\rho_{\theta}) = \sum_{t=0}^{\infty}\nabla_{\theta}log(\pi_{\theta}(a_t, s_t)$$

We will use the expression above as well as batches of trajectories in order to calculate the integral above. In particular, we will use the following expression

$$\nabla_{\theta}J(\pi_{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T}\nabla_{\theta}log(\pi_{\theta}(a_t, s_t)\right)G(\tau)$$

where N is the size of the batch. There are various expressions for $G(\tau)$ (see e.g. [4]) . Belowe, we review some of them. The first expression is given by

$$G(\tau) = \sum_{t=0}^{T}\gamma^t R(s_t, a_t)$$

and this is the expression used by the REINFORCE algorithm [2]. However, this is a full Monte Carlo estimate and when N is small the gradient estimation may exhibit high variance. In such cases learning may not be stable. Another expression we could employ is known as the reward-to-go term [2]:

$$G(\tau) = \sum_{t'=t}^{T}\gamma^t R(s_{t'}, a_{t'})$$

Another idea is to use a baseline in order to reduce further the gradient variance [2]. One such approach is to use the so-called advantage function $A(s_t, \alpha_t)$ defined as [2]

$$A(s_t, a_t) = Q_{\pi}(s_t, a_t) - V_{\pi}(s_t)$$

The advantage function measures how much the agent is better off by taking action a_t when in state s_t as opposed to following the existing policy. Let's see how we can estimate the advantage function.

Estimate $A(s_t, a_t)$

The advantage function involes both the state-action value function $Q_{\pi}(s_t, a_t)$ as well as the value function $V_{\pi}(s_t)$. Given a model that somehow estimates $V_{\pi}(s_t)$, we can estimate $Q_{\pi}(s_t, a_t)$ from

$$Q_{\pi}(s_t, a_t) \approx G(\tau)$$

or

$$Q_{\pi}(s_t, a_t) \approx r_{t+1} + \gamma V_{\pi}(s_{t+1})$$

Resulting in

$$A(s_t, a_t) = r_{t+1} + \gamma V_{\pi}(s_{t+1}) - V_{\pi}(s_t)$$

GAE

The advantage actor-critic model we use in this section involves a more general form of the advanatge estimation known as Generalized Advantage Estimation or GAE. This is a method for estimating targets for the advantage function [3]. Specifically, we use the following expression for the advantage function [4]

$$A(s_t, a_t)^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+1}$$

when $\lambda = 0$ this expression results to the the expression for $A(s_t, a_t)$ [4].

A2C model

As we already mentioned, in actor-critic methods, there are two models; the actor and the critic. The role of the policy or actor model is to indicate which action to take on a given state There are two main architectures for actor-critic methods; completely isolated actor and critic models or weight sharing models [2]. In the former, the two models share no common aspects. The advantage of such an approach is that it is usually more stable. The second architecture allows for the two models to share some characteristics and differentiate in the last layers. Although this second option requires careful tuning of the hyperparameters, it has the advantage of cross learning and use common extraction capabilities [2].

In this example, we will follow the second architecture. Moreover, to speed up training, we will use a multi-process environment that gathers samples from multiple environments at once.

The loss function, we minimize is a weighted sum of the two loss functions of the participating models i.e.

$$L(\theta) = w_1 L_{\pi}(\theta) + w_2 L_{V_{\pi}}(\theta)$$

where

$$L_{\pi}(\theta) = J(\pi(\theta)) \;\; L_{V_{\pi}}(\theta) = MSE(y_i, V_{\pi}(s_i))$$

where MSE is the mean square error function and y_i are the state-value targets i.e.

$$y_i = r_i + \gamma V_{\pi}(s_i^{'}), \;\; i = 1, \cdots, N$$

Code

```
import random
from pathlib import Path
import numpy as np
import torch

from src.algorithms.a2c import A2C, A2CConfig
```

(continues on next page)

(continued from previous page)

```
from src.networks.a2c_networks import A2CNetSimpleLinear
from src.examples.helpers.load_full_mock_dataset import load_discrete_env, get_
↪ethinicity_hierarchy, \
    get_gender_hierarchy, get_salary_bins, load_mock_subjects
from src.datasets import ColumnType
from src.spaces.env_type import DiscreteEnvType
from src.spaces.action_space import ActionSpace
from src.spaces.actions import ActionIdentity, ActionStringGeneralize,␣
↪ActionNumericBinGeneralize
from src.utils.iteration_control import IterationControl
from src.examples.helpers.plot_utils import plot_running_avg
from src.spaces.multiprocess_env import MultiprocessEnv
from src.trainers.pytorch_trainer import PyTorchTrainer, PyTorchTrainerConfig
from src.maths.optimizer_type import OptimizerType
from src.maths.pytorch_optimizer_config import PyTorchOptimizerConfig
from src.utils import INFO
```

```
N_STATES = 10
N_ITRS_PER_EPISODE = 400
ACTION_SPACE_SIZE = 10
N_WORKERS = 3
N_EPISODES = 1001
GAMMA = 0.99
ALPHA = 0.1
PUNISH_FACTOR = 2.0
MAX_DISTORTION = 0.7
MIN_DISTORTION = 0.4
SAVE_DISTORTED_SETS_DIR = "/home/alex/qi3/drl_anonymity/src/examples/a2c_all_cols_multi_
↪state_results/distorted_set"
USE_IDENTIFYING_COLUMNS_DIST = True
IDENTIFY_COLUMN_DIST_FACTOR = 0.1
OUT_OF_MAX_BOUND_REWARD = -1.0
OUT_OF_MIN_BOUND_REWARD = -1.0
IN_BOUNDS_REWARD = 5.0
OUTPUT_MSG_FREQUENCY = 100
N_ROUNDS_BELOW_MIN_DISTORTION = 10
N_COLUMNS = 11
```

```
def env_loader(kwargs):

    column_types = {"NHSno": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "given_name": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "surname": ColumnType.IDENTIFYING_ATTRIBUTE,
                    "gender": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "dob": ColumnType.SENSITIVE_ATTRIBUTE,
                    "ethnicity": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "education": ColumnType.SENSITIVE_ATTRIBUTE,
                    "salary": ColumnType.QUASI_IDENTIFYING_ATTRIBUTE,
                    "mutation_status": ColumnType.SENSITIVE_ATTRIBUTE,
                    "preventative_treatment": ColumnType.SENSITIVE_ATTRIBUTE,
                    "diagnosis": ColumnType.INSENSITIVE_ATTRIBUTE}
```

(continues on next page)

(continued from previous page)

```
    # define the action space
    action_space = ActionSpace(n=ACTION_SPACE_SIZE)

    # all the columns that are SENSITIVE_ATTRIBUTE will be kept as they are
    # because currently we have no model
    # also INSENSITIVE_ATTRIBUTE will be kept as is
    # in order to declare this we use an ActionIdentity
    action_space.add_many(ActionIdentity(column_name="dob"),
                          ActionIdentity(column_name="education"),
                          ActionIdentity(column_name="salary"),
                          ActionIdentity(column_name="diagnosis"),
                          ActionIdentity(column_name="mutation_status"),
                          ActionIdentity(column_name="preventative_treatment"),
                          ActionIdentity(column_name="ethnicity"),
                          ActionStringGeneralize(column_name="ethnicity",
                                                 generalization_table=get_ethinicity_
↪hierarchy()),
                          ActionStringGeneralize(column_name="gender",
                                                 generalization_table=get_gender_
↪hierarchy()),
                          ActionNumericBinGeneralize(column_name="salary",
                                                     generalization_table=get_salary_
↪bins(ds=load_mock_subjects(),
                                                                                        ␣
↪ n_states=N_STATES)))
    # shuffle the action space
    # using different seeds
    action_space.shuffle(seed=kwargs["rank"] + 1)

    env = load_discrete_env(env_type=DiscreteEnvType.MULTI_COLUMN_STATE, n_states=N_
↪STATES,
                            min_distortion={"ethnicity": 0.133, "salary": 0.133, "gender
↪": 0.133,
                                            "dob": 0.0, "education": 0.0, "diagnosis": 0.
↪0,
                                            "mutation_status": 0.0, "preventative_
↪treatment": 0.0,
                                            "NHSno": 0.0, "given_name": 0.0, "surname":␣
↪0.0},
                            max_distortion={"ethnicity": 0.133, "salary": 0.133, "gender
↪": 0.133,
                                            "dob": 0.0, "education": 0.0, "diagnosis": 0.
↪0,
                                            "mutation_status": 0.0, "preventative_
↪treatment": 0.0,
                                            "NHSno": 0.1, "given_name": 0.1, "surname":␣
↪0.1},
                            total_min_distortion=MIN_DISTORTION, total_max_
↪distortion=MAX_DISTORTION,
                            out_of_max_bound_reward=OUT_OF_MAX_BOUND_REWARD,
                            out_of_min_bound_reward=OUT_OF_MIN_BOUND_REWARD,
```

(continues on next page)

(continued from previous page)

```
                              in_bounds_reward=IN_BOUNDS_REWARD,
                              punish_factor=PUNISH_FACTOR,
                              column_types=column_types,
                              action_space=action_space,
                              save_distoreted_sets_dir=SAVE_DISTORTED_SETS_DIR,
                              use_identifying_column_dist_in_total_dist=USE_IDENTIFYING_
↪COLUMNS_DIST,
                              use_identifying_column_dist_factor=IDENTIFY_COLUMN_DIST_
↪FACTOR,
                              gamma=GAMMA,
                              n_rounds_below_min_distortion=N_ROUNDS_BELOW_MIN_DISTORTION)

    # we want to get the distances as states
    # not bin indices
    env.config.state_as_distances = True

    return env
```

```
def action_sampler(logits: torch.Tensor) -> torch.distributions.Distribution:

    action_dist = torch.distributions.Categorical(logits=logits)
    return action_dist
```

```
if __name__ == '__main__':
    # set the seed for random engine
    random.seed(42)

    # set the seed for PyTorch
    torch.manual_seed(42)

    # this the A2C network
    net = A2CNetSimpleLinear(n_columns=N_COLUMNS, n_actions=ACTION_SPACE_SIZE)

    # agent configuration
    a2c_config = A2CConfig(action_sampler=action_sampler, n_iterations_per_episode=N_
↪ITRS_PER_EPISODE,
                           a2cnet=net, save_model_path=Path("./a2c_three_columns_output/
↪"),
                           n_workers=N_WORKERS,
                           optimizer_config=PyTorchOptimizerConfig(optimizer_
↪type=OptimizerType.ADAM,
                                                                   optimizer_learning_
↪rate=ALPHA))

    # create the agent
    agent = A2C(a2c_config)

    # create a trainer to train the Qlearning agent
    configuration = PyTorchTrainerConfig(n_episodes=N_EPISODES)

    # set up the arguments
```

(continues on next page)

(continued from previous page)

```
    env = MultiprocessEnv(env_builder=env_loader, env_args={}, n_workers=N_WORKERS)

    try:

        env.make(agent=agent)
        trainer = PyTorchTrainer(env=env, agent=agent, config=configuration)

        # train the agent
        trainer.train()

        avg_rewards = trainer.total_rewards
        plot_running_avg(avg_rewards, steps=100,
                         xlabel="Episodes", ylabel="Reward",
                         title="Running reward average over 100 episodes")

        avg_episode_dist = np.array(trainer.total_distortions)
        print("{0} Max/Min distortion {1}/{2}".format(INFO, np.max(avg_episode_dist), np.
↪min(avg_episode_dist)))

        plot_running_avg(avg_episode_dist, steps=100,
                         xlabel="Episodes", ylabel="Distortion",
                         title="Running distortion average over 100 episodes")


        # play the agent on the environment.
        # call the environment builder to create
        # an instance of the environment
        discrte_env = env.env_builder()

        stop_criterion = IterationControl(n_itrs=10, min_dist=MIN_DISTORTION, max_
↪dist=MAX_DISTORTION)
        agent.play(env=discrte_env, criteria=stop_criterion)

    except Exception as e:
        print("An excpetion was thrown...{0}".format(str(e)))
    finally:
        env.close()
```

Results

The following images show the performance of the learning process

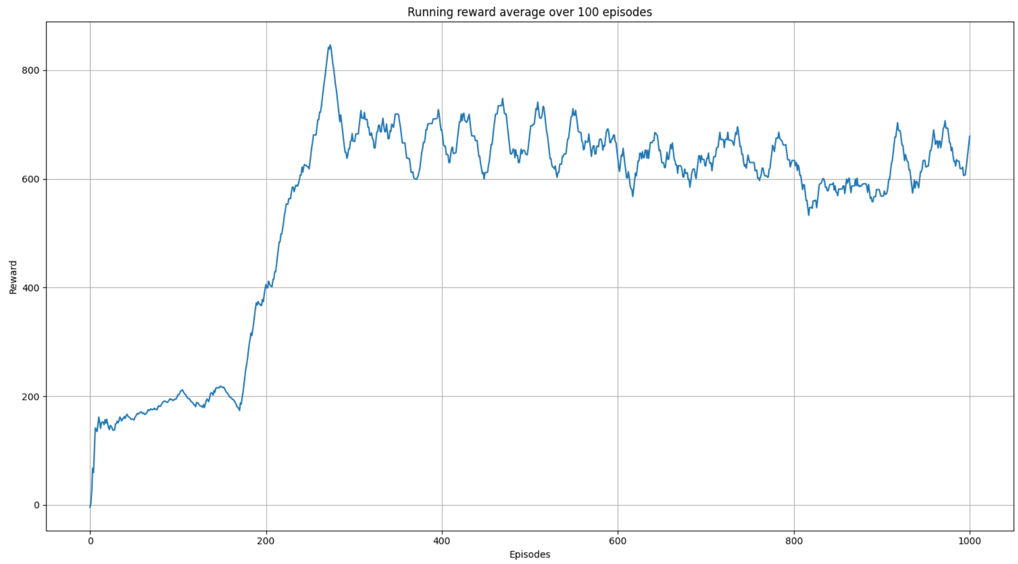


Fig. 12: Running average reward.

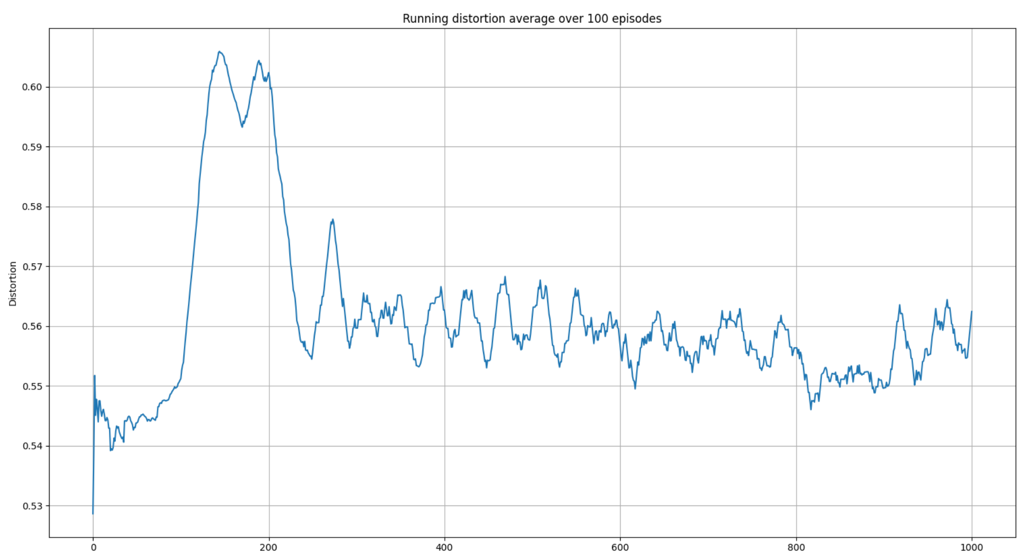


Fig. 13: Running average total distortion.

References

1. Ivo Grondman, Lucian Busoniu, Gabriel A. D. Lopes, Robert Babuska, A survey of Actor-Critic reinforcement learning: Standard and natural policy gradients. IEEE Transactions on Systems, Man and Cybernetics-Part C Applications and Reviews, vol 12, 2012.
2. Enes Bilgin, Mastering reinforcement learning with python. Packt Publishing.
3. Miguel Morales, Grokking deep reinforcement learning. Manning Publications.
4. John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, Pieter Abbeel, High-Dimensional Continuous Control Using Generalized Advantage Estimation, Last download 26/04/2022.

1.4 API

1.4.1 epsilon_greedy_q_estimator

Module epsilon_greedy_q_estimator. Implements a q-estimator by assuming linear function approximation

class epsilon_greedy_q_estimator.**EpsilonGreedyQEstimatorConfig**(*eps: float = 1.0, n_actions: int = 1, decay_op: EpsilonDecayOption = EpsilonDecayOption.NONE, max_eps: float = 1.0, min_eps: float = 0.001, epsilon_decay_factor: float = 0.01, user_defined_decrease_method: Optional[UserDefinedDecreaseMethod] = None, gamma: float = 1.0, alpha: float = 1.0, env: Optional[Env] = None*)

Configuration class for EpsilonGreedyQEstimator

class epsilon_greedy_q_estimator.**EpsilonGreedyQEstimator**(*config:* EpsilonGreedyQEstimatorConfig)

Q-function estimator using an epsilon-greedy policy for action selection

__init__(*config:* EpsilonGreedyQEstimatorConfig)

Constructor. Initialize the estimator with a given configuration

Parameters

config (*The instance configuration*) –

initialize() → None

Initialize the underlying weights

Return type

None

on_state(*state: State*) → Action

Returns the action on the given state

Parameters

state (*The state observed*) –

Return type

An environment specific Action type

q_hat_value(*state_action_vec: StateActionVec*) → float

Returns the *:math: hat{q}* approximate value for the given state-action vector

Parameters
`state_action_vec` (*The state-action tiled vector*) –

Return type
float

1.4.2 a2c

a2c.**calculate_discounted_returns**(*rewards: array*, *discounts: array*, *n_workers: int = 1*) → array

Calculate the discounted returns from the episode rewards

Parameters

- **rewards** (*The list of rewards*) –
- **discounts** (*The discount factor*) –
- **n_workers** (*The number of workers*) –

a2c.**create_discounts_array**(*end: int*, *base: float*, *start=0*, *endpoint=False*)

Create an array of floating point numbers in [start, end) with the given base

Parameters

- **end** –
- **base** –
- **start** –
- **endpoint** –

class a2c.**A2CConfig**(*gamma: float = 0.99*, *tau: float = 0.1*, *beta: Optional[float] = None*, *policy_loss_weight: float = 1.0*, *value_loss_weight: float = 1.0*, *max_grad_norm: float = 1.0*, *n_iterations_per_episode: int = 100*, *n_workers: int = 1*, *batch_size: int = 0*, *normalize_advantages: bool = True*, *device: str = 'cpu'*, *action_sampler: Optional[Callable] = None*, *a2cnet: Optional[Module] = None*, *save_model_path: Optional[Path] = None*, *optimizer_config: Optional[PyTorchOptimizerConfig] = None*)

Configuration for A2C algorithm

class a2c.**_ActResult**(*logprobs: torch.Tensor*, *values: torch.Tensor*, *actions: torch.Tensor*, *entropies: torch.Tensor*)

class a2c.**A2C**(*config: A2CConfig*)

__init__(*config: A2CConfig*)

_do_train(*env: Env*, *episode_idx: int*, ***options*) → EpisodeInfo

Train the algorithm on the episode. In fact this method simply plays the environment to collect batches

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type
An instance of EpisodeInfo

classmethod from_path(*config: A2CConfig*, *path: Path*)

Load the A2C model parameters from the given path

Parameters

- **config** (*The configuration of the algorithm*) –
- **path** (*The path to load the parameters*) –

Return type
An instance of A2C class

on_episode(*env: Env*, *episode_idx: int*, ***options*) → EpisodeInfo

Train the algorithm on the episode

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type
An instance of EpisodeInfo

parameters() → Any

The parameters of the underlying model

Return type
An array with the model parameters

1.4.3 q_learning

Simple Q-learning algorithm

class q_learning.**QLearnConfig**(*gamma: float = 1.0*, *alpha: float = 0.1*, *n_itrs_per_episode: int = 100*, *policy: Optional[Policy] = None*)

Configuration for Q-learning

class q_learning.**QLearning**(*algo_config: QLearnConfig*)

Q-learning algorithm implementation

__init__(*algo_config: QLearnConfig*)

Constructor. Construct an instance of the algorithm by passing the configuration parameters

Parameters
algo_config (*The configuration parameters*) –

_do_train(*env: Env*, *episode_idx: int*, ***option*) → EpisodeInfo

Train the algorithm on the episode

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type
An instance of EpisodeInfo

`_update_q_table`(*state: int*, *action: int*, *n_actions: int*, *reward: float*, *next_state: Optional[int] = None*) → None

Update the tabular state-action function

Parameters

- **state** (*State observed*) –
- **action** (*The action taken*) –
- **n_actions** (*Number of actions in the data set*) –
- **reward** (*The reward observed*) –
- **next_state** (*The next state observed*) –

Return type
None

`actions_after_episode_ends`(*env: Env*, *episode_idx: int*, ***options*) → None

Execute any actions the algorithm needs after the episode ends

Parameters

- **env** (*The environment that training occurs*) –
- **episode_idx** (*The episode index*) –
- **options** (*Any options passed by the client code*) –

Return type
None

`actions_before_training`(*env: Env*, ***options*) → None

Any actions before training begins

Parameters

- **env** (*The environment that training occurs*) –
- **options** (*Any options passed by the client code*) –

Return type
None

`on_episode`(*env: Env*, *episode_idx: int*, ***options*) → EpisodeInfo

Train the algorithm on the episode

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type
An instance of EpisodeInfo

`play`(*env: Env*, *stop_criterion: Criterion*) → None

Play the agent on the environment. This should produce a distorted dataset

Parameters

- **env** (*The environment to*) –
- **stop_criterion** (*The criteria to use to stop*) –

Return type
None

1.4.4 semi_gradient_sarsa

Module semi_gradient_sarsa. Implements episodic semi-gradient SARSA for estimating the state-action value function. the im[plementation follows the algorithm at page 244 in the book by Sutton and Barto: Reinforcement Learning An Introduction second edition 2020

class semi_gradient_sarsa.**SemiGradSARSAConfig**(*gamma: float = 1.0*, *alpha: float = 0.1*, *n_itrs_per_episode: int = 100*, *policy: Optional[Policy] = None*)

Configuration class for semi-gradient SARSA algorithm

class semi_gradient_sarsa.**SemiGradSARSA**(*config:* SemiGradSARSAConfig)

SemiGradSARSA class. Implements the semi-gradient SARSA algorithm as described

__init__(*config:* SemiGradSARSAConfig) → None

_do_train(*env: Env*, *episode_idx: int*, ***options*) → EpisodeInfo

Train the algorithm on the episode

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type
An instance of EpisodeInfo

_init() → None

Any initializations needed before starting the training

Return type
None

_validate() → None

Validate the state of the agent. Is called before any training begins to check that the starting state is sane

Return type
None

_weights_update(*env: Env*, *state: State*, *action: Action*, *reward: float*, *next_state: State*, *next_action: Action*, *t: float = 1.0*) → None

Update the weights due to the fact that the episode is finished

Parameters

- **env** (*The environment instance that the training takes place*) –
- **state** (*The current state*) –
- **action** (*The action we took at state*) –

- **reward** (*The reward observed when taking the given action when at the given state*) –
- **next_state** (*The observed new state*) –
- **next_action** (*The action to be executed in next_state*) –

Return type
None

_weights_update_episode_done(*env: Env*, *state: State*, *action: Action*, *reward: float*, *t: float = 1.0*) → None

Update the weights of the underlying Q-estimator

Parameters

- **state** (*The current state it is assumed to be a raw state*) –
- **reward** (*The reward observed when taking the given action when at the given state*) –
- **action** (*The action we took at the state*) –

Return type
None

actions_after_episode_ends(*env: Env*, *episode_idx: int*, ***options*) → None

Any actions after the training episode ends

Parameters

- **env** (*The training environment*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any options passed by the client code*) –

Return type
None

actions_before_episode_begins(*env: Env*, *episode_idx: int*, ***options*) → None

Any actions to perform before the episode begins

Parameters

- **env** (*The instance of the training environment*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any keyword options passed by the client code*) –

Return type
None

actions_before_training(*env: Env*, ***options*) → None

Specify any actions necessary before training begins

Parameters

- **env** (*The environment to train on*) –
- **options** (*Any key-value options passed by the client*) –

Return type
None

on_episode(*env: Env*, *episode_idx: int*, ***options*) → EpisodeInfo

Train the algorithm on the episode

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The index of the training episode*) –
- **options** (*Any keyword based options passed by the client code*) –

Return type

An instance of EpisodeInfo

play(*env: Env*, *stop_criterion: Criterion*) → None

Play the agent on the environment. This should produce a distorted dataset

Parameters

- **env** (*The environment to*) –
- **stop_criterion** (*The criteria to use to stop*) –

Return type

None

1.4.5 column_type

Module column_type specifies an enumeration of the column. This is similar to the ARX software. See the ARX documentation at: https://arx.deidentifier.org/wp-content/uploads/javadoc/current/api/org/deidentifier/arx/AttributeType.html

class column_type.**ColumnType**(*value*)

An enumeration.

1.4.6 datasets_loaders

1.4.7 dataset_wrapper

1.4.8 exceptions

class exceptions.**Error**(*message*)

General error class to handle generic errors

__init__(*message*) → None

__str__()

Return str(self).

class exceptions.**IncompatibleVectorSizesException**(*size1: int*, *size2: int*)

__init__(*size1: int*, *size2: int*) → None

__str__()

Return str(self).

class exceptions.**InvalidDataTypeException**(*param_name: str*, *param_type: Any*, *param_types: str*)

__init__(*param_name: str*, *param_type: Any*, *param_types: str*)

__str__()

Return str(self).

class exceptions.**InvalidFileFormat**(*filename*)

__init__(*filename*)

__str__()

Return str(self).

class exceptions.**InvalidParamValue**(*param_name: str*, *param_value: str*)

__init__(*param_name: str*, *param_value: str*)

__str__()

Return str(self).

class exceptions.**InvalidSchemaException**(*message: str*)

__init__(*message: str*) → None

__str__()

Return str(self).

class exceptions.**InvalidStateException**(*type_name: str*, *state_type: str*)

__init__(*type_name: str*, *state_type: str*) → None

__str__()

Return str(self).

1.4.9 optimizer_type

Module optimizer_type. Specifies an enumeration for various PyTorch optimizers

class optimizer_type.**OptimizerType**(*value*)

An enumeration.

1.4.10 pytorch_optimizer_builder

Module pytorch_optimizer_builder. Specifies a simple factory for building PyTorch optimizers

pytorch_optimizer_builder.**pytorch_optimizer_builder**(*opt_type: OptimizerType*, *model_params: Any*, ***options*) → Optimizer

Factory method for building PyTorch optimizers

Parameters

- **opt_type** (*The type of the optimizer*) –
- **model_params** (*Model parameters to optimize on*) –
- **options** (*Options for the optimizer*) –

Return type

A concrete instance of the optim.Optimizer class

1.4.11 loss_functions

Module loss_functions. Implements basic loss functions geared towards using PyTorch

`loss_functions.mse`(*returns: Tensor*, *values: Tensor*) → Tensor

Mean square error loss function

Parameters

- **returns** (*Values 1*) –
- **values** (*Values 2*) –

Return type

A torch tensor representing the MSE loss

1.4.12 distortion_calculator

1.4.13 numeric_distance_type

Enumeration helper for quick and uniform access of the various distance metrics

`class numeric_distance_type.NumericDistanceType`(*value*)

Enumeration of the various distance types

1.4.14 numeric_distance_calculator

1.4.15 pytorch_optimizer_config

Module pytorch_optimizer_configuration. Specifies a data class for configuring PyTorch optimizers

`class pytorch_optimizer_config.PyTorchOptimizerConfig`(*optimizer_type: OptimizerType = OptimizerType.ADAM*, *optimizer_learning_rate: float = 0.01*, *optimizer_betas: tuple = (0.9, 0.999)*, *optimizer_weight_decay: float = 0*, *optimizer_amsgrad: bool = False*)

Configuration class for the optimizer

1.4.16 string_distance_calculator

1.4.17 a2c_networks

Module a2c_networks. Specifies various networks for A2C algorithm

`class a2c_networks.A2CNetSimpleLinear`(*n_columns: int*, *n_actions: int*)

A2CNetSimpleLinear. Specifies a network architecture consisting of three linear layers

`__init__`(*n_columns: int*, *n_actions: int*)

Constructor.

Parameters

- **n_columns** (*Number of columns*) –

- **n_actions** (*Number of actions*) –

forward(*x: Tensor*) → tuple

Pass the state from the network

Parameters
x (*The torch tensor that represents the state*) –

Return type
The actor and the critic values

1.4.18 processes_manager

module process_manager. Utilities for managing processes

class processes_manager.**TorchProcsHandler**(*n_procs: int*)

The TorchProcsHandler class. Utility class to handle PyTorch processe

__init__(*n_procs: int*) → None

Constructor

Parameters
n_procs (*The number of processes to handle*) –

__len__() → int

The number of workers handled by this instance

1.4.19 epsilon_greedy_policy

Module epsilon_greedy_policy. Implements epsilon-greedy policy with various decay options

class epsilon_greedy_policy.**EpsilonDecayOption**(*value*)

Options for reducing epsilon

class epsilon_greedy_policy.**EpsilonGreedyConfig**(*eps: float = 1.0, n_actions: int = 1, decay_op: EpsilonDecayOption = EpsilonDecayOption.NONE, max_eps: float = 1.0, min_eps: float = 0.001, epsilon_decay_factor: float = 0.01, user_defined_decrease_method: Optional[UserDefinedDecreaseMethod] = None*)

Configuration class for EpsilonGreedyPolicy

class epsilon_greedy_policy.**EpsilonGreedyPolicy**(*eps: float, n_actions: int, decay_op: EpsilonDecayOption, max_eps: float = 1.0, min_eps: float = 0.001, epsilon_decay_factor: float = 0.01, user_defined_decrease_method: Optional[UserDefinedDecreaseMethod] = None*)

Epsilon-greedy policy implementation

__call__(*q_table: QTable, state: State*) → int

Execute the policy

Parameters

- **q_table** (*The q-table to use*) –
- **state** (*The state observed*) –

Return type
An integer representing the action index

`__init__`(*eps: float*, *n_actions: int*, *decay_op:* EpsilonDecayOption, *max_eps: float = 1.0*, *min_eps: float = 0.001*, *epsilon_decay_factor: float = 0.01*, *user_defined_decrease_method: Optional[UserDefinedDecreaseMethod] = None*)

Constructor. Initialize a policy with the given options

Parameters

- **eps** (*The initial epsilon*) –
- **n_actions** (*How many actions the environment assumes*) –
- **decay_op** (*How to decay epsilon*) –
- **max_eps** (*The maximum epsilon*) –
- **min_eps** (*The minimum epsilon*) –
- **epsilon_decay_factor** (*A decay factor used when decay_op = CONSTANT_RATE*) –
- **user_defined_decrease_method** (*A user defined callable to decay epsilon*) –

`__str__`() → str

Returns the name of the policy

Return type
A string representing the name of the policy

`actions_after_episode`(*episode_idx: int*, ***options*) → None

Any actions the policy should execute after the episode ends

Parameters

- **episode_idx** (*The episode index*) –
- **options** (*Any options passed by the client code*) –

Return type
None

`classmethod from_config`(*config:* EpsilonGreedyConfig)

Construct a policy from the given configuration

Parameters
config (*The configuration to use*) –

Return type
An instance of EpsilonGreedyPolicy class

`on_state`(*state: State*) → int

Returns the optimal action on the current state

Parameters
state (*The state observed*) –

Return type
An integer representing the action index

1.4.20 preprocess_utils

1.4.21 actions

The actions module. This module includes various actions to be applied by the implemented RL agents

class actions.**ActionType**(*value*)

Defines the type of an Action

class actions.**ActionBase**(*column_name: str*, *action_type: ActionType*)

Base class for actions

__init__(*column_name: str*, *action_type: ActionType*) → None

Constructor

Parameters

- **column_name** (*The name of the column this is acting on*) –
- **action_type** (*The type of the action*) –

abstract act(***ops*) → Any

Perform the action

Parameters

ops (*The data to distort*) –

Return type

Typically the action returns the distorted subset of the data

class actions.**ActionIdentity**(*column_name: str*)

Implements the identity action. Use this action to signal that no distortion should be applied.

__init__(*column_name: str*) → None

Constructor

Parameters

column_name (*The name of the column this is acting on*) –

act(***ops*) → Any

Perform the action

Parameters

ops (*The data to distort*) –

Return type

The distorted column

class actions.**ActionNumericBinGeneralize**(*column_name: str*, *generalization_table: Hierarchy*)

Generalization Action for numeric columns using bins

__init__(*column_name: str*, *generalization_table: Hierarchy*)

Constructor :param column_name: :type column_name: The name of the column this is acting on :param generalization_table: :type generalization_table: The bins to use

act(***ops*) → Any

Perform the action :param ops: :type ops: The data to distort

Return type

Typically the action returns the distorted subset of the data

class actions.**ActionNumericStepGeneralize**(*column_name: str*, *step: float*)

__init__(*column_name: str*, *step: float*)

Constructor

Parameters

- **column_name** (*The name of the column this is acting on*) –
- **action_type** (*The type of the action*) –

act(***ops*)

Perform an action :return:

class actions.**ActionRestore**(*column_name: str*, *restore_values: Hierarchy*)

Implements the restore action

__init__(*column_name: str*, *restore_values: Hierarchy*)

Constructor

Parameters

- **column_name** (*The name of the column this is acting on*) –
- **action_type** (*The type of the action*) –

act(***ops*) → Any

Perform an action :return:

class actions.**ActionStringGeneralize**(*column_name: str*, *generalization_table: Hierarchy*)

Implements the generalization action. The generalization_table must implement the __getitem__ function

__init__(*column_name: str*, *generalization_table: Hierarchy*) → None

Constructor

Parameters

- **column_name** (*The column name this action is acting on*) –
- **generalization_table** (*The hierarchy for the generalization*) –

act(***ops*) → Any

Performs the action

Parameters

ops (*The data to distort*) –

Return type

The distorted data

add(*key: Any*, *value: Any*) → None

Add a new item in the underlying hierarchy

Parameters

- **key** (*The key to use for the new item*) –
- **value** (*The value of the new item*) –

Return type

None

class actions.**ActionSuppress**(*column_name: str*, *suppress_table: Hierarchy*)

Implements the suppress action

__init__(*column_name: str*, *suppress_table: Hierarchy*)

Constructor

Parameters

- **column_name** (*The name of the column this is acting on*) –
- **action_type** (*The type of the action*) –

act(***ops*) → None

Perform the action :return: None

class actions.**ActionTransform**(*column_name: str*, *transform_value: Any*)

Implements the transform action

__init__(*column_name: str*, *transform_value: Any*)

Constructor

Parameters

- **column_name** (*The name of the column this is acting on*) –
- **action_type** (*The type of the action*) –

act(***ops*) → Any

Perform an action :return:

1.4.22 action_space

Module action_space Specifies a wrapper to the discrete actions in the actions.py module

class action_space.**ActionSpace**(*n: int*)

ActionSpace class models a discrete action space of size n

1.4.23 state

The state module. Specifies a wrapper to a state such that it exposes column distortions and the bin index of the overall distortion.

class state.**StateIterator**(*values: List*)

StateIterator class. Helper class to iterate over the columns of a State object

__init__(*values: List*)

__len__()

Returns the total number of items in the iterator :return:

property at: Any

Returns the value of the iterator at the current position without incrementing the position of the iterator :return: Any

property finished: bool

Returns true if the iterator is exhausted :return:

`class state.State`

Helper to represent a State

`__contains__(column_name: str) → bool`

Returns true if column_name is in the column_distortions keys

Parameters
`column_name` (*The column name to query*) –

Returns

- *A boolean indicating if column_name is in the column_distortions*
- *keys or not.*

`__getitem__(name: str) → float`

Get the distortion corresponding to the name-th column

Parameters
`name` (*The name of the column*) –

Return type
The column distortion

`__init__()`

1.4.24 discrete_state_environment

RL Environment API taken from https://github.com/deepmind/dm_env/blob/master/dm_env/_environment.py

Classes

`DiscreteEnvConfig`([data_set, action_space, ...])	Configuration for discrete environment
`DiscreteStateEnvironment`(env_config)	The DiscreteStateEnvironment class.

1.4.25 tiled_environment

1.4.26 time_step

Module time_step. Specifies a wrapper for representing a step in the environment

`time_step.copy_time_step(time_step: TimeStep, **copy_options) → TimeStep`

Helper to copy partly or in whole a TimeStep namedtuple. If copy_options is None or empty it returns a deep copy of the given time step

Parameters

- `time_step` (*The time step to copy*) –
- `copy_options` (*Members to be copied*) –

Return type
An instance of the TimeStep namedtuple

`class time_step.StepType(value)`

Defines the status of a *TimeStep* within a sequence.

class time_step.**TimeStep**(*step_type*, *info*, *reward*, *discount*, *observation*)

1.4.27 multiprocess_env

Module multiprocess_env. Specifies a vectorized environment where each instance of the environment is run independently. The implementation of the environment is taken from the book Grokking Deep Reinforcement Learning Algorithms by Manning publications

class multiprocess_env.**MultiprocessEnv**(*env_builder: Callable*, *env_args: dict*, *n_workers: int*)

MultiprocessEnv class

__init__(*env_builder: Callable*, *env_args: dict*, *n_workers: int*)

__len__() → int

The number of workers handled by this instance

_broadcast_msg(*msg*)

Broadcast the message to all workers

Parameters
msg –

_send_msg(*msg: Any*, *rank: int*)

Send the message to the process with the given rank

Parameters

- **msg** (*The message to send*) –
- **rank** (*The rank of the proces to send the message*) –

make(*agent: Agent*)

Create the workers

work(*rank*, *env_builder: Callable*, *env_args: dict*, *agent: Agent*, *pipe_end*) → None

The worker function

Parameters

- **rank** (*The rank of the worker*) –
- **env_builder** (*The callable that builds the worker environment*) –
- **env_args** (*The callable arguments*) –
- **worker_end** –

Return type
None

1.4.28 replay_buffer

class replay_buffer.**ReplayBuffer**(*buffer_size: int*)

The ReplayBuffer class. Models a fixed size replay buffer. The buffer is represented by using a deque from Python's built-in collections library. This is basically a list that we can set a maximum size. If we try to add a new element whilst the list is already full, it will remove the first item in the list and add the new item to the end of the list. Hence new experiences replace the oldest experiences. The experiences themselves are tuples of (state1, reward, action, state2, done) that we append to the replay deque and they are represented via the named tuple ExperienceTuple

__getitem__(*name_attr: str*) → List

Return the full batch of the name_attr attribute

Parameters

- **name_attr** (*The name of the attribute to collect the*) –
- **values** (*batch*) –

Return type

A list

__init__(*buffer_size: int*)

Constructor

Parameters

buffer_size (*The maximum capacity of the buffer*) –

__len__() → int

Return the current size of the internal memory.

add(*state: Any*, *action: Any*, *reward: Any*, *next_state: Any*, *done: Any*, *info: dict = {}*) → None

Add a new experience tuple in the buffer

Parameters

- **state** (*The current state*) –
- **action** (*The action taken*) –
- **reward** (*The reward observed*) –
- **next_state** (*The next state observed*) –
- **done** (*Whether the episode is done*) –
- **info** (*Any other info needed*) –

Return type

None

get_item_as_torch_tensor(*name_attr: str*) → Tensor

Returns a torch.Tensor representation of the the named item

Parameters

name_attr (*The name of the attribute*) –

Return type

An instance of torch.Tensor

reinitialize() → None

Reinitialize the internal buffer

Return type
None

sample(*batch_size: int*) → List[ExperienceTuple]

Randomly sample a batch of experiences from memory.

Parameters
batch_size (*The batch size we want to sample*) –

Return type
A list of ExperienceTuple

1.4.29 trainer

Module trainer. Specifies a utility class for training serial reinforcement learning algorithms

class trainer.**TrainerConfig**(*n_episodes: int = 1*, *output_msg_frequency: int = - 1*)

class trainer.**Trainer**(*env: Env*, *agent: Agent*, *configuration: TrainerConfig*)

__init__(*env: Env*, *agent: Agent*, *configuration: TrainerConfig*) → None

Constructor. Initialize a trainer by passing the training environment instance the agen to train and configuration dictionary

Parameters

- **env** (*The environment to train the agent*) –
- **agent** (*The agent to train*) –
- **configuration** (*Configuration parameters for the trainer*) –

actions_after_episode_ends(*env: Env*, *episode_idx: int*, ***options*) → None

Any actions after the training episode ends

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any options passed by the client code*) –

Return type
None

actions_before_episode_begins(*env: Env*, *episode_idx: int*, ***options*) → None

Perform any actions necessary before the training begins

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any options passed by the client code*) –

Return type
None

actions_before_training() → None

Any actions to perform before training begins

Return type

None

avg_distortion() → array

Returns the average reward per episode :return:

avg_rewards() → array

Returns the average reward per episode :return:

train() → None

Train the agent on the given environment

Return type

None

1.4.30 pytorch_trainer

Module pytorch_multi_process_trainer. Specifies a trainer for PyTorch-based models.

pytorch_trainer.**worker**(*worker_idx: int*, *worker_model: Module*, *params: dir*)

Executes the process work

Parameters

- **worker_idx** (*The id of the worker*) –
- **worker_model** (*The model the worker is using*) –
- **params** (*Parameters needed*) –

class pytorch_trainer.**PyTorchTrainerConfig**(*n_procs: int = 1*, *n_episodes: int = 100*)

Configuration for PyTorchMultiProcessTrainer

class pytorch_trainer.**PyTorchTrainer**(*env: Env*, *agent: Agent*, *config: PyTorchTrainerConfig*)

The class PyTorchMultiProcessTrainer. Trainer for multiprocessing with PyTorch

__init__(*env: Env*, *agent: Agent*, *config: PyTorchTrainerConfig*) → None

Constructor. Initialize a trainer by passing the training environment instance the agent to train and configuration dictionary

Parameters

- **agent** (*The agent to train*) –
- **config** (*Configuration parameters for the trainer*) –

actions_after_episode_ends(*env: Env*, *episode_idx: int*, ***options*) → None

Any actions after the training episode ends

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any options passed by the client code*) –

Return type

None

actions_before_episode_begins(*env: Env*, *episode_idx: int*, ***options*) → None

Perform any actions necessary before the training begins

Parameters

- **env** (*The environment to train on*) –
- **episode_idx** (*The training episode index*) –
- **options** (*Any options passed by the client code*) –

Return type

None

actions_before_training() → None

Any actions to perform before training begins

Return type

None

avg_distortion() → array

Returns the average reward per episode :return:

avg_rewards() → array

Returns the average reward per episode :return:

1.4.31 iteration_control

module iteration_control. Utility to control iteration

class iteration_control.**IterationControl**(*n_itrs: int*, *min_dist: float*, *max_dist: float*)

Helper class to control iteration

__init__(*n_itrs: int*, *min_dist: float*, *max_dist: float*) → None

1.4.32 function_wraps

function_wraps.**time_func**(*fn: Callable*)

Execute the given callable and time the time

it took to execute

Parameters

fn (*Callable to execute*) –

1.4.33 episode_info

Module episode_info. Specifies the dataclass EpisodeInfo that is used as the return item of on_episode() agent function to wrap episode results. This is a helper class to wrap the output after an episode has finished

class episode_info.**EpisodeInfo**(*episode_itrs: int = 0*, *episode_score: float = 0.0*, *total_distortion: float = 0.0*, *total_execution_time: float = 0.0*, *info: dict = <factory>*)

1.4.34 mixins

module mixins. Various mixin classes to use for simplifying code

class mixins.**WithHierarchyTable**

__init__() → None

add_hierarchy(*key: str*, *hierarchy: Hierarchy*) → None

Add a hierarchy for the given key :param key: The key to attach the Hierarchy :param hierarchy: The hierarchy to attach :return: None

finished() → bool

Returns true if the action has exhausted all its transforms :return:

reset_iterators()

Reinitialize the iterators in the table :return:

class mixins.**WithQTableMixinBase**(*table: Optional[QTable] = None*)

Base class to impose the concept of Q-table

__init__(*table: Optional[QTable] = None*)

class mixins.**WithQTableMixin**(*table: Optional[QTable] = None*)

Helper class to associate a q_table with an algorithm

__init__(*table: Optional[QTable] = None*)

Constructor

Parameters

table (*The Q-table representing the Q-function*) –

class mixins.**WithMaxActionMixin**(*table: Optional[QTable] = None*)

The class WithMaxActionMixin.

__init__(*table: Optional[QTable] = None*)

Constructor

Parameters

table (*The Q-table representing the Q-function*) –

max_action(*state: Any*, *n_actions: int*) → int

Return the action index that presents the maximum value at the given state :param state: state index :param n_actions: Total number of actions allowed :return: The action that corresponds to the maximum value

class mixins.**WithEstimatorMixin**

1.4.35 reward_manager

module reward_manager specifies a class that handles the rewards awarded by the environment.

class reward_manager.**RewardManager**(*bounds: tuple*, *out_of_max_bound_reward: float*, *out_of_min_bound_reward: float*, *in_bounds_reward: float*, *punish_factor: float*, *min_distortions: Any*, *max_distortions: Any*)

The RewardManager class

__init__(*bounds: tuple*, *out_of_max_bound_reward: float*, *out_of_min_bound_reward: float*, *in_bounds_reward: float*, *punish_factor: float*, *min_distortions: Any*, *max_distortions: Any*) → None

get_reward_for_state(*total_distortion: float*, *current_state: State*, *next_state: State*, *min_dist_bins: Any*, ***options*) → float

Returns a user specified reward signal depending on the state and the options given

Parameters

- **state** –
- **options** –

1.4.36 serial_hierarchy

module serial_hierarchy. A SerialHierarchy represents a hierarchy of transformations that are applied one after the other

class serial_hierarchy.**SerialHierarchy**(*values: dict*)

A SerialHierarchy represents a hierarchy of transformations that are applied one after the other. Applications should explicitly provide the list of the ensuing transformations. For example assume that the data field has the value 'foo' then values

the following list ['fo*', 'f**', '***']

__getitem__(*item*)

Returns the item-th item :param item: :return:

__init__(*values: dict*) → None

Constructor. Initialize the hierarchy by passing the list of the ensuing transformations. :param values:

__len__()

Returns the size of the hierarchy :return:

__setitem__(*key*, *value*)

Set the key-th item to the given value. If the key-th item has already been set it overrides the existing value :param key: :param value: :return:

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

P

Q

R

S

T

W