Exploring reinforcement learning techniques for discrete and continuous control tasks in the MuJoCo environment

Vaddadi Sai Rahul [1], **Debajyoti Chakraborty** [1]
[1]Northeastern University
360 Huntington Avenue
Boston, Massachusetts 02115
vaddadi.s@northeastern.edu, chakraborty.de@northeastern.edu

Abstract

We leverage the fast physics simulator, MuJoCo to run tasks in a continuous control environment and reveal details like the observation space, action space, rewards, etc. for each task. We benchmark value-based methods for continuous control by comparing Q-learning and SARSA through a discretization approach, and using them as baselines, progressively moving into one of the state-of-the-art deep policy gradient method DDPG. Over a large number of episodes, Q-learning outscored SARSA, but DDPG outperformed both in a small number of episodes. Lastly, we also fine-tuned the model hyper-parameters expecting to squeeze more performance but using lesser time and resources. We anticipated that the DDPG's new design would vastly improve performance, yet after only a few episodes, we were able to achieve decent average rewards. We expect to improve the performance provided adequate time and computational resources.

Introduction

We will look at several reinforcement learning strategies for solving problems in discrete and continuous observation spaces, as well as discrete and continuous action spaces, in this work. In a continuous environment, predicting behaviors over a continuous space has always been a challenging challenge for an agent. Distinguishing the observation and action spaces is an obvious approach to these problems. The loss of information that occurs with dividing a continuous region into 'K' buckets is a significant constraint. Increasing the number of buckets can assist, but for continuous areas, it becomes intractable when the action-value table grows exponentially huge. We investigate the various environments in OpenAI gym's MuJoCo in this paper. To cope with continuous methods via bucketing, we use model-free temporal difference learning approaches - Q-learning and SARSA as a baseline. To improve the findings, the Deep Deterministic Policy Gradient (DDPG) was used.

We consider a typical reinforcement learning setup in which an agent interacts with its environment. The environment we have taken into consideration here, is **MuJoCo** (stands for **Mu**lti-**Jo**int dynamics with **Co**ntact). It is a general-purpose physics engine designed to help with research and development in robotics, biomechanics, machine learning, and other fields that need quick and precise modeling of articulated structures interacting with their surroundings.

Environment.

MuJoCo is a C/C++ library with a C API, which operates on low-level data structures which are pre-allocated by the built-in XML parser and compiler. Interactive visualization with a native GUI, produced in OpenGL, is included in the package. It provides continuous control tasks, running in a fast physics simulator and a plethora of utility functions for computing physics-related numbers. MuJoCo offers a range of continuous control task:

Model elements.

The elements of a MuJoCo model are as:

1. *Body:* Bodies are the components that make up kinematic trees, *i.e.*, a tree of rigid bodies, such as, the human body, only having a predefined mass and inertia. Bodies do not posses any geometric properties.
2. *Joint:* Joints are defined inside bodies. Joints help to create motion between the particular body and its parent, otherwise they would be stiff and immovable. MuJoCo joints have four primitive types: *slide*, *hinge*, *ball* and *free*.
3. *DOF:* Degrees of freedom (DOFs) refers to the limits to which physical movement of the rigid bodies are possible. They are closely related to joints, however, different joints can have multiple DOFs. DOFs can have properties like damping, maximum velocity, armature, inertia, friction and other relevant data from coordinate systems.
4. *Geom:* Geoms are mass-less geometric objects primarily used in collision detection. MuJoCo supports geom types as *plane*, *sphere*, *capsule*, *ellipsoid*, *box*, *cone*, and *mesh*.
5. *Site:* Sites are locations of interest that are defined in the bodies' local frames and hence move with them. They are utilized in the engine to route tendons and apply various sorts of forces, but they may also be used by the application to encode sensor positions and other information.
6. *Constraint:* Constraints are used to define a set of preformulated rules that specify how they will behave in the environment, like restraining ball or hinge joints.

7. *Tendon:* A tendon can be used to impose constraints as "...the shortest path that passes through a sequence of specified sites or wraps around specified geoms."
8. *Actuator:* Actuators receive control inputs from the environment that directly co-relate to the movement or kinematics of the model. They can transmit forces (*e.g.* torque), on any of joints, sites or tendons.

Related work

Researchers have recently achieved substantial success by integrating deep learning capabilities for learning feature representations with reinforcement learning. Some instances include teaching agents to play video games using raw pixel data and teaching them sophisticated manipulation skills. Other instances include designing generalized agents that can "reinforce" itself into any task, given enough time and resources.

Expected SARSA might be employed for TD-learning approaches. However, due to the high spatial complexity, Tabular representations proved inefficient.

Another solution to our problem might be deep Q-learning. The features are calculated using a neural network in a deep form of approximate Q-learning. Despite the fact that it operates with continuous data, it models the probability distribution of discrete actions, necessitating the binning of the action space.

For our scenario, Actor-Critic, a policy gradient approach, might have also been employed. It performs effectively in areas where continuous control is required. However, the intended Q-value and present Q-value are both created by the same network, which is a huge disadvantage.The calculated TD error becomes inconsistent as a result of inconsistent weight changes.

In recent times, the state of the art in reinforcement learning in continuous control tasks are achieved in some or the other variation of deterministic policy gradient methods, *e.g.* , Deep Deterministic Policy Gradient, Advantage Actor Critic (A2C), Asynchronous Advantage Actor Critic (A3C), Twin delayed deep deterministic policy gradient (TD3) etc.,.

Background

The tasks in the environment can be primarily associated with Locomotion, although few overlap with basic and hierarchical task as well, *i.e.*, , Locomotion + Food collection.

Ant.

The task is to make a 3-dimensional four-legged robot walk.

XML description The ant has a spherical torso, with each of its four legs constituting of three capsule mesh geoms, connected by two hinge joints. The four legs are connected to the "torso" by four free joints.

```
World Body
└─ Torso
   └─ Front left leg
      └─ Hip 1
```

Ant-v2

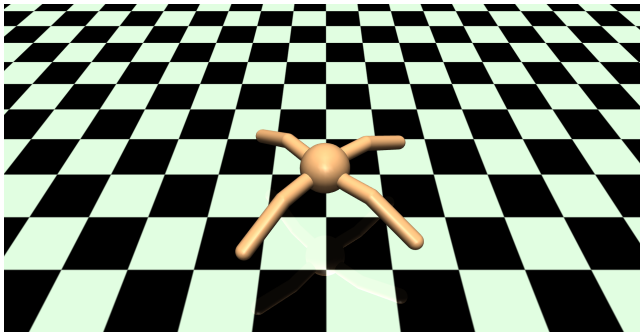

Figure 1: Four-legged ant navigating the environment

```
      └─ Ankle 1
   └─ Front right leg
      └─ Hip 2
         └─ Ankle 2
   └─ Back leg
      └─ Hip 3
         └─ Ankle 3
   └─ Right back leg
      └─ Hip 4
         └─ Ankle 4
Actuator
└─ Hip 1
└─ Ankle 1
└─ Hip 2
└─ Ankle 2
└─ Hip 3
└─ Ankle 3
└─ Hip 4
└─ Ankle 4
```

State space. Has a shape of (111,).

```python
def _get_obs(self):
    return np.concatenate([
        self.sim.data.qpos.flat[2:],
        self.sim.data.qvel.flat,
        np.clip(self.sim.data.cfrc_ext, -1, 1).flat,
    ])
```

- **self.sim.data.qpos** are the positions, with the first 7 element being the 3D position (x,y,z) and orientation (quaternion x,y,z,w) of the torso, and the remaining 8 positions being the joint angles.
- The **[2:], operation** removes the first 2 elements from the position *i.e.*, the X and Y position of the agent's torso.
- **self.sim.data.qvel** are the velocities, with the first 6 elements being the 3D velocity (x,y,z) and 3D angular velocity (x,y,z) and the remaining 8 are the joint velocities.
- The **cfrc_ext** are the external forces (force x,y,z and torque x,y,z) applied to each of the links at the center of

mass. This is 14 * 6: the ground link, the torso link plus the 12 links for all legs (3 links for each leg).

Action space. Has a shape of (8,), translating directly as torque upon the 8 hinge joint actuators (2 for each leg).

Rewards. The rewards are represented as:

```
ctrl_cost = self.control_cost(action)
contact_cost = self.contact_cost

forward_reward = x_velocity
healthy_reward = self.healthy_reward

rewards = forward_reward + healthy_reward
costs = ctrl_cost + contact_cost

reward = rewards - costs
```

- Episodic reward is calculated by inflicting a cost on the total reward for the ant.
- One of the cost is a control cost for taking actions in the environment. Another is directly proportional to how many contacts the ant makes with the ground.
- This cost is deducted from the summed reward for moving forward and for being upright most of the time.

HalfCheetah.

The task is to make a 2-dimensional cheetah robot run.

HalfCheetah-v2

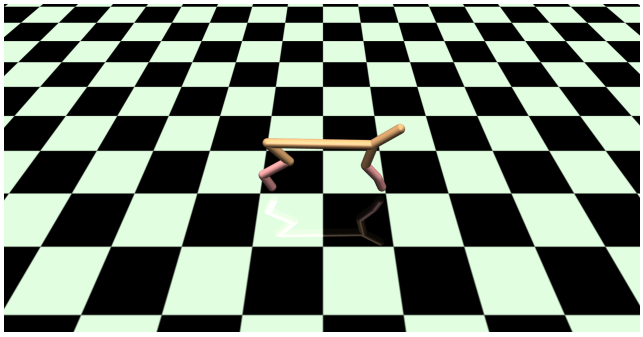

Figure 2: Two-legged cheetah running in the environment

XML description The head and torso of the HalfCheetah are both capsule mesh geoms. Each thigh, shin and feet are capsules with a hinge joint connecting them together. This is obviously true for both the front and the back legs.

```
World Body
  Head
  Torso
    Back thigh
      Back shin
        Back foot
    Front thigh
      Front shin
        Front foot
Actuator
  Back thigh
  Back shin
  Back foot
  Front thigh
  Front shin
  Front foot
```

State space Has a shape of (17,), as position and velocity for the slider joints, and angle and angular velocities for the hinge joints (3 for each leg, 3 axes for body).

Name	Joint	Parameter
rootx	slider	position (m)
rootz	slider	position (m)
rooty	hinge	angle (rad)
bthigh	hinge	angle (rad)
bshin	hinge	angle (rad)
bfoot	hinge	angle (rad)
fthigh	hinge	angle (rad)
fshin	hinge	angle (rad)
ffoot	hinge	angle (rad)
rootx	slider	velocity (m/s)
rootz	slider	velocity (m/s)
rooty	hinge	angular velocity (rad/s)
bthigh	hinge	angular velocity (rad/s)
bshin	hinge	angular velocity (rad/s)
bfoot	hinge	angular velocity (rad/s)
fthigh	hinge	angular velocity (rad/s)
fshin	hinge	angular velocity (rad/s)
ffoot	hinge	angular velocity (rad/s)

Action space Has a shape of (6,), translating directly as torque upon the 2 hinge joint actuators (3 for each leg).

Name	Actuator	Parameter
bthigh	hinge	torque (Nm)
bshin	hinge	torque (Nm)
bfoot	hinge	torque (Nm)
fthigh	hinge	torque (Nm)
fshin	hinge	torque (Nm)
ffoot	hinge	torque (Nm)

Rewards. The rewards are represented as:

```
def control_cost(self, action):
    control_cost = self._ctrl_cost_weight * np.sum(np.
     square(action))
    return control_cost

ctrl_cost = self.control_cost(action)
forward_reward = self._forward_reward_weight *
     x_velocity
reward = forward_reward - ctrl_cost
```

Humanoid.

The task is to make a 3-dimensional two-legged robot walk.

Humanoid-v2

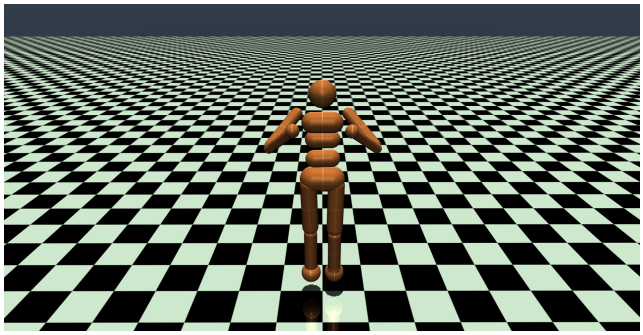

Figure 3: Two-legged humanoid learning how to walk

XML description The head, torso and uwaist are sphere and two capsule geom meshes respectively. Also, conjoined are the left arm, right arm and the lower waist. Pelvis is a part of the lower waist which have the legs connected as a hinge joint. Both the 2 legs and 2 arms have 2 hinge joints each, responsible for moving both the lower arm and hand, and both the shin and foot respectively.

```
World Body
├── Head
├── Torso
├── Right upper arm
│   └── Right lower arm
│       └── Right hand
├── Left upper arm
│   └── Left lower arm
│       └── Left hand
├── Upper waist
├── Lower waist
│   └── Pelvis
│       ├── Right thigh
│       │   └── Right shin
│       │       └── Right foot
│       └── Left thigh
│           └── Left shin
│               └── Left foot
Tendon
├── Left hip
├── Left knee
├── Right hip
└── Right knee
Actuator
├── Abdomen Y
├── Abdomen Z
├── Abdomen X
├── Right hip X
├── Right hip Z
├── Right hip Y
├── Right knee
└── Left hip X
├── Left hip Z
├── Left hip Y
├── Left knee
├── Right shoulder 1
├── Right shoulder 2
├── Right elbow
├── Left shoulder 1
├── Left shoulder 2
└── Left elbow
```

State space. Has a shape of (376,).

```python
def _get_obs(self):
    position = self.sim.data.qpos.flat.copy()
    velocity = self.sim.data.qvel.flat.copy()

    com_inertia = self.sim.data.cinert.flat.copy()
    com_velocity = self.sim.data.cvel.flat.copy()

    actuator_forces = self.sim.data.qfrc_actuator.flat.
     copy()
    external_contact_forces = self.sim.data.cfrc_ext.
     flat.copy()

    if self._exclude_current_positions _from_observation
     :
        position = position[2:]

    return np.concatenate(
        (
            position,
            velocity,
            com_inertia,
            com_velocity,
            actuator_forces,
            external_contact_forces,
        )
    )
```

- **self.sim.data.qpos** are the positions, with the first 7 element being the 3D position (x,y,z) and orientation (quaternion x,y,z,w) of the torso, and the remaining 8 positions being the joint angles.
- The **[2:], operation** removes the first 2 elements from the position *i.e.*, the X and Y position of the agent's torso.
- **self.sim.data.qvel** are the velocities, with the first 6 elements being the 3D velocity (x,y,z) and 3D angular velocity (x,y,z) and the remaining 8 are the joint velocities.
- The **cfrc_ext** are the external forces (force x,y,z and torque x,y,z) applied to each of the links at the center of mass. This is 14 * 6: the ground link, the torso link plus the 12 links for all legs (3 links for each leg).
- **qfrc_actuator** are likely the actuator forces. **cinert** seems the center of mass based inertia and **cvel** the center of mass based velocity.

Action space. Has a shape of (17,), translating directly as torque upon the 17 hinge joint actuators listed in the tree.

Reward. Represented same as for the Ant agent.

InvertedDoublePendulum

The task is to balance a pole on a pole, on a cart.

InvertedDoublePendulum-v2

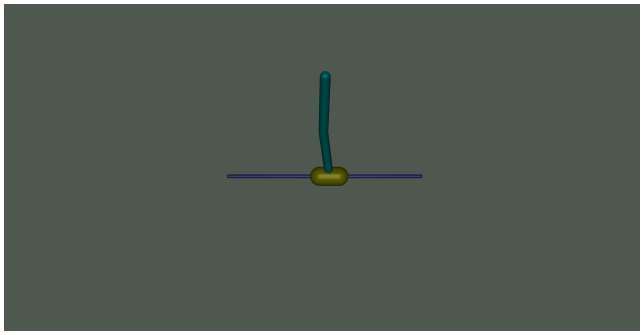

Figure 4: Cart balancing the pole in the environment

XML description The root of this model is a capsule geom on a rail of joint type slide. It is connected to a pole of geom type capsule which in turn, is connected to another pole of geom type capsule via a hinge joint.

```
World body
└── Cart
    └── Pole
        └── Tip
Actuator
└── Slide
```

State space. Has a shape of (11,), represented as slider position and velocity for the cart, and angle and angular velocity for the two pole joints.

```
def _get_obs(self):
    return np.concatenate(
        [
            self.sim.data.qpos[:1],
            np.sin(self.sim.data.qpos[1:]),
            np.cos(self.sim.data.qpos[1:]),
            np.clip(self.sim.data.qvel, -10, 10),
            np.clip(self.sim.data.qfrc_constraint, -10, 10),
        ]
    ).ravel()
```

Name	Joint	Parameter
cart	slider	position (m)
pole	hinge	angle (rad)
cart	slider	velocity (m)
pole	hinge	angular velocity (rad/s)

Action space. Has a shape of (1,), represented as force in X-axis on the cart, resulting it to translate on the rail.

Name	Actuator	Parameter
cart	motor	force x (N)

Reward. The rewards are represented as:

```
dist_penalty = 0.01 * x ** 2 + (y - 2) ** 2
v1, v2 = self.sim.data.qvel[1:3]
vel_penalty = 1e-3 * v1 ** 2 + 5e-3 * v2 ** 2
alive_bonus = 10
r = alive_bonus - dist_penalty - vel_penalty
```

Reacher.

The task is to make a 2-dimensional robot reach to a randomly located target.

Reacher-v2

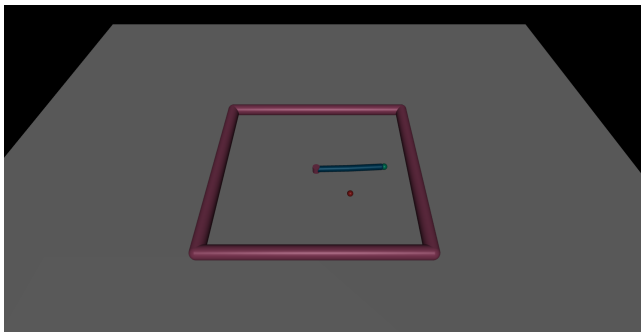

Figure 5: Robot trying to reach the randomly generated target in the environment

XML description The "arm" of the agent consists of two rigid capsule geoms connected via hinge joint and ending in a sphere geom, termed the "fingertip". The environment also contains the target as a spherical geom which the agent expects to reach to and also, four capsule geom "arena" which does not directly affect the agent's performance.

```
World body
├── Body 0
│   └── Body 1
│       └── Fingertip
└── Target
Actuator
├── Joint 0
└── Joint 1
```

State space. Has a shape of (11,).

```
def get_body_com(self, body_name):
    return self.data.get_body_xpos(body_name)

def _get_obs(self):
    theta = self.sim.data.qpos.flat[:2]
    return np.concatenate(
        [
            np.cos(theta),
            np.sin(theta),
            self.sim.data.qpos.flat[2:],
            self.sim.data.qvel.flat[:2],
            self.get_body_com("fingertip") - self.get_body_com("target"),
```

```
        ]
    )
```

Name	Joint	Parameter
joint0	hinge	angle (rad)
joint1	hinge	angle (rad)
joint0	hinge	angular velocity (rad/s)
joint1	hinge	angular velocity (rad/s)
target	slider	position (m)

Action space. Has a shape of (2,), represented as torque on the two joints, resulting in the agent reaching the target.

Name	Actuator	Parameter
joint0	motor	torque (Nm)
joint1	motor	torque (Nm)

Reward. The rewards are represented as:

```
vec = self.get_body_com("fingertip") - self.get_body_com
    ("target")
reward_dist = -np.linalg.norm(vec)
reward_ctrl = -np.square(a).sum()
reward = reward_dist + reward_ctrl
```

Swimmer.

The task is to make a 2-dimensional robot swim.

Swimmer-v2

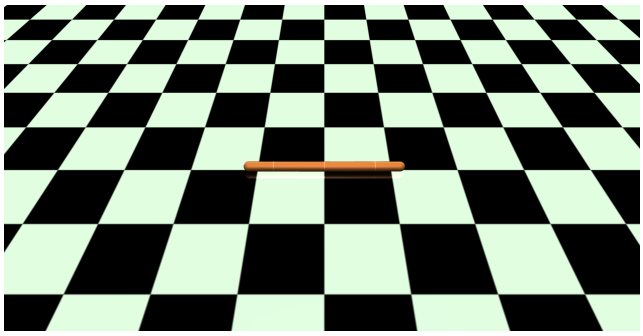

Figure 6: Robot trying to swim in the environment

XML description. The torso of the agent has two rigid capsule geoms, connected over two hinge joints.

```
World body
└── Torso
    └── Mid
        └── Back
Actuator
├── Rot 2
└── Rot 3
```

State space. Has a shape of (8,).

```
def _get_obs(self):
    position = self.sim.data.qpos.flat.copy()
    velocity = self.sim.data.qvel.flat.copy()

    if self._exclude_current_positions_from_observation:
        position = position[2:]

    observation = np.concatenate([position, velocity]).
     ravel()
    return observation
```

Name	Joint	Parameter
slider1	slide	position (m)
slider2	slide	position (m)
slider1	slide	velocity (m/s)
slider2	slide	velocty (m/s)
rot2	hinge	angle (rad)
rot3	hinge	angle (rad)
rot2	hinge	angular velocity (rad/s)
rot3	hinge	angular velocity (rad/s)

Action space. Has a shape of (2,), represented as torque on the two joints, resulting in the agent "swimming".

Name	Actuator	Parameter
rot2	motor	torque (Nm)
rot3	motor	torque (Nm)

Reward. The rewards are represented as:

```
def control_cost(self, action):
    control_cost = self._ctrl_cost_weight * np.sum(np.
     square(action))
    return control_cost

xy_position_before = self.sim.data.qpos[0:2].copy()
xy_position_after = self.sim.data.qpos[0:2].copy()

xy_velocity = (xy_position_after - xy_position_before) /
      self.dt
x_velocity, y_velocity = xy_velocity

forward_reward = self._forward_reward_weight *
     x_velocity

ctrl_cost = self.control_cost(action)
reward = forward_reward - ctrl_cost
```

Hopper.

The task is to make a 2-dimensional robot hop.

XML description. The torso of the agent sequences a single thigh, a single leg and a single foot, all of them being mesh capsule geoms, and all connected via a hinge joint.

```
World Body
└── Torso
    └── Thigh
        └── Leg
            └── Foot
Actuator
└── Thigh joint
```

Hopper-v2

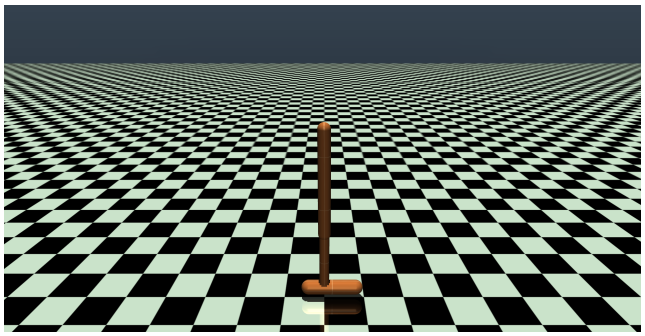

Figure 7: One-legged robot learning to hop

```
Leg joint
Foot joint
```

State space. Has a shape of (11,).

```
def _get_obs(self):
    position = self.sim.data.qpos.flat.copy()
    velocity = np.clip(self.sim.data.qvel.flat.copy(),
     -10, 10)

    if self._exclude_current_positions_from_observation:
        position = position[1:]

    observation = np.concatenate((position, velocity)).
     ravel()
    return observation
```

Name	Joint	Parameter
rootx	slider	position (m)
rootz	slider	position (m)
rooty	hinge	angle (rad)
thigh	hinge	angle (rad)
leg	hinge	angle (rad)
foot	hinge	angle (rad)
rootx	slider	velocity (m/s)
rootz	slider	velocity (m/s)
rooty	hinge	angular velocity (rad/s)
thigh	hinge	angular velocity (rad/s)
leg	hinge	angular velocity (rad/s)
foot	hinge	angular velocity (rad/s)

Action space. Has a shape of (3,), represented as torque on the three joints, resulting in the agent reaching the target.

Name	Actuator	Parameter
thigh_joint	motor	torque (Nm)
leg_joint	motor	torque (Nm)
foot_joint	motor	torque (Nm)

Reward. The rewards are represented as:

```
x_position_before = self.sim.data.qpos[0]
x_position_after = self.sim.data.qpos[0]
x_velocity = (x_position_after - x_position_before) /
     self.dt

ctrl_cost = self.control_cost(action)

forward_reward = self._forward_reward_weight *
     x_velocity
healthy_reward = self.healthy_reward

rewards = forward_reward + healthy_reward
reward = rewards - ctrl_cost
```

Project description

Online Value-Based Methods

"Bootstrapping" in reinforcement learning means that the estimate of one state $V_\pi(s)$ builds upon the estimate of successor states $V_\pi(s')$. Dynamic programming uses bootstrapping and is a model-based learning. Other methods do not rely on bootstrapping and are known as model-free methods like Monte-Carlo. Temporal difference learning combines Monte-Carlo (model-free) and Dynamic programming (model-based).

The Temporal difference (TD) error is given by:

$$\delta_t = R_{t+1} + \gamma * V(s_{t+1}) - V(s_t)$$

δ_t: TD error
$V(s_t)$: value estimate of state 's_t'
$V(s_{t+1})$: value estimate of next state 's_{t+1}'
R_{t+1}: reward obtained on transition from 's_t' to 's_{t+1}'

SARSA. SARSA combines Generalized Policy Iteration with Temporal Difference learning to find improved policies. It uses action-values (Q-value) form of TD. The name 'SARSA' stands for $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1} \rightarrow (state, action, reward, nextstate, nextaction)$. It is an on-policy TD control method.
The update equation used by SARSA is:

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t) + \alpha * [R_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

$Q(s_t, a_t)$: action-value estimate for state s_t and action a_t
α: learning rate γ: discount factor $Q(s_{t+1}, a_{t+1})$: action-value estimate for state s_{t+1} and action a_{t+1}

In Generalized policy iteration with SARSA, we continually estimate Q_π for the behavior policy π, at the same time change π towards greediness with respect to Q_π.

Q-Learning. Q-learning is an off-policy Temporal difference control algorithm. In Q-learning, the incremental update is given by

$$Q(s_t, a_t) \Leftarrow Q(s_t, a_t) + alpha * [R_{t+1} + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

α: learning rate
γ: discount factor
$Q(s_t, a_t)$: action-value estimate for state s_t and action a_t
$Q(s_{t+1}, a_{t+1})$: action-value estimate for state s_{t+1} and action a_{t+1}

The target policy is $\pi^* = \text{argmax}_a Q(s, a)$. The term $\max_a Q(s_{t+1}, a)$ selects greedy actions irrespective of actual policy π (behavior policy).

Policy gradient

Until now, we considered action-value estimates for learning an optimal policy. As the observation and action spaces tend to grow, tabular methods prove inefficient due to the exponential growth of Q-table size, resulting in the curse of dimensionality problem. Now, we consider the class of methods that can select actions without using a value function. These are called as policy gradient methods. This method is applicable for learning optimal policies in the continuous observation space, using probability distributions over the action space.

For this, we use a parameterized policy given by $\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}$ where, $'\theta'$ is the policy's parameter vector. Like the weight parameter vector 'w' we use for approximate action-value functions $\hat{q}(s,a,w)$, here we use $'\theta$. The constraints on policy parameterization are:

$$\pi(a|s,\theta) \geq 0 \; \forall \; a \; \epsilon \; A \; and \; s \; \epsilon \; S$$

$$\Sigma_{a\epsilon A}\pi(a|s,\theta) = 1 \; \forall \; s \; \epsilon \; S$$

$\pi(a|s,\theta)$ is differentiable with respect to parameter $'\theta'$ *i.e.*, $\nabla \; \pi(a|s,\theta)$ exists. In order to satisfy these conditions, we use a "softmax policy parameterization".

$$\pi(a|s,\theta) \; = \; e^{h(a,s,\theta)} \; / \; \Sigma_{b\in A} e^{h(b,s,\theta)}$$

$h(s,a,\theta)$ is knows as parameterized numerical preferences where $h(s,a,\theta) \; \epsilon \; R$. The action with the highest preferences in each state are given the highest probabilities of being selected according to equation. Numerical preferences can be computed by a deep Artificial Neural Network, where θ is the vector of all connection weights of the network or could simply be linear in features.

$h(s,a,\theta) = \theta^T X(s,a)$ where X(s,a) is some feature vector.

The goal of RL is maximizing rewards in the long run $R_t, R_{(t+1)}, R_{(t+2)}$.... In the policy gradient case, our objective maximizing the average reward r_π hence, we use gradient ascent.

$$r(\pi) = \Sigma_s \mu(s) \Sigma_a \pi(a|s,\theta) \Sigma_{s',r} p(s',r|s,a) * r$$

$$\nabla_\theta r(\pi) = \nabla_\theta [\Sigma_s \mu(s) \Sigma_a \pi(a|s,\theta) \Sigma_{s',r} p(s',r|s,a) * r]$$

From the product rule of calculus,

$$\begin{aligned}\nabla_\theta r(\theta) = \Sigma_s \mu(s) \nabla_\theta \Sigma_a \pi(a|s,\theta) \Sigma_{s',r} p(s',r|s,a) * r \\ + \Sigma_s \nabla_\theta \mu(s) \Sigma_a \pi(a|s,\theta) \Sigma_{s',r} p(s',r|s,a) * r\end{aligned}$$

The challenge of this method lies in computing the gradient of the state distribution $\pi(s)$ as it changes with θ. To address this, we use the "policy gradient theorem" which returns a simplified expression independent of $\nabla_\theta \mu(s)$.

Deep Deterministic Policy Gradient

Earlier method worked well with discrete action spaces but fails for continuous control problems. Deep Deterministic Policy Gradient (DDPG) incorporates Deterministic Policy Gradient (DPG) into the Actor-Critic structure to extend to continuous action spaces. It relies on off-policy updates using target networks.

DDPG makes use of 4 networks in total – **actor network**, **critic network**, **target-actor network**, and **target-critic network**. The actor network computes the deterministic policy $a_t = \mu(s_t|\theta^\mu)$, where θ^μ are the weights for the actor network. However, this policy might not explore the full state and action space. To encourage exploration, it makes use of a random process called the Ornstein-Uhlenbeck Noise N_t. In a continuous setting, it is defined as:

$$dN_t = \beta * (\mu - N_t) * dt + \sigma * dW_t$$

In the discrete case,

$$N_{t+1} = (1-\beta) * N_t - \mu + \sigma * (W_{t+1} - W_t)$$

N_t : noise at time 't'
β : decay or growth rate of the system
μ : asymptotic mean
σ : variation or size of noise
W : wiener process

The Weiner process also known as Brownian motion is a stationary process with white noise increments of a noise distribution N_t with $\mu = 0$ and $\sigma = 1$.

The Critic network $Q(s_t, a_t|\theta^Q)$ evaluates state-action pairs where θ^Q are its weights. The target actor and critic network denoted by Q' and μ' with weights $\theta^{Q'}$ and $\theta^{\mu'}$ respectively are a soft copy of the weights of actor and critic network θ^μ and θ^Q respectively.

$$\theta^{Q'} \Leftarrow \theta^Q$$

$$\theta^{\mu'} \Leftarrow \theta^\mu$$

Replay Buffer R stores the transition dynamics of the environment *i.e.*, $R = \{(s_t, a_t, r_t, s_{t+1})\} \; \forall \; a_t \; \epsilon \; A \; and \; s_t \; \epsilon \; S; t \; \epsilon \; [1, M']$ where M' is the memory limit. Whenever an agent takes an action in the environment, the transition tuple (s_t, a_t, r_t, s_{t+1}) is added to the replay buffer. The objective of the critic network is minimizing the temporal difference between the target-critic network's output and the estimated Q-value from its network.

$$y_i = r_i + \gamma * Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

$$L = (1/n) * (y_i - Q(s_i, \mu(s_i|\theta^\mu)|\theta^Q))$$

$\mu'(s_{i+1}|\theta^\mu)$: estimated target-actor network's policy
$\mu(s_i|\theta^\mu)|\theta^Q)$: estimated actor network's policy
$Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$: estimated target-critic network's Q-value
$Q(s_i, \mu(s_i|\theta^\mu)|\theta^Q)$: estimated critic network's Q-value
n: number of random samples from replay buffer
L: critic loss

The objective of the actor network is to learn the optimal policy that maximizes the expected return. It uses the policy gradient to achieve its goal.

$$J(\theta) = E[Q(s,a)|s = s_t, a_t = \mu(s_t)]$$

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_(\theta^\mu) \mu(s|\theta^\mu)$$

Across 'n' mini-batch samples from replay buffer,

$$\nabla_{\theta^\mu} J(\theta) \approx (1/n) * \Sigma_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

The target networks are updated using a moving average equation with parameter 'τ', which indicates the fraction of weights carried over from the original actor-critic networks to the corresponding target networks.

$$\theta^{Q'} \Leftarrow \tau * \theta^Q + (1 - \tau) * \theta^{Q'}$$

$$\theta^{\mu'} \Leftarrow \tau * \theta^\mu + (1 - \tau) * \theta^{\mu'}$$

The original pseudo-code for DDPG is illustrated below:

Deep Deterministic Policy Gradient (DDPG)

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

Figure 8: Pseudo-code from the original paper

Experiments

All the experiments were run on a Nvidia GeForce GTX 1060 with Max-Q Design and Intel Core i7-7700HQ CPU, with a physical memory (RAM) size of 16 GB. Tabular Q-learning and SARSA (State–action–reward–state–action) were the baseline methods chosen. Our initial approach was to experiment with the performance of discrete observation and action space methods on continuous observation and control tasks. As the ranges were [-inf, inf] for each observation, we sampled across 10k observations and clipped the maximum and minimum ranges to [-25, 25]. We discretized the continuous values into 2 buckets categorized into 0, 1, for both the action and observation spaces. We varied the learning rate starting from 0.2, 0.3, ..., 0.9. Other parameters chosen were: $\gamma = 0.99$, number of episodes (epochs) = 500 and number of steps per episode = 1000. Actions were selected using an epsilon-greedy policy with $\epsilon = 0.99$ decaying at a rate of:

$$\epsilon = log_{10}((e^\epsilon + 1)/25)$$

The following curves were observed for Tabular Q-learning and SARSA.

Q-learning vs. SARSA

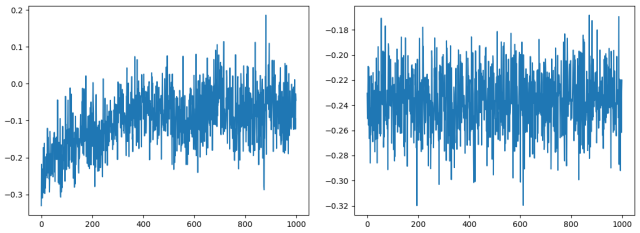

Figure 9: Average rewards for Q-learning (left) and SARSA (right) when run under the same conditional parameters in the HalfCheetah-v2 task from the MuJoCo environment.

The following observations listed are from Figure 10:

- The plot shows that Q-learning has extremely stochastic behavior, whereas SARSA exhibits more stable behavior over time. This is due to Q-learning's off-policy nature, in which the target and behavior policy are not the same.
- The plot shows that with a learning rate of 0.5, both Q-learning and SARSA acquire sub-optimal rewards, while for a learning rate of 1 they perform poorly. This may be linked back to the update equation, in which (1 - learning rate) * Q(s, a) = 0 and we rely only on greediness in Q-learning or randomness in SARSA.
- Q-learning performance improves as the learning rate rises until the learning rate reaches one. However, SARSA's performance is rather stable across all learning rates.

Q-learning vs. SARSA (Varying learning rate)

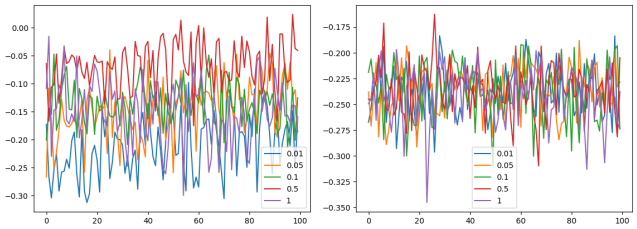

Figure 10: Average rewards for Q-learning (left) and SARSA (right) with α = [0.01, 0.05, 0.1, 0.5, 1] in the HalfCheetah-v2 task from the MuJoCo environment.

It was followed by a Deep Deterministic Policy Gradient (DDPG) network as it was known to perform well on con-

tinuous control tasks. The parameter values taken were:

$$\gamma = 0.4, \tau = 0.99, \theta = 0.15,$$
$$\mu = 0.0, \sigma = 0.3, M' = 10000, n = 100$$

The actor and critic networks were built using two different architectures. The initial architecture included two hidden layers, with the 1st and 2nd hidden layers containing 32 and 16 neurons, respectively. The other architecture used 4 hidden layers, each having 32, 64, 32, and 16 neurons for the 1st, 2nd, 3rd, and 4th hidden layers, respectively. The Adam optimizer was employed for adaptive moment estimation. Mini-batch size = n was used to train the critic network, and mini-batch size = 1 was used to train the actor network.

The number of episodes (epochs) were 10, and each episode had 1000 steps. For the former architecture [32, 16] we observe the average rewards begin with -1200 and after 10 episodes amounted to -300. The latter architecture [32, 64, 32, 16] comparatively performed much better with an initial average reward of -0.621 and converged to -0.401 at the end of 10 episodes.

Actor network (HalfCheetah-v3 task).

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	576
activation (Activation)	(None, 32)	0
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 64)	2112
activation_1 (Activation)	(None, 64)	0
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2080
activation_2 (Activation)	(None, 32)	0
dropout_2 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 16)	528
activation_3 (Activation)	(None, 16)	0
dropout_3 (Dropout)	(None, 16)	0
dense_4 (Dense)	(None, 6)	102
activation_4 (Activation)	(None, 6)	0

Critic network (HalfCheetah-v3 task).

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 17)]	0
input_1 (InputLayer)	[(None, 6)]	0
concatenate (Concatenate)	(None, 23)	0
dense_5 (Dense)	(None, 32)	768
activation_5 (Activation)	(None, 32)	0
dropout_4 (Dropout)	(None, 32)	0
dense_6 (Dense)	(None, 64)	2112
activation_6 (Activation)	(None, 64)	0
dropout_5 (Dropout)	(None, 64)	0
dense_7 (Dense)	(None, 32)	2080
activation_7 (Activation)	(None, 32)	0
dropout_6 (Dropout)	(None, 32)	0
dense_8 (Dense)	(None, 16)	528
activation_8 (Activation)	(None, 16)	0
dropout_7 (Dropout)	(None, 16)	0
dense_9 (Dense)	(None, 1)	17
activation_9 (Activation)	(None, 1)	0

DDPG rewards

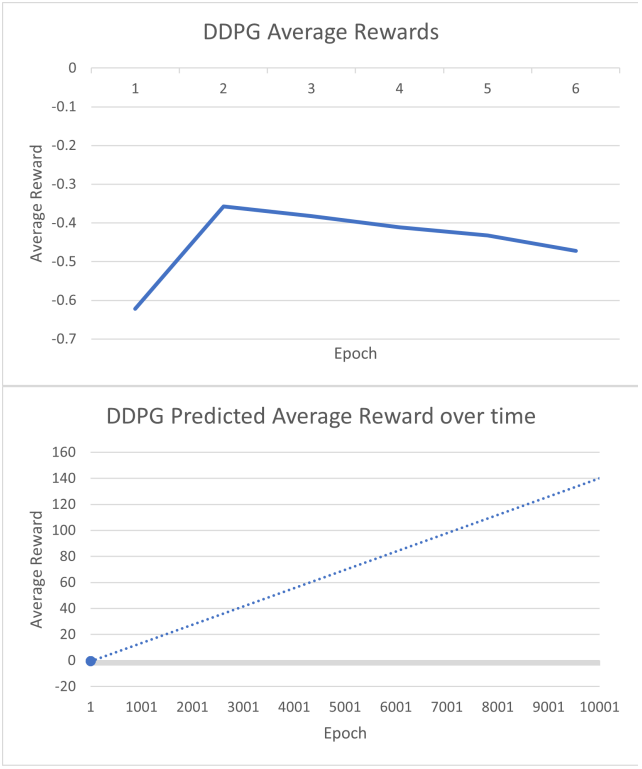

Figure 11: Current average and predicted average rewards for DDPG in HalfCheetah-v2 task with 10000+ iterations.

Improvements. The significant improvement involved few changes:

1. In contrast to the Linear activation utilized in the actor network of the former architecture, the later used a hyperbolic tangent (tanh) activation function. We assumed that a tanh activation would be meaningful because the values for the actions varied from [-1, 1], and it maps its input from that range.
2. More hidden layers were added with a 0.2 probability of dropout in the later architecture as compared to fewer layers in the former. As a result, the networks may have learned additional features to better estimate the action-value and policy. The decline in the latter phases of training can be ascribed to overfitting or a greater learning rate leading to overshooting the point of maximum average reward.

Conclusion

The average rewards received by Q-learning and SARSA for different learning rates are compared in this paper. Q-learning had somewhat better rewards than SARSA, while DDPG, a deterministic policy gradient approach, had even better outcomes than Q-learning and SARSA. There are two things that may be deduced from this.

1. Off-policy methods work better compared to on-policy methods on continuous tasks.

2. Deterministic Policy gradient methods work well in continuous control problems.

By enumerating through the replay buffer, we were able to get minibatches using a non-vectorized version of DDPG. This might be the cause of the algorithm's slowness. In the future, we want to employ vectorized implementations. Furthermore, given additional simulation time, the anticipated plot in Figure 11 shows that DDPG would eventually lead to greater and better payouts.

References

Coulom, R. 2002. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. Ph.D. thesis, Institut National Polytechnique de Grenoble.

Duan, Y.; Chen, X.; Houthooft, R.; Schulman, J.; and Abbeel, P. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control.

Erez, T.; Tassa, Y.; and Todorov, E. 2011. Infinite-Horizon Model Predictive Control for Periodic Tasks with Contacts. doi:10.15607/RSS.2011.VII.010.

Fujimoto, S.; van Hoof, H.; and Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods.

Jang, B.; Kim, M.; Harerimana, G.; and Kim, J. W. 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access* 7: 133653–133667. doi:10.1109/ACCESS.2019.2941229.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2019. Continuous control with deep reinforcement learning.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518: 529–533.

OpenAI; Akkaya, I.; Andrychowicz, M.; Chociej, M.; Litwin, M.; McGrew, B.; Petron, A.; Paino, A.; Plappert, M.; Powell, G.; Ribas, R.; Schneider, J.; Tezak, N.; Tworek, J.; Welinder, P.; Weng, L.; Yuan, Q.; Zaremba, W.; and Zhang, L. 2019. Solving Rubik's Cube with a Robot Hand.

Rummery, G.; and Niranjan, M. 1994. On-Line Q-Learning Using Connectionist Systems. *Technical Report CUED/F-INFENG/TR 166* .

Schulman, J.; Levine, S.; Moritz, P.; Jordan, M. I.; and Abbeel, P. 2017. Trust Region Policy Optimization.

Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2018. High-Dimensional Continuous Control Using Generalized Advantage Estimation.

Silver, D.; Lever, G.; Heess, N. M. O.; Degris, T.; Wierstra, D.; and Riedmiller, M. A. 2014. Deterministic Policy Gradient Algorithms. In *ICML*.

Singh, S.; Jaakkola, T.; Littman, M. L.; and Szepesvari, C. 2004. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning* 38: 287–308.

Sutton, R. S.; and Barto, A. G. 2005. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks* 16: 285–286.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033. doi:10.1109/IROS.2012.6386109.