HW3 Q-learning and Actor-Critic

Xuemin Liu
Center for Informatics and Computational Science
Department of Aerospace and Mechanical Engineering
University of Notre Dame
xliu24@nd.edu

1 Reinforcement Learning

In order to achieve the desired behavior of an agent, Reinforcement Learning (RL) proposed a formal framework in which an agent learns by interacting with an environment(Sutton, 1988) through the gathering of experience (François-lavet et al., 2018). Significant results were obtained during the following decade (Tesauro, 1995), although they were limited to low-dimensional problems.
At its core, reinforcement learning models an agent interacting with an environment and receiving observations and rewards from it, as shown in figure 2. The goal of the agent is to determine the best action in any given state in order to maximize its cumulative reward over an episode, i.e. over one instance of the scenario in which the agent takes actions.

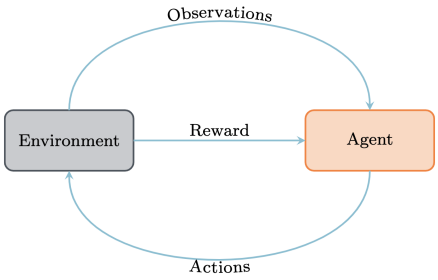


Figure 1: Markov chain.

1.1 OpenAI Gym

Although RL is a very powerful tool that has been successfully applied to problems ranging from the optimization of chemical reactions to teaching a computer to play video games, it has historically been difficult to get started with, due to the lack of availability of interesting and challenging environments on which to experiment.
OpenAI Gym is a Python package comprising a selection of RL environments, ranging from simple "toy" environments to more challenging environments, including simulated robotics environments and Atari video game environments(figure 3). It was developed with the aim of becoming a standardized environment and benchmark for RL research (Brockman et al., 2016).

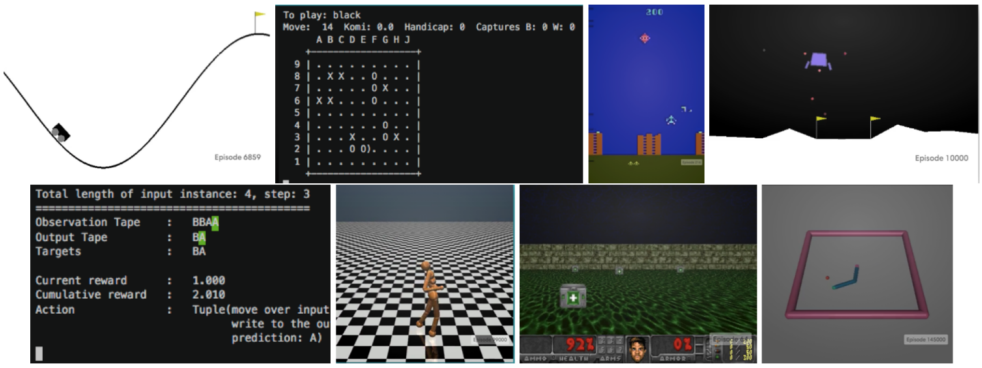


Figure 2: Images of some environments that are currently part of OpenAI Gym.

1.2 Markov Chain

A MDP can be defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathbb{P}, \mathcal{R})$ (Bellman, 1957; Howard, 1960), where:

- $\mathcal{S}$: set of non-termination states.
- $\mathcal{A}$: set of all actions.
- $\mathcal{R}$: set of all rewards.
- $\mathbb{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}^+ \to [0, 1]$, where $\mathbb{P}(s'|s, a)$ is the probability of getting to state s' from state s following action a.

For each iteration, an agent takes current state s_t, picks best (based on model prediction) action a_t and executes it on an environment. Subsequently, environment returns a reward r_{t+1} for a given action, a new state s_{t+1} and an information if the new state is terminal. The process repeats until termination.

Generally, the reward r is a signal of how 'good' a board situation s is, which helps the agent to learn distinguish more promising from less attractive decision trajectories. A trajectory is a sequence of states and actions experienced by the agent:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{1}$$

The cumulative reward (i.e. the quantity to maximize) is expressed along a trajectory, and includes a discount factor $\gamma \in [0, 1]$ that smoothes the impact of temporally distant rewards (it is then called discounted cumulative reward):

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t) \tag{2}$$

In the formalism of MDP, the goal is to find a policy $\pi(s)$ that maximizes the expected reward $\mathbb{E}[\mathcal{R}(\tau)]$. However, in the context of RL, $\mathbb{P}$ and $\mathcal{R}$ are unknown to the agent, which is expected to come up with an efficient decisional process by interacting with the environment. The way the agent induces this decisional process classifies it either in value-based or policy-based methods.

2 Method

2.1 Value-based method

In value-based methods, the agent learns to optimally estimate a value function, which in turn dictates the policy of the agent by selecting the action of the highest value. One usually defines the state value function

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[r(\tau)|s] \tag{3}$$

and the state-action value function:

$$Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[r(\tau)|s, a] \tag{4}$$

which respectively denote the expected discounted cumulative reward starting in state s (resp. starting in state a and taking action a) and then follow trajectory τ according to policy π. It is fairly straightforward to see that these two concepts are linked as follows:

$$V^{\pi}(s,a) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] \tag{5}$$

meaning that in practice, $V^{\pi}(s)$ is the weighted average of$Q^{\pi}(s,a)$ over all possible actions by the probability of each action. One of the main value-based methods in use is called Q-learning, as it relies on the learning of the Q-function to find an optimal policy. In classical Q-learning, the Q-function is stored in a Q-table, which is a simple array representing the estimated value of the optimal Q-function $Q^*(s,a)$ for each pair $(s,a) \in \mathcal{S}\times\mathcal{A}$. The Q-table is initialized randomly, and its values are progressively updated as the agent explores the environment, until the Bellman optimality condition (Bellman & Dreyfus, 1962) is reached:

$$Q^*(s,a) = r(s,a) + \gamma \max_{a'} Q^*(s',a') \tag{6}$$

where (s',a') is state-action pair of next step.The Bellman equation indicates that the Q-table estimate of the Q-value has converged and that systematically taking the action with the highest Q-value leads to the optimal policy. In practice, the expression of the Bellman equation (Bellman & Dreyfus, 1962) is used to update the Q-table estimates.

2.2 Deep Q-networks

To actually learn Q^{π}, we can use many function approximators such as linear combinations of features, neural network, decision tree, nearest neighbour, fourier/wavelet bases. Large state and/or action spaces make it intractable to learn Q value estimates for each state and action pair independently. Therefore, the neural network is accepted widely due to its strong representation capacity. To map $\mathcal{S}\times\mathcal{A}\rightarrow\mathbb{R}$, deep Q-network (DQN) provides an estimate of the Q-value for each possible action given an input state. The parameters of the neural network are optimized by using stochastic gradient descent to minimize the loss based on the Bellman optimality equation(6):

$$L(\theta) = \left[\frac{1}{2}\left(r(s,a) + \gamma \max_{a'} Q_{\theta}(s',a') - Q_{\theta}(s,a)\right)\right]^2 \tag{7}$$

where $Q_{\theta}(s,a)$ represents the Q-value estimate provided by the DQN for action s and state a under network parameterization θ; $r(s,a) + \gamma \max_{a'} Q_{\theta}(s',a')$ is the target which appears in the Bellman optimality equation (6). When the optimal set of parameters θ^* is reached, $Q_{\theta}(s,a)$ is equal to the target, and $L(\theta)$ is equal to zero.

In vanilla deep Q-learning (and more generally in Q-learning), an exploration/exploitation trade-off must be implemented, to ensure a sufficient exploration of the environment. To do so, a parameter$\epsilon \in [0,1]$ is defined, and a random value is drawn in the same range before each action. If this value is below ϵ, a random action is taken. Otherwise, the algorithm follows the action prescribed by $\max_a Q_{\theta}(s_t,a)$. Most often, ϵ follows a schedule, starting with high values at the beginning of learning, and progressively decreasing. The vanilla algorithm using DQN is shown below :

Algorithm 1: Vanilla deep Q-learning

Initialize action-value function Q parameters θ_0 ;
for *episode* $i = 1, 2, ..., N$ **do**
  Initialize state $\mathcal{S}$; **for** $t = 1, 2, ..., T$ **do**
    Draw random value $\omega \in [0,1]$ **if** $\omega < \epsilon$ **then**
      Choose action a_t randomly
    else
      Choose action $a_t = \max_a Q_{\theta}(s_t,a)$
    end
    Execute action a_t and observe reward r_t and state s_{t+1}
    Calculate target $y_t = r_t + \max_a Q_{\theta}(s_{t+1},a)$
    Update $\theta \leftarrow \theta + \alpha\nabla_{\theta} y_t - Q_{\theta}(s_t,a))^2$
  end
end

To avoid forgetting about its past learning, a replay buffer can be created that contains random previous experiences. This buffer is regularly fed to the network as learning material, so previously acquired behaviors are not erased by new ones. This improvement also reduces the correlation between experiences: as the replay buffer is randomly shuffled, the network experiences past (s_t, a_t, r_t, s_{t+1})tuples in a different order, and is therefore less prone to learn correlation between these (Tsitsiklis & Van Roy, 1997; Lin, 1993).

Algorithm 2: Deep Q-learning with experience replay and fixed target

Initialize action-value function Q parameters θ_0, target action-value function $\hat{Q}$ parameters θ_0^-
Initialize replay memory D to capacity M
for *episode* $i = 1, 2, ..., N$ **do**
  Initialize state $\mathcal{S}$; **for** $t = 1, 2, ..., T$ **do**
    Draw random value $\omega \in [0, 1]$
    if $\omega < \epsilon$ **then**
      Choose action a_t randomly
    else
      Choose action $a_t = \max_a Q_\theta(s_t, a)$
    end
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D
    if *episode terminates at step* $t + 1$ **then**
      $y_t = r_t$
    else
      $y_t = r_t + \max_a Q_{\theta^-}(s_{t+1}, a)$
    end
    Update $\theta \leftarrow \theta + \alpha \nabla_\theta (y_t - Q_\theta(s_t, a))^2$
    Every F steps reset $\hat{Q} = Q$ by $\theta^- \leftarrow \theta$
  end
end

Algorithm 3: Double Deep Q-learning with experience reply and fixed target

Initialize action-value function Q parameters θ_0, target action-value function $\hat{Q}$ parameters θ_0^-
Initialize replay memory $\mathcal{D}$ to capacity M
for *episode* $i = 1, 2, ..., N$ **do**
  Initialize state s_0
  for $t = 0, 1, ..., T$ **do**
    Draw random value $\omega \in [0, 1]$
    if $\omega < \epsilon$ **then**
      Choose action a_t randomly
    else
      Choose action $a_t = \max_a Q_\theta(s_t, a)$
    end
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D
    if *episode terminates at step* $t + 1$ **then**
      Calculate target $y_t = r_t$
    else
      Calculate target $y_t = r_t + Q_{\theta^-}(s_{t+1}, \text{argmax}_a Q_\theta(s_{t+1}, a))$
    end
    Update $\theta \leftarrow \theta + \alpha \nabla_\theta (y_t - Q_\theta(s_t, a))^2$
    Every F steps reset $\hat{Q} = Q$ by $\theta^- \leftarrow \theta$
  end
end

Conventional Q-learning is affected by an overestimation bias, due to the maximization step in Equation 1, and this can harm learning. **Double Q-learning** (van Hasselt 2010), addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation. It is possible to effectively combine this with DQN (van Hasselt, Guez, and Silver 2016) as shown in Algorithm 2. This change was shown to reduce harmful overestimations that were present for DQN, thereby improving performance.

2.3 Actor Critic

As mentioned in section 2.2, actor-critic methods propose to simultaneously exploit two networks to improve the learning performance. One of them is policy-based $\pi_\theta(a|s)$, while the other one is value-based $V_w(s)$.Both these networks are updated in parallel, in a time-difference (TD) fashion (one update at each time station) instead of the usual Monte-Carlo configuration (one update at the end of the episode). In practice, the advantage function $A(s,a)$ is preferred to the Q-value for its lower variability. In order to avoid having to evaluate two value functions, the advantage function is usually approximated, using the reward obtained by the actor after action at and the value evaluated by the critic in state s_{t+1}:

$$A^\pi_w(s_t,a_t) \approx r(s_t,a_t) + V^\pi_w(s_{t+1}) - V^\pi_w(s_t) \tag{8}$$

The vanilla actor-critic process then goes as follows:

Algorithm 4: online Actor-Critic

initialize actor network π_θ parameters θ_0 and critic network V^π_w parameters w_0
for *episode* $i = 1, 2, ..., N$ **do**
  Initialize state s_0
  for $t = 0, 1, ..., T$ **do**
    Sample action $a_t \sim \pi_\theta$
    Execute action a_t and observe reward r_t and state s_{t+1}
    if *episode terminates at step* $t+1$ **then**
      Calculate target $y_t = r_t$
    else
      Calculate target $y_t = r_t + \gamma V^\pi_w(s_{t+1})$
    end
    Update critic $w \leftarrow w + \alpha_w \nabla_w (y_t - V^\pi_w(s_t))^2$
    Calculate advantage $A^\pi_w(s_t,a_t) = r(s_t,a_t) + V^\pi_w(s_{t+1}) - V^\pi_w(s_t)$
    Update actor $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi_w(s_t,a_t)$
  end
end

However, online update of actor network and critic network is computational costly and has high variance. A batch of tuples (s_t, a_t, r_t, s_{t+1}) will work better.Therefore, we can sample several trajectories before the updates. In Monte carlo policy gradient method, we use true return as the evaluation for the state which is unbiased and reliable. While, in the actor-critic method, we replace it with the advantage function as the evaluation for the state. Thus, the value of advantage is of key importance in this method. We can update critic network multiple times before the updating actor network to have a better estimate of the value function. In total, we'll take $K \times M$ gradient update

steps for critic network in the following algorithm:

Algorithm 5: batch Actor-Critic

initialize actor network π_θ parameters θ_0 and critic network V_w^π parameters w_0
for *iteration* $j = 1, 2, ..., J$ **do**
  Sample trajectory $\{\tau^i\} = \{s_1^i, a_1^i, ..., s_T^i, a_T^i\}$ where $i = 1, 2, ..., N$ from $\pi_\theta(a_t|s_t)$ and build array of tuples (s_t, a_t, r_t, s_{t+1})
  for $k = 1, 2, ..., K$ **do**
    if *episode terminates at step* $t + 1$ **then**
      Calculate target $y_t = r_t$
    else
      Calculate target $y_t = r_t + \gamma V_w^\pi(s_{t+1})$
    end
    for $m = 1, 2, ..., M$ **do**
      Update critic $w \leftarrow w + \alpha_w \nabla_w (y_t - V_w^\pi(s_t))^2$
    end
  end
  Calculate advantage $A_w^\pi(s_t, a_t) = r(s_t, a_t) + V_w^\pi(s_{t+1}) - V_w^\pi(s_t)$
  Update actor $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t|s_t) A_w^\pi(s_t, a_t)$
end

3 Implementation

Deep Q-learning
According to the algorithm in section 2, we'll use a neural network to approximate the Q value. The input for the network is the state of the environment. The output of the network is the Q value for state-action pair. Here, we use DQN to train Atari games. The state for this game is the snapshot fram, which is an RGB image tensor of shape (210,160,3). We input 4 previous frames as the input the network. The system was controlled by some 6 discrete actions. NN used in this experiment consists of 5 layers, with three convolutional layers and two fully connected layers. The size of the convolutional layers are shown in Figure 3. I use TensorFlow to implement our algorithm with the Adam optimizer to optimize parameters. Learning rate is set to decrease from 1e-4 to 5e-5 pieceswisely. The size of replay buffer is 10^6. Learning starts when samples in the replay buffer exceed 50000. Batch size is 32. The discount is 0.99 in this task. During training, we update the parameters in the target network every 10000 steps.

```
def atari_model(img_in, num_actions, scope, reuse=False):
    # as described in https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf
    with tf.variable_scope(scope, reuse=reuse):
        out = img_in
        with tf.variable_scope("convnet"):
            # original architecture
            out = layers.convolution2d(out, num_outputs=32, kernel_size=8, stride=4, activation_fn=tf.nn.relu)
            out = layers.convolution2d(out, num_outputs=64, kernel_size=4, stride=2, activation_fn=tf.nn.relu)
            out = layers.convolution2d(out, num_outputs=64, kernel_size=3, stride=1, activation_fn=tf.nn.relu)
        out = layers.flatten(out)
        with tf.variable_scope("action_value"):
            out = layers.fully_connected(out, num_outputs=512,         activation_fn=tf.nn.relu)
            out = layers.fully_connected(out, num_outputs=num_actions, activation_fn=None)

        return out
```

Figure 3: structure of the network

Actor-Critic
According to the requirement in the homework, we implement Actor-Critic algorithm on three different environments, which include Cart Pole, Inverted Pendulum and Half Cheetah. The introduction for these tasks are provided in Homework 2. There are two networks in the algorithm, one for policy and the other for value function. The input and output of the policy network is same as policy gradient, which are state of the environment and a distribution over the action. The structure of the network is fully connected and consists of four layers, with one input layer, two hidden layers and one output layer. The size of hidden unit is 64. For value function network, it is also a 4 layer fully connected network which are same as the policy network. The input for the value function is till states of the environment. The output, however, is the value of the corresponding state. The optimizer used to update the parameter is Adam. Training settings of three different tasks are listed in Table 1.

Table 1: Environment information and training settings

Environment	Sate	Action	Discount γ	Batch size	Episode length	Learning rate
Cart Pole-v0	$\mathbb{R}^4$	$\{-1, 1\}$	1.00	1000	-	0.005
Inverted Double Pendulum-v2	$\mathbb{R}^{11}$	$[-1, 1]$	0.95	5000	1000	0.01
Half Cheetah-v2	$\mathbb{R}^{17}$	$[-1, 1]^6$	0.90	30000	150	0.02

4 Results and discussion

4.1 Deep Q-learning experiment - Pong

We use DQN to train an Atari game, Pong, which is a table tennis sports game featuring simple two-dimensional graphics(Figure 4). For PongNoFrameskip-v4 environment, there is no frame skip between each state and there are 6 discrete actions to control and emulate game.

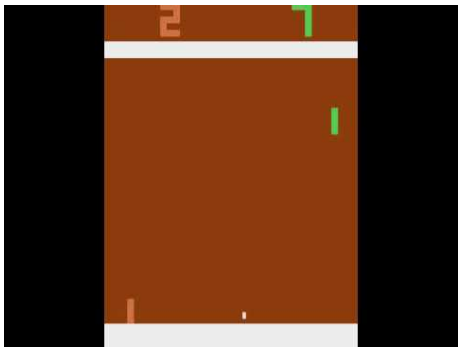
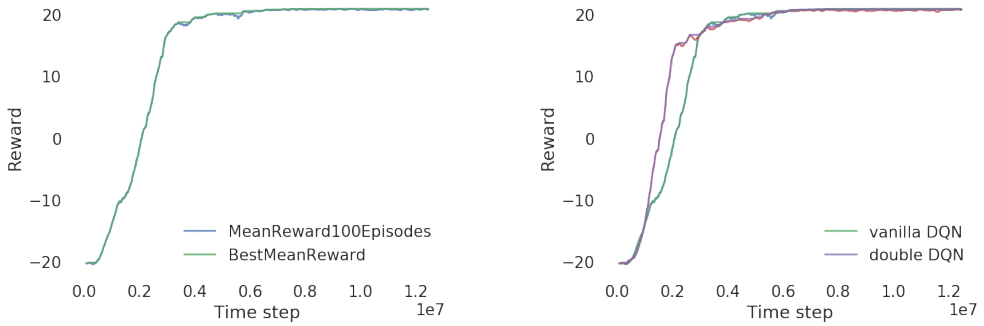


Figure 4: Pong No Frameskip-v4

According to the requirement in the homework, we first implement DQN with experience replay and fixed target and record the mean and best rewards during the training process. The performance is shown in Figure 5(a). The x-axis correspond to number of time steps and the y-axis show the mean 100-episode rewards(blue line) as well as the best mean reward(green line). These two shows a good accordance with each other.

To compare the performance of double DQN to vanilla DQN, we also implement these two algorithms on game Pong. From Figure 5(b) we can find, double DQN converges faster than vanilla DQN. Therefore, it is a better estimator of the Q values.



(a) Mean and best return using DQN

(b) DQN and double DQN

Figure 5: DQN and double DQN on Pong

To analyze the sensitivity of Q-learning to hyperparameters, we use different discount $\gamma = 0.9, 0.99, 0.999$ to train the network. The experiment result is shown in Figure 6.

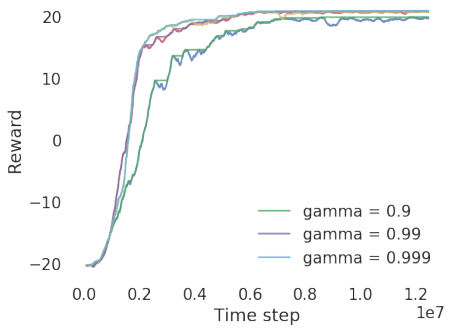


Figure 6: Sensitivity of double DQN to discount factors

4.2 Actor-Critic experiment

According to the requirement in the homework, we implement Actor-Critic algorithm on three different environments, which include Cart Pole, Inverted Pendulum and Half Cheetah. The introduction for these tasks are provided in Homework 2.

4.2.1 Cartpole

In the experiment, we first implement batch actor-critic method and alternate between performing one target update and one gradient update step for the critic. As you can see from the line labeled 1_1, this probably doesn't work. Therefore, we need to increase both the number of target updates(first number in Figure 8) and number of gradient updates(second number in Figure 8). We'll take 100 gradient update steps for critic network in one iteration in total. According to performance of four experiments in Figure8, setting both num_grad_steps_per_target_update and num_target_updates to 10 works best.

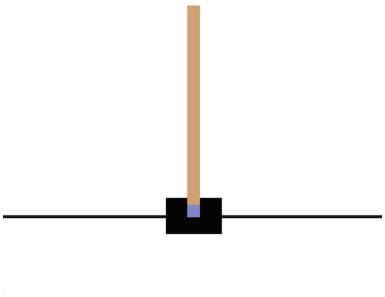


Figure 7: Cart Pole-v0

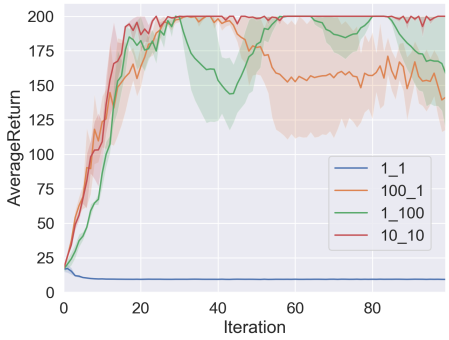


Figure 8: Cart Pole-v0

4.2.2 Inverted Pendulum and Half Cheetah

We can also implement actor-critic algorithm on more difficult tasks such as Inverted Pendulum and Half Cheetah. The environment version of these two are InvertedPendulum-v2 and HalfCheetah-v2 in Gym(MuJoCo). According to the result in Figure 8, we choose 10 both for gradient steps per target update and target update steps to obtain better performance. In total, we'll take $10 \times 10 = 100$ gradient update steps for critic network in one iteration. The training result is shown in Figure 11, which roughly match those of policy gradient (in homework 2).

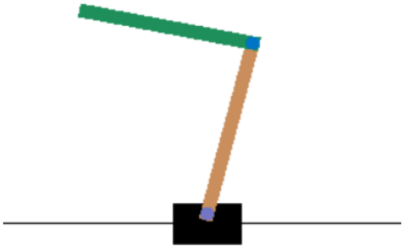


Figure 9: Inverted Double Pendulum-v2

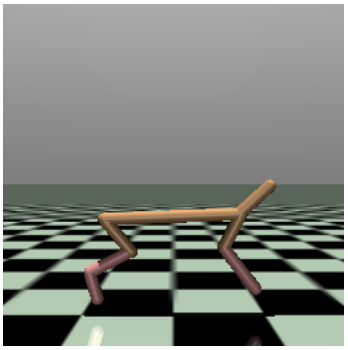


Figure 10: Half Cheetah-v2

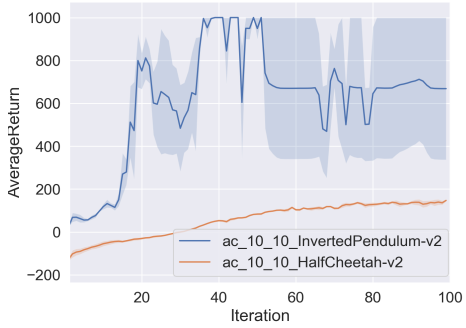


Figure 11: Return of training