HW2 Policy Gradient

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.

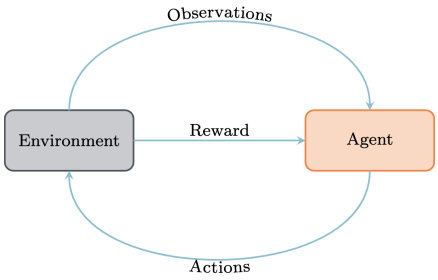


Figure 1: Markov chain.

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).

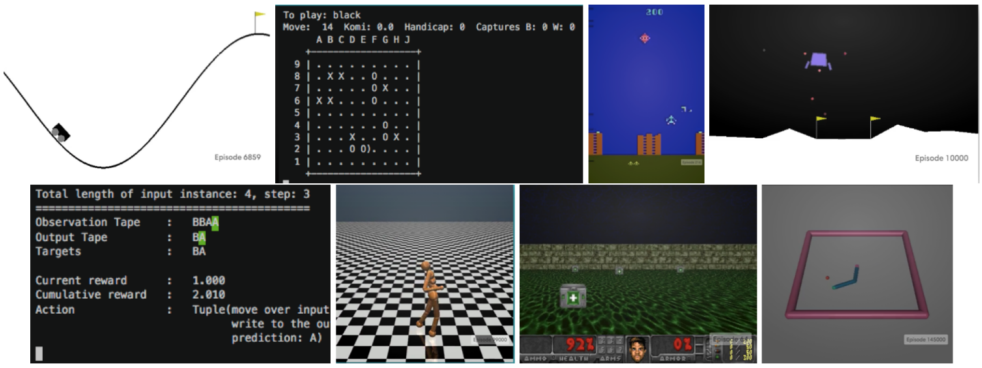


Figure 2: Images of some environments that are currently part of OpenAI Gym.

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 Policy gradient method

In RL, the policy π is a probability distribution over the action space, conditioned on the state. In the agent-environment loop, the agent samples an action a_t at time t from $\pi(\cdot|s_t)$ and the environment responds with a reward $r(s_t, a_t)$.
In policy-based method, we directly optimizing a parameterized policy $\pi_\theta(a|s)$ which maps state to action, instead of resorting to a value function estimate to determine the optimal policy. In general, policy-based methods offer three main advantages:

- They have better convergence properties, although they tend to be trapped in local minima;

- They naturally handle high dimensional action spaces;
- They can learn stochastic policies.

To determine how "good" a policy is, one defines an objective function based on the expected cumulative reward, and the learning objective is to learn a θ^* that maximizes the objective function:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] \tag{3}$$

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] \tag{4}$$

where

$$\pi_\theta(\tau) = p(s_1, a_1, ..., s_T, a_T) = p(s_1)\pi_\theta(a_1|s_1)\prod_{t=2}^{T} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t) \tag{5}$$

$$r(\tau) = r(s_1, a_1, ..., s_T, a_T) = \sum_{t=1}^{T} r(s_t, a_t) \tag{6}$$

The policy gradient approach is to directly take the gradient of the objective function $J(\theta)$ and then use gradient ascent to find the best parameter θ that improves the policy.

To that end, the gradient of the cost function is needed. This is not an obvious task at first, as one is looking for the gradient with respect to the policy parameters θ, in a context where the effects of policy changes on the state distribution are unknown. Indeed, modifying the policy will most certainly modify the set of visited states, which could affect performance in an unknown manner. This derivation is a classical exercise that relies on the log-probability trick, which allows expressing $\nabla_\theta J(\theta)$ as an evaluable expected value:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right] \tag{7}$$

where

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] = \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau \tag{8}$$

$$\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right] = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \tag{9}$$

In practice, the expectation over trajectories τ can be approximated from a batch of N sampled trajectories $\tau_i \in \{\tau_1, \tau_2, ..., \tau_N\}$ using Monte Carlo method:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) \tag{10}$$

$$= \frac{1}{N}\sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\right)\left(\sum_{t=1}^{T} r(s_t^i, a_t^i)\right) \tag{11}$$

However, the variance of Monte Carlo gradient method is very big. And there are two ways to reduce the variance in the following part.

2.1 Causality

One way to reduce the variance of the policy gradient is to exploit **causality**: the notion that the policy cannot affect rewards in the past, yielding following the modified objective, where the sum of rewards here is a sample estimate of the Q function, known as the "**reward-to-go**:"

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\left(\sum_{t'=t}^{T} r(s_{t'}^i, a_{t'}^i)\right). \tag{12}$$

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus on rewards closer in the future, which can also be thought of as a means for reducing variance (because there is more variance possible futures further into the future). We saw in lecture that the discount factor can be incorporated in two ways.

The first way applies the discount on the rewards from full trajectory:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \right) \left(\sum_{t=1}^{T} \gamma^{t-1} r(s_t^i, a_t^i) \right) \tag{13}$$

and the second way applies the discount on the "reward-to-go:"

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right). \tag{14}$$

Therefore the Monte-Carlo policy gradient method with/without "reward-to-go" can be summarized in Algorithm 1.

Algorithm 1: Monte-Carlo policy gradient w/wo "reward-to-go"

initialize π_θ
for *iteration* $k = 1, 2, ..., K$ **do**
  Sample trajectory $\{\tau^i\} = \{s_1^i, a_1^i, ..., s_T^i, a_T^i\}$ where $i = 1, 2, ..., N$ from π_θ
  if *"reward to go" is True* **then**
    $\nabla_\theta J(\theta) \approx \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right)$
  else
    $\nabla_\theta J(\theta) \approx \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \right) \left(\sum_{t=1}^{T} \gamma^{t'-1} r(s_t^i, a_t^i) \right)$
  end
  Update $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end

2.2 Baseline

Another way to reduce the variance is subtracting a baseline b which is independent to action a from the sum of rewards.

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [r(\tau) - b] \tag{15}$$

This approach leaves the policy gradient unbiased because the gradient of expectation of the baseline b equals to 0 when for a trajectory τ under policy π_θ.

$$\nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [b] = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) \cdot b] = 0 \tag{16}$$

According to the definition of value function V_ϕ^π, it can act as a *state-dependent* and *action-independent* baseline.

$$V^\pi(s_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta} [r(s_{t'}, a_{t'}) | s_t] \tag{17}$$

We can use also Monte Carlo samples as unbiased estimate of the value function and multiply a discount factor γ to focus on rewards closer in the future.

$$V^\pi(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right) \tag{18}$$

Therefore, the approximate policy gradient now looks like eqn(19) and the Monte-Carlo policy gradient method with/without "baseline" can be summarized in Algorithm 2.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right) - V^\pi \left(s_t^i \right) \right). \tag{19}$$

Algorithm 2: Monte-Carlo policy gradient w/wo baseline

initialize π_θ

for *iteration* $k = 1, 2, ..., K$ **do**

  Sample trajectory $\{\tau^i\} = \{s_1^i, a_1^i, ..., s_T^i, a_T^i\}$ where $i = 1, 2, ..., N$ from π_θ

  if *baseline is True* **then**

    $\nabla_\theta J(\theta) \approx \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right) - V^\pi \left(s_t^i \right) \right)$

  else

    $\nabla_\theta J(\theta) \approx \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \right) \left(\sum_{t=1}^{T} \gamma^{t'-1} r(s_t^i, a_t^i) \right)$

  end

  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

end

3 Implementation

According to the algorithm in section 2, we'll use a neural network to approximate the policy $\pi_\theta(a|s)$. The input for the network is the state of the environment. The output of the network is the distribution over the action. If the action is discrete variable, then policy $\pi_\theta(a|s)$ is a categorical distribution. If the action is continuous over a certain range, then output is a tuple (mean, log(standard deviation)) of a multivariate Gaussian where the dimension of this distribution equals to the dimension of action space.

Each NN 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. The shape of input an output layer is determined by the state and action of the environment and agent. I choose the **tanh** as activation function. I use TensorFlow to implement our algorithm with the Adam optimizer to optimize parameters. In each training iteration, we'll first sample several trajectories. The batch size is the minimum total time steps for these trajectories. The episode length is the maximum steps in one trajectory. We apply 100 iterations for each experiment.

In this homework, we use 4 different environment to test the policy gradient method which are Cart Pole-v0, Inverted Double Pendulum-v2, Lunar Lander Continuous-v2 and Half Cheetah-v2. Only the action of Cart Pole-v0 is discrete and all the others are continuous. The environment information and training settings of these 4 experiments 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, 5000	1000	0.005
Inverted Double Pendulum-v2	$\mathbb{R}^{11}$	$[-1, 1]$	0.90	1000	1000	0.01
Lunar Lander Continuous-v2	$\mathbb{R}^8$	$[-1, 1]^2$	0.95	40000	1000	0.005, 0.01, 0.02
Half Cheetah-v2	$\mathbb{R}^{17}$	$[-1, 1]^6$	0.90	10000, 30000, 50000	150	0.005, 0.01, 0.02

4 Results and discussion

4.1 Cartpole

Cartpole is a inverted pendulum with a center of gravity above its pivot point. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The state dimension of the system is 4 which consists of position of cart, velocity of cart, angle of pole and rotation rate of pole. The system is controlled by applying a force of +1 or -1 to the cart. Therefore the action dimension is 2. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

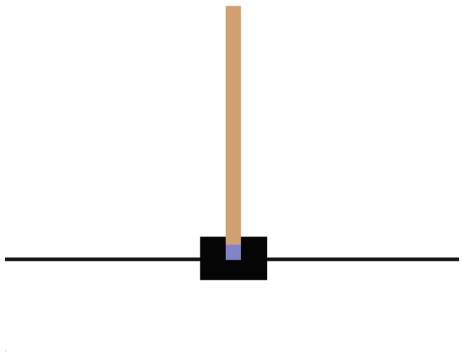
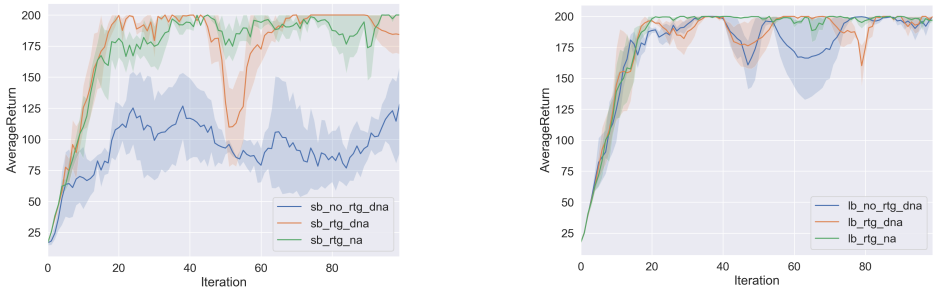


Figure 3: Cart Pole-v0

To figure out the influence of "reward-to-go" and advantage centering, we run the experiments with or without "reward-to-go" and advantage centering on a small (1000) and large (5000) batch size. The discount γ is set as 1.0. We also run with the same configuration 3 times with a different randomly initialized policy, and have a different stream of random numbers to get a better result.



(a) bash size = 1000

(b) bash size = 5000

Figure 4: Return of training

The average return(accumulative reward) for each experiment is shown in Figure 5. The solid lines are average returns of trajectories in one batch, the shadow is the largest and smallest return for one trajectory in that batch. The meaning of labels in the figure are:

- `-sb` : Small batch size.
- `-lb` : Large batch size.
- `-dna` : Flag: if present, sets `normalize_advantages` to False. Otherwise, by default, `normalize_advantages=True`.
- `-rtg` : Flag: if present, sets `reward_to_go=True`.
- `-no_rtg` : Flag: if present, sets `reward_to_go=False`.

From the figure we can answer the questions from the homework:
(1) Larger batch size helps reduce the variance((a) vs (b)).
(2) Advantage centering helps reduce the variance after convergence (orange line vs green line).
(3) Reward-to-go(green line) has better performance than the trajectory-centric one without advantage-centering(blue line); reward-to-go converges faster and has lower variance.

4.2 Inverted Pendulum

Inverted Pendulum system consists of two joint pendulums connected to a cart that is moving on a track, see Figure 5 below. The state dimension of the system is 11 which consists of position of

cart(1),velocity of cart(1), sin of two joint angles(2), cos of two joint angles(2), rotation velocity of two joint angles(2), constraint forces on cart and two joint angles(3). The system is is controlled by applying a continuous force of [-1,1]. The pendulum starts upright, and the goal is to prevent it from falling over.

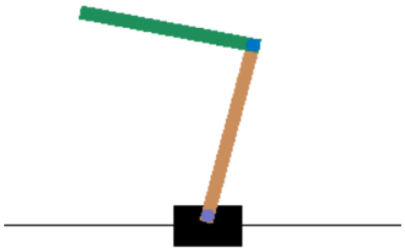


Figure 5: Inverted Double Pendulum-v2

The task is to find the smallest batch size b* and largest learning rate r* that gets to optimum (maximum score of 1000) in less than 100 iterations. (The policy performance may fluctuate around 1000 – this is fine.) The discount is 0.9 in this task. After trying, I use 1000 batch size and 0.01 learning rate to achieve the goal and the training result is shown in the Figure 6.

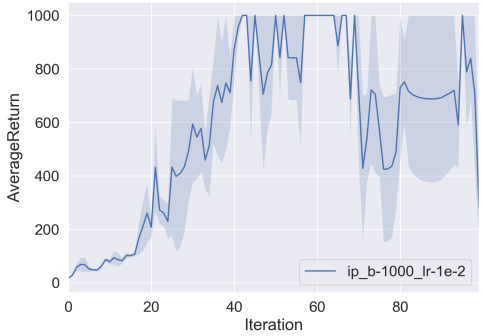
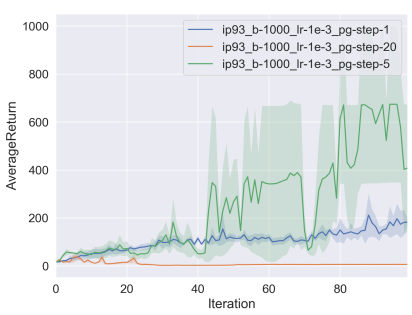
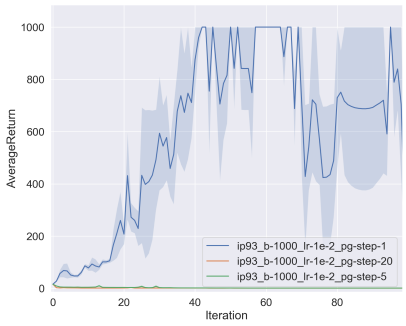


Figure 6: Inverted Double Pendulum-v2

In policy gradient(PG), we collect a batch of data, estimate a single gradient, and then discard the data and move on. To accelerate training, we explore the method by taking multiple gradient descent steps (1,5,20) with the same batch of data. When learning rate equals 0.001, 5 gradient descent step performs much better than 1 and 20 (Figure 7(a)). However, if we use a large learning rate 0.01, the multiple gradient step will have a negative influence of the training(Figure 7(b)).



(a) learning rate = 0.001



(b) learning rate = 0.01

Figure 7: Return of training

4.3 Lunar Lander

In Lunar Lander system, vehicle starts from the top of the screen (with random initial velocity) and landing pad is always at coordinates (0,0). Reward for moving from the top of the screen to landing pad and zero speed is about 100-140 points. If lander moves away from landing pad it loses reward back. Each simulation episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Firing side engine is -0.03 points each frame. Solved is 200 points. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real values vector from -1 to +1. First controls main engine, [-1,0] off, [0,+1] throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value [-1.0,-0.5] fire left engine, [+0.5,+1.0] fire right engine, [-0.5,0.5] off. The goal is to land the vehicle on the target fast, safely, and efficiently.

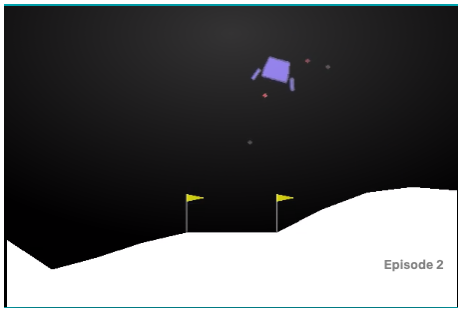


Figure 8: Lunar Lander Continuous-v2

Due to the complexity of this task, we set batch size equals to 40000. The learning rate is 0.005 and discount is 0.99. The network is trained for 100 iterations and the result is shown in Figure 9. The average return can achieve 180 and increase to 200 after 60 training iterations which satisfies the requirement of the problem.

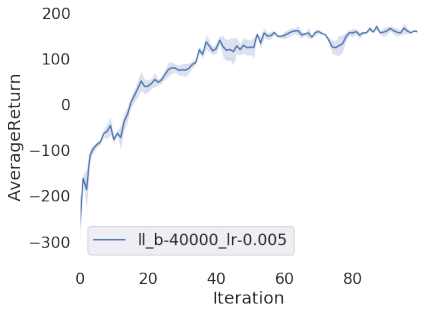


Figure 9: Return of training

4.4 Half Cheetah

Half Cheetach system is provide by MuJoCo (Multi-Joint dynamics with Contact). It is a proprietary physics engine for detailed, efficient rigid body simulations with contacts (Figure 10). The state dimension is 17 including position, velocity and forces of multiple bodies in the system. The action is forces dimension is 6. The system is controlled by applying a force in the range of [-1,1] at different bodies.

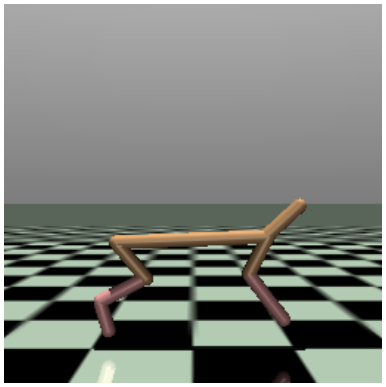
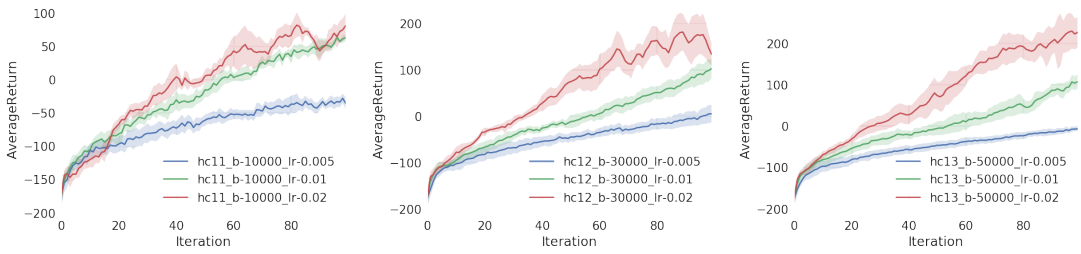


Figure 10: Half Cheetah-v2

We Use an episode length of 150, which is shorter than the default of 1000 which would speed up training significantly. To figure out the influence of batch size and learning rate, we run the experiment at batch sizes $b \in [10000, 30000, 50000]$ and learning rates $r \in [0.005, 0.01, 0.02]$. Discount is 0.95. The training results are listed in Figure 11.



(a) bash size = 10000 (b) bash size = 30000 (c) bash size = 50000

Figure 11: Return of training

Among the parameters I tested, better performance was observed for larger batch size or higher learning rate. Therefore, I chose batch size of 50000 and learning rate of 0.02 to test the influence of baseline and ”reward-to-go”. nn_baseline means we use the baseline to train the policy.

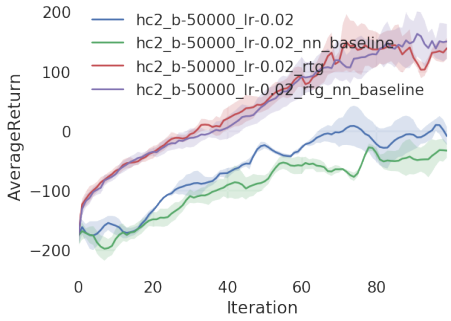


Figure 12: Return of training

From the figure above, we can find the influence of ”reward-to-go” is obvious. It can not only accelerate convergence, but also reduce the variance. The impact of baseline is not strong. However, it can also reduce the variance for models with ”reward-to-go”.