Project 5 - Pytorch and Policy Gradients Methods

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Curve Fitting

A. Using an alternate update rule

The loss function-v2 is shown in Fig. 1. We set the term $(y_{red} - y)$ without gradient.

```
def compute_loss_v2(y, y_pred):
    y_pred_constant = (y_pred-y).detach()
    return y_pred_constant*y_pred
```

Fig. 1: loss function-v2

The result is shown in Fig. 2. The estimated values are so similar to the true values. However, the loss value is larger than the loss function $(y_{pred} - y)^2$.

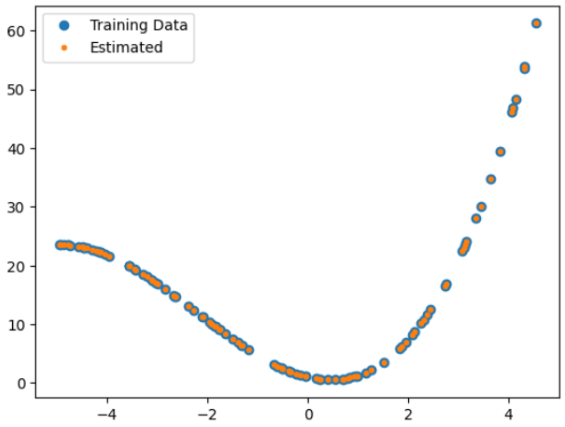

Fig. 2: results of loss function -v2

B. Fitting to 2D data

```
class FitFunction2D(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))
        self.e = torch.nn.Parameter(torch.randn(()))
        self.f = torch.nn.Parameter(torch.randn(()))

    def forward(self, xs, ys):
        return self.a+self.b*xs+self.c*ys+self.d*xs**2 \
               + self.e*ys**2 + self.f*xs*ys
```

Fig. 3: Fit Function for 2D

For 2D data, we need 6 parameters to estimate the true data, and the result is shown in Fig. 4. The training works well and the estimated data is overlapped on the provided data.

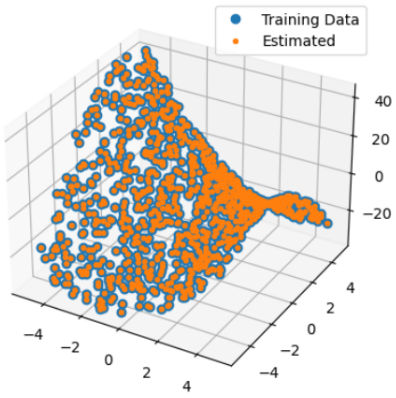

Fig. 4: Results of FitFunction2D

II. Policy Learning in the CartPole Environment

A. Defining the Policy

```
def sample_action(self, state, epsilon=0.0):
    # Move data out of PyTorch and into numpy
    policy_outputs = self.forward(state).detach().numpy()
    if policy_outputs[0] < 1e-6:
        policy_outputs[0] = 1e-6
        policy_outputs[1] = 1-1e-6
    if policy_outputs[1] < 1e-6:
        policy_outputs[0] = 1-1e-6
        policy_outputs[1] = 1e-6
    if np.random.uniform() > epsilon:
        action = np.random.choice([0,1], 1, p=policy_outputs)[0]
    else:
        action = np.random.choice([0,1], 1)[0]
    return action

def get_action_log_prob(self, state, action):
    """Return the log of the probability."""
    probs = self.linear_relu_stack(torch.tensor(state, dtype=torc
    return torch.log(probs[action])
```

Fig. 5: CartPole Policy

The two functions *sample_action* and *get_action_log_prob* are shown in Fig. 5. Function *sample_action* uses soft-policy, and all actions have non-zero probability to be selected. In my implementation, there is a problem of *nan* value. If one value of the output is too small (*1e-40*), the policy will return *nan*, and the algorithm will be broken down. To prevent this happening, we check the output of the policy, if any value is

smaller than *1e-6*, we set its value equal to *1e-6*. This makes the algorithm running smoothly without any error.

In function *get_action_log_prob*, we input the state into the network, and select the output value which is corresponding to the inputted action, *log* it then return.

Question: "Roughly how many steps does the typical trial manage before the pendulum falls to the side? Describe what behavior you should expect to keep the pendulum up.
Answer: The pendulum is at the up position from 12 to 18 steps. To keep the pendulum up, the cart needs to move to the opposite direction of the falling down direction of the pendulum. However, the movement should not be large to keep the pendulum stable.

B. Implementing REINFORCE

In this algorithm, we just learn the policy. Its loss function is defined as the return G multiply with the probability log of the selected action. Actually, in the loss function we also consider the probability of taking the action (*prob=1*). However, this value will be disappeared after taking the gradient, so we ignore it in the code (Fig. 6).

```
G = 0
loss = 0
total_reward = 0
for state, action, reward, new_state, done in reversed(rollout_data):
    total_reward += reward
    G = gamma * G + reward
    pred = policy.get_action_log_prob(state, action)
    loss -= G*pred
loss.backward()
optim.step()
optim.zero_grad()
```

Fig. 6: Reinforce Code

Question Run the Results & Plotting code a few more times (3–5 times will be sufficient) and comment on the similarities and differences between the results and behavior of the system (e.g., how does the random initial configuration impact the performance of the system?)

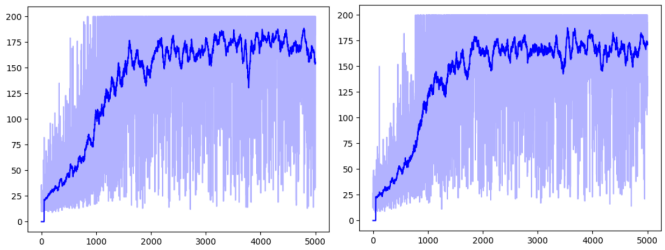

Fig. 7: Results of Reinforce Cartpole-v0 - 1

Answer: We ran four times the Baseline algorithm and the statics are shown in Fig. 7 and 8. The curves have similar shapes with slow start and reach the stability after 1800 training episodes. They also share high variance during the training. With random initial configurations, the peak can come within some hundreds of training episodes, and the average reward are fluctuated with different magnitudes.

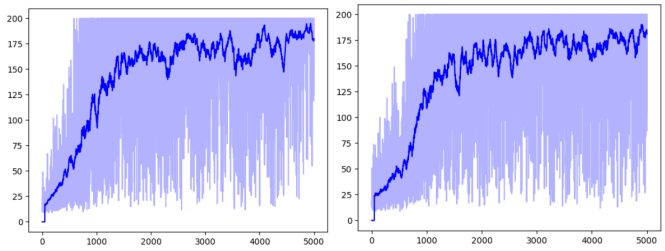

Fig. 8: Results of Reinforce Cartpole-v0 - 2

TASK/QUESTION/PLOT (3–5 sentences) Increase the learning rate to *learning_rate=0.01* and run the results and plotting code a few more times. Include one of the plots in your writeup. How does is the performance similar or different to the base configuration?

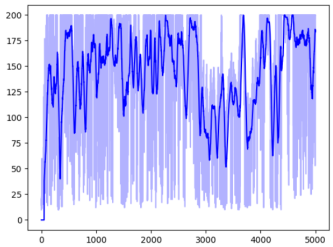

Fig. 9: Results of Reinforce Cartpole-v0 - learning rate = 1e-2 - 1

Answer: The policy sometimes can guide the pendulum reaching the optimal behavior in some short periods of time, however, the learning is not stable. The large *learning_rate* makes a huge update amount to the parameters, thus the approximate function jumps over the optimal point, making the function fluctuation around the optimal point.

TASK/QUESTION/PLOT: Decrease the learning rate to *learning_rate=1e-4* and run the results and plotting code a few more times. Include one of the plots in your writeup. How does is the performance similar or different to the base configuration?

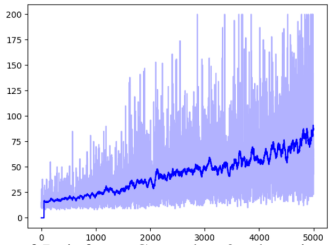

Fig. 10: Results of Reinforce Cartpole-v0 - learning rate = 1e-4

Answer: The small *learning_rate* (*1e-4*) leads to too small update to the network's parameters; as a result, the learning process is stable, however, it is very slow. As shown in Fig. 10, the reward sum is getting better and better, however, after 5000 training episodes, it reaches to 75 steps. This is much smaller to the original configurations.

C. Implementing REINFORCE with Baseline

In this algorithm, we learn both the policy and value function V in the same time. Instead of using the return,

we use the difference between the return G and the state value function V, and they will reduce the fluctuation during training. The loss function for the policy is the same. For the value function, the loss function is defined as the power 2 of the return and the current state value $(G - v)^2$. All the updating is occurred after an episode is done. To avoid the conflict between two updating of two network, we ignore the gradient of the term $(G - v)$ for the policy network.

```
for state, action, reward, new_state, done in reversed(rollout_data):
    total_reward += reward
    G = gamma * G + reward
    v = V(state)
    delta = G - v
    pred = policy.get_action_log_prob(state, action)
    loss -= delta.detach()*pred
    v_loss += delta**2
v_loss.backward()
v_optim.step()
v_optim.zero_grad()

optim.zero_grad()
loss.backward()
optim.step()
```

Fig. 11: Reinforce with Baseline

QUESTION (2–4 sentences) Run the Results & Plotting code a few more times and comment on the similarities and differences between the results and behavior of the system (e.g., how does the random initial configuration impact the performance of the system?)

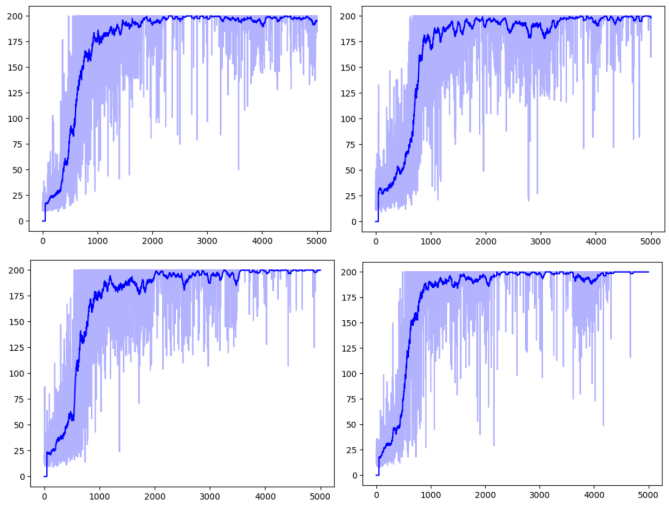

Fig. 12: Reinforce with Baseline Cartpole-v0

Answer: The results of four runs in shown in Fig. 12, and they are quite similar in the shape. The returns get peak after 1200 training episodes. The variance of the return value is smaller during the training. The algorithm helps the pendulum get the optimal values most of the time after 2000 training episodes.

The difference of the initial configurations make the convergence of the performance varying within some hundred episodes. Sometimes, there are some value jumping as the training is already stable. Some two configurations, the learning is very stable and optimal after 4000 training episodes.

QUESTION (3–5 sentences) How does the performance of REINFORCE with Baseline compare to the plain REINFORCE algorithm from P5.2.2? Comment on their relative performance. Which would you prefer to use and why?

Answer: The performance of REINFORCE with Baseline is much stabler than the REINFORCE algorithm. The former algorithm reaches the optimal value much earlier (1200 to 2000 episodes), and the variances are also much smaller. We can say REINFORCE with Baseline outperforms the REINFORCE algorithm.

D. Implementing Actor-Critic

The training loop of Actor Critic algorithm is shown in Fig. 13. Similar to REINFORCE with Baseline, we both learn the policy and the value function V. The difference is the utilizing of *TD* technique instead of *Monte Carlo* approach. So, instead of summing the loss of two networks to the episode end, then update the parameters one time, we update the networks for each time the robot taking an action. It utilizes the bootstrapping technique, and speeds up the training progress. The results are shown in Fig. 14.

```
for ind in range(num_episodes):
    state, _ = env.reset()
    done = False
    total_reward = 0
    loss = 0
    v_loss = 0
    v = 0
    while not done:
        action = policy.sample_action(state, epsilon0)
        try:
            new_state, reward, terminated, truncated, _ = env.step(action)
        except:
            new_state, reward, terminated, truncated, _ = env.step([action])
        done = terminated or truncated
        total_reward += reward
        v = V(state)
        v_new = V(new_state)
        if done:
            delta = reward - v
        else:
            delta = reward + gamma*v_new.detach() - v
        pred = policy.get_action_log_prob(state, action)
        loss = -gamma*delta.detach()*pred
        v_loss = delta**2
        state = new_state.copy()
        v_loss.backward()
        v_optim.step()
        v_optim.zero_grad()
        optim.zero_grad()
        loss.backward()
        optim.step()
```

Fig. 13: Actor Critic's code

We can see that in both runs, the learning progress quickly, and can reach the optimal value within only 800 training episodes. However, there are huge variance after training of 1000 episodes. We suspect that the fluctuation is the results of large learning rate. So, we reduce the learning rate to 0.0002 and its result is shown in Fig. 15.

The results of several runs from all three algorithms are shown in Fig. 16.

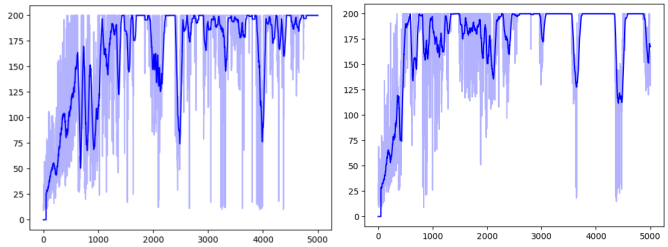

Fig. 14: Results of Actor Critic algorithm - with learning rate of 0.0005

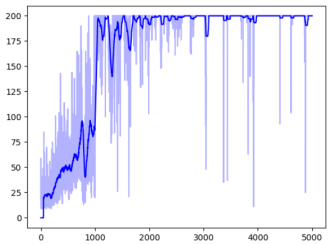

Fig. 15: Result of Actor Critic algorithm - with learning rate of 0.0002

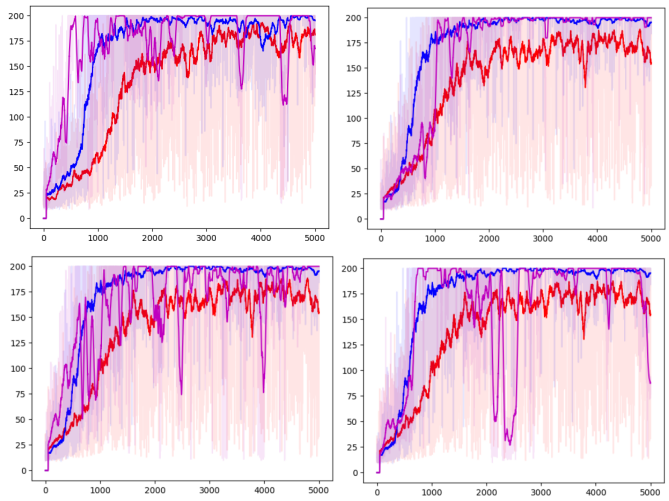

Fig. 16: Results of 3 algorithms: Reinforce, Reinforce with Baseline, and Actor-Critic

From all the runs, Actor-Critic algorithm is the fastest one to reach the optimal value, however, it consists of many fluctuations after that. REINFORCE with Baseline is the most stable one. It reaches the optimal value after 1200 training episodes (slower than Actor-Critic), and the performance is stable with very little variation. REINFORCE algorithm is the slowest one, and takes a lot of training effort to reach not even the optimal value.

III. Continuous Action Policy Optimization

To work in continuous environments, we need to build a different policy. Two function *sample_action* and *get_action_log_prob* are shown in Fig. 17. To prevent the policy network return *nan* for the action mean, we added an if condition. If its value is smaller than *1e-6*, we set the action mean $action\_means$ equal to 0. Similarly to the standard deviation $action\_stddevs$. Then we build a Gaussian distribution from the mean and std. From the distribution, we can sample an action and return it.

```
def sample_action(self, state, epsilon=0.0):
    action_means, action_stddevs = self.forward(state)
    # action_stddevs += 1e-6  # A trick to improve numerical stability
    if torch.abs(action_means) <1e-6:
        action_means = torch.tensor(0, dtype=torch.float)
    if torch.abs(action_stddevs) < 1e-6:
        action_stddevs = torch.tensor(1e-6, dtype=torch.float)
    dist = torch.distributions.Normal(action_means, action_stddevs)
    action=0
    if epsilon > 0.0:
        raise ValueError("No 'epsilon greedy' for this policy.")
    action = dist.sample()
    # You may need to use action.numpy()
    return action.numpy()

def get_action_log_prob(self, state, action):
    action_means, action_stddevs = self(state)
    # action_stddevs += 1e-6  # A trick to improve numerical stability
    if torch.abs(action_means) <1e-6:
        action_means = torch.tensor(0, dtype=torch.float)
    if torch.abs(action_stddevs) < 1e-6:
        action_stddevs = torch.tensor(1e-6, dtype=torch.float)
    dist = torch.distributions.Normal(action_means, action_stddevs)
    return dist.log_prob(torch.tensor(action, dtype=torch.float))
```

Fig. 17: Continuous Policy Code

Similarly, function *get_action_log_prob* is the same, except we need to get the likelihood of that action, calculate the log value then return it.

After the new policy and the new value function networks are reconstructed, we integrate them into three algorithms REINFORCE, REINFORCE with Baseline, and Actor-Critic to run on *Pendulum-v1*. The results are shown in Fig. 19. I have run the training several times (two are shown here), and REINFORCE with Baseline provides the most stable and high values. Actor-Critic usually reaches the optimal value very early, however, there is a lot of fluctuation during the training. Sometimes, it gets a huge drop in the performance. REINFORCE performs worst in all algorithms.

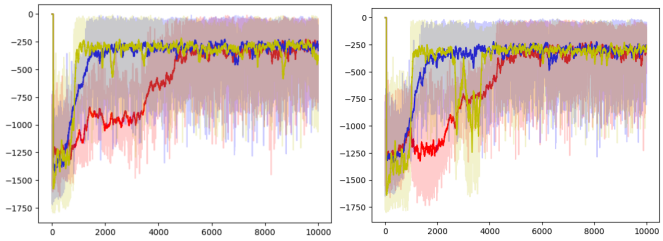

Fig. 18: Results of 3 algorithms: Reinforce, Reinforce with Baseline, and Actor-Critic in Continuous Pendulum-v1 $g = 2.0$

Question How does the performance change when you increase the gravity. Why? What does ideal behavior look like? Why is it so difficult to learn this behavior? Comment on what might be necessary to achieve better performance.
Answer: As increasing the g value to 5, the algorithms struggle to control the pendulum and the results are not good (Fig. 19). The large g is, the more torque/force we need to control the system. As the maximal torque/force (≤ 1 with *softmax*

function) is the same for both g, it is logical to think that the algorithms have to struggle more to handle a larger g value. Furthermore, on the pendulum environment, the boundary for the action value is in $(-2.0, 2.0)$. If we use the *Softmax* as the output of the policy network, the mean value will always be positive, and the output tends to be a positive value. A negative torque for the action only occurs if the *stddevs* has a large value. For those reason, it is so challenging for the above algorithms to control well the continuous pendulum.

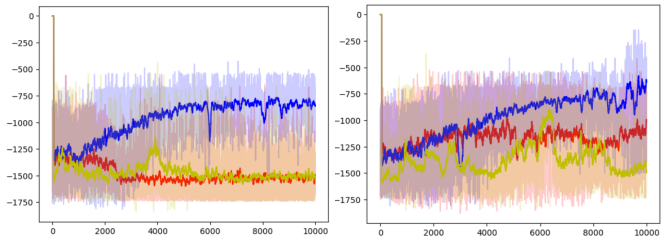

Fig. 19: Results of 3 algorithms: Reinforce, Reinforce with Baseline, and Actor-Critic in Continuous Pendulum-v1 - $g = 5$

To fit the action value into the allowance range, I think we should change the output layer of the policy network from *Softmax* to *Tanh*. When we call functions *sample_action()* and *get_action_log_prob()*, the action mean value should be multiplied by a factor of 2. The new policy should like in Fig. 20. (The modification are in the red boxes).

```
class PolicyApproxGaussian(torch.nn.Module):
    def __init__(self, n_states):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(n_states, 16),
            nn.Tanh(),
            nn.Linear(16, 16),
            nn.Tanh(),
            nn.Linear(16, 2),
            nn.Tanh()
        )
    def sample_action(self, state, epsilon=0.0):
        action_means, action_stddevs = self.forward(state)
        # action_stddevs += 1e-6  # A trick to improve numerical stability
        if torch.abs(action_means) <1e-6:
            action_means = torch.tensor(0, dtype=torch.float)
        if torch.abs(action_stddevs) < 1e-6:
            action_stddevs = torch.tensor(1e-6, dtype=torch.float)
        dist = torch.distributions.Normal(2*action_means, action_stddevs)
        if epsilon > 0.0:
            raise ValueError("No 'epsilon greedy' for this policy.")
        action = dist.sample()
        return action.numpy()

    def get_action_log_prob(self, state, action):
        action_means, action_stddevs = self(state)
        if torch.abs(action_means) <1e-6:
            action_means = torch.tensor(0, dtype=torch.float)
        if torch.abs(action_stddevs) < 1e-6:
            action_stddevs = torch.tensor(1e-6, dtype=torch.float)
        dist = torch.distributions.Normal(2*action_means, action_stddevs)
        return dist.log_prob(torch.tensor(action, dtype=torch.float))
```

Fig. 20: New Continuous Policy

IV. Running PPO

After running PPO several times for environment *Pendulum-v1* with different g values, the results are shown in Fig. 21, 22, and 23.

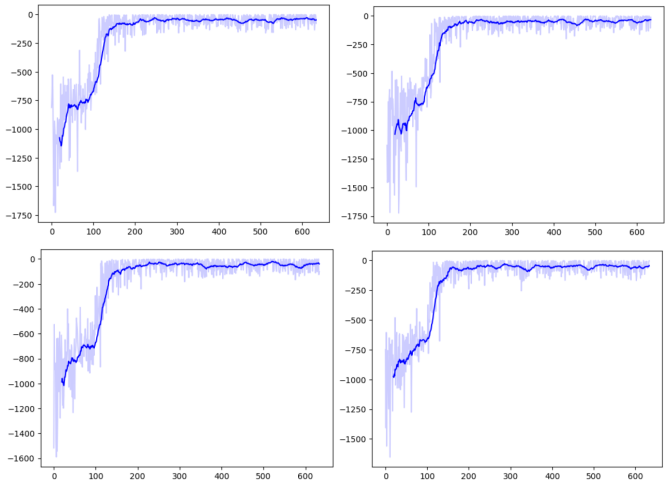

Fig. 21: Several run results of PPO with g=2

As comparing the output figures, we can say that PPO outperforms all the algorithms above. It can reach higher average return value within few hundred of steps with very little variation.

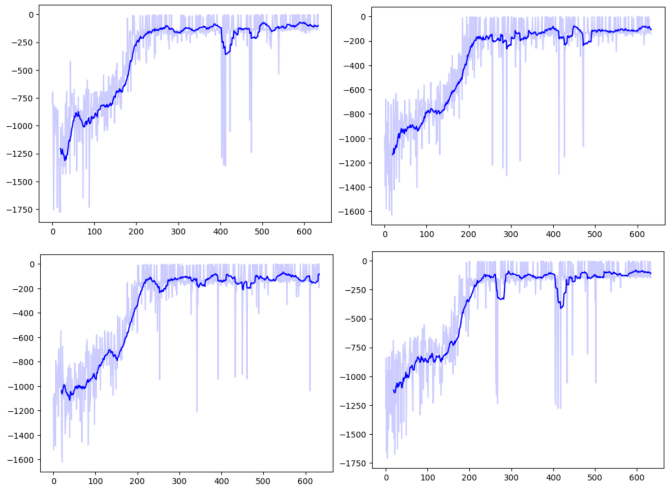

Fig. 22: Several run results of PPO with g=5

As we increase the value of g to 5 and 9.81, there are more challenge for the algorithm to handle. The large g is, the more torque/force we need to control the system. As limiting the maximal torque/force, we can see that PPO performs worse and worse to the increasing value of g. With $g = 2$, PPO needs only around 130 step to reach the optimal value. It needs around 210 steps for $g = 5$, and for $g = 9.81$, more than 300 steps is required for PPO reaching the optimal value. Moreover, the average returns reduces with the increasing of g value. For $g = 2$, the total return is close to -80, and the total returns for $g = 5$ and $g = 9.81$ are -180 and -150, respectively. Those results are consistent with the output in section 5.3.

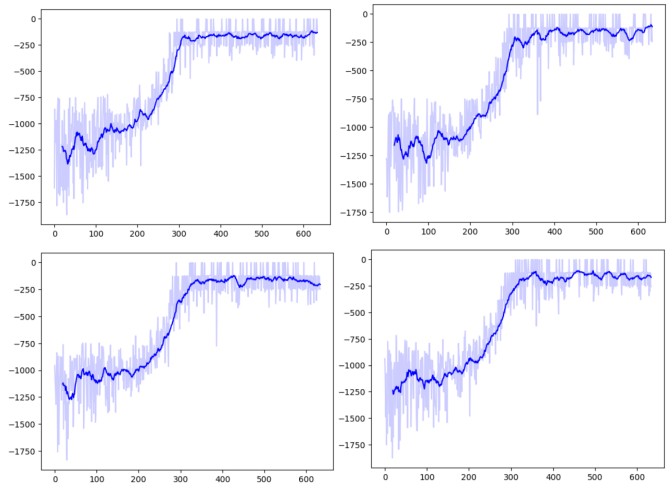

Fig. 23: Several run results of PPO with g=9.81