Deep Learning Homework 7 Report

Cheng-Liang Chi

May 18, 2025

Contents

1 Introduction

This report details the implementation and analysis of two prominent policy-based reinforcement learning methods: Advantage Actor-Critic (A2C) [1] and Proximal Policy Optimization (PPO) [2], specifically employing PPO-Clip [3] with Generalized Advantage Estimation (GAE) [4]. Utilizing PyTorch [5] and OpenAI Gym [6] environments, this work systematically explores and evaluates these methods on the `Pendulum-v1` environment as well as a more challenging MuJoCo [7] locomotion task, `Walker2d-v4`.

The objective of this assignment is to:

Develop a thorough understanding of the key components and algorithms underlying policy-based reinforcement learning approaches.

Gain practical experience by implementing and optimizing A2C [1] and PPO [2] algorithms with GAE [4] in PyTorch [5].

Analyze and compare the performance, training stability, and sample efficiency of A2C versus PPO through empirical experimentation.

The report is organized into several key sections: Section 2 describes the specifics of the implemented algorithms, including methods used to obtain stochastic policy gradients, advantage estimations, and sample collections. Section 3 presents experimental results, with detailed comparisons between A2C and PPO implementations.

2 Implementation Details

2.1 Task1: A2C in `Pendulum-v1`

2.1.1 Stochastic Policy Gradient Calculation

In the `update_model()` method, the policy gradient is computed as:

```python
def update_model(self) -> Tuple[float, float]:
    """Update the model by gradient descent."""
    state, log_prob, next_state, reward, done = self.transition

    next_state = torch.FloatTensor(next_state).to(self.device)
    reward = torch.FloatTensor([reward]).to(self.device)
    done = torch.FloatTensor([done]).to(self.device)

    # Q_t   = r + gamma * V(s_{t+1})  if state != Terminal
    #       = r                       otherwise
    mask = 1 - done

    value_next = self.critic(next_state)
    Q_t = reward + self.gamma * value_next * mask

    value_loss = F.mse_loss(self.critic(state), Q_t.detach())

    # Update value
    self.critic_optimizer.zero_grad()
    value_loss.backward()
    self.critic_optimizer.step()
```

```python

    # Advantage = Q_t - V(s_t)
    advantage = Q_t - self.critic(state)
    policy_loss = (
        -log_prob * advantage.detach()
    ).mean() - self.entropy_weight * log_prob.mean()

    # Update policy
    self.actor_optimizer.zero_grad()
    policy_loss.backward()
    self.actor_optimizer.step()

    return policy_loss.item(), value_loss.item()
```

This represents the advantage-weighted log probability gradient, augmented with an entropy regularization term to encourage exploration.

2.1.2 TD Error Estimation

Temporal Difference (TD) error is computed using the Bellman equation, as shown bellow:

```python
def update_model(self) -> Tuple[float, float]:
    """Update the model by gradient descent."""
    state, log_prob, next_state, reward, done = self.transition

    next_state = torch.FloatTensor(next_state).to(self.device)
    reward = torch.FloatTensor([reward]).to(self.device)
    done = torch.FloatTensor([done]).to(self.device)

    # Q_t   = r + gamma * V(s_{t+1})  if state != Terminal
    #       = r                       otherwise
    mask = 1 - done

    value_next = self.critic(next_state)
    Q_t = reward + self.gamma * value_next * mask

    value_loss = F.mse_loss(self.critic(state), Q_t.detach())

    # Update value
    self.critic_optimizer.zero_grad()
    value_loss.backward()
    self.critic_optimizer.step()

    # Advantage = Q_t - V(s_t)
    advantage = Q_t - self.critic(state)
    policy_loss = (
        -log_prob * advantage.detach()
    ).mean() - self.entropy_weight * log_prob.mean()

    # Update policy
```

```python
    self.actor_optimizer.zero_grad()
    policy_loss.backward()
    self.actor_optimizer.step()

    return policy_loss.item(), value_loss.item()
```

This helps the critic estimate future returns using bootstrapped values.

2.2 Task2: PPO-Clip with GAE on `Pendulum-v1`

2.2.1 Clipped Objective

In the `update_model()` method of `PPOAgent` (in `ppo_pendulum.py`), we compute the clipped objective as:

```python
def update_model(self, next_state: np.ndarray) -> Tuple[float, float]:
    """Update the model by gradient descent."""
    next_state = torch.FloatTensor(next_state).to(self.device)  # type: ignore
    next_value = self.critic(next_state)

    returns = compute_gae(
        next_value,
        self.rewards,
        self.masks,
        self.values,
        self.gamma,
        self.tau,
    )
    states = torch.cat(self.states).view(-1, self.obs_dim)
    actions = torch.cat(self.actions)
    returns = torch.cat(returns).detach()
    values = torch.cat(self.values).detach()
    log_probs = torch.cat(self.log_probs).detach()
    advantages = (returns - values).detach()

    actor_losses, critic_losses = [], []

    for state, action, old_value, old_log_prob, return_, adv in ppo_iter(
        update_epoch=self.update_epoch,
        mini_batch_size=self.batch_size,
        states=states,
        actions=actions,
        values=values,
        log_probs=log_probs,
        returns=returns,
        advantages=advantages,
    ):
        # calculate ratios
        _, dist = self.actor(state)
```

```
            log_prob = dist.log_prob(action)
            ratio = (log_prob - old_log_prob).exp()

            surr1 = ratio * adv
            surr2 = torch.clamp(ratio, 1.0 - self.epsilon, 1.0 + self.epsilon) * adv
            actor_loss = (
                -torch.min(surr1, surr2).mean()
                - self.entropy_weight * dist.entropy().mean()
            )

            critic_loss = F.mse_loss(self.critic(state), return_)

            # train critic
            self.critic_optimizer.zero_grad()
            critic_loss.backward(retain_graph=True)
            self.critic_optimizer.step()

            # train actor
            self.actor_optimizer.zero_grad()
            actor_loss.backward(retain_graph=True)
            self.actor_optimizer.step()

            actor_losses.append(actor_loss.item())
            critic_losses.append(critic_loss.item())

        self.states, self.actions, self.rewards = [], [], []
        self.values, self.masks, self.log_probs = [], [], []

        actor_loss = sum(actor_losses) / len(actor_losses)
        critic_loss = sum(critic_losses) / len(critic_losses)

        return actor_loss, critic_loss
```

This clipping mechanism ensures that the policy update does not move too far from the old policy, improving training stability.

2.2.2 Generalized Advantage Estimator (GAE)

The GAE is implemented in the `compute_gae()` method, which calculates the advantage estimates using a combination of TD errors and bootstrapped values:

```
def compute_gae(next_value, rewards, masks, values, gamma, tau):
    gae = 0
    returns = []
    values = values + [next_value]
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
        gae = delta + gamma * tau * masks[step] * gae
```

```
        returns.insert(0, gae + values[step])
    return returns
```

Here, gamma (γ) and tau (τ) are user-configurable hyperparameters that balance bias and variance in advantage estimation.

2.3 Task3: PPO with GAE on Walker2d-v4

2.3.1 Sample Collection from the Environment

Transitions are collected similarly as in Task 2 (Section 2.2), using the `select_action()` and `step()` methods in `ppo_walker.py`. These transitions are stored in memory and later batched during training with `ppo_iter()`.

2.3.2 Exploration Enforcement

The entropy bonus is included in the policy loss to ensure exploration:

```
surr1 = ratio * adv
surr2 = torch.clamp(ratio, 1.0 - self.epsilon, 1.0 + self.epsilon) * adv
actor_loss = (
    -torch.min(surr1, surr2).mean()
    - self.entropy_weight * dist.entropy().mean()
)
```

he coefficient for this term can be adjusted using the `--entropy-weight` CLI parameter (see `main()` function).

2.4 Model Performance Tracking

Since we are required to complete 3 different tasks in two different environments, I used the Weight & Biases (WandB) library to track the training process and visualize the results.

2.4.1 Categorizing the Results

To categorize the results, I used the `wandb.init` method to create a new run for each task with different Project names. The reson why not using tags to catogorize the results as I did in homework 5 is that I wanted to have a clear separation between the different tasks and environments. This allows me to drag different dashboards avoid confusion and easily compare the results of different runs.

2.4.2 Snapshotting the Code

I also used set argument `save_code` to `True`. This allows me to easily reproduce the results and compare the code used for different runs.

2.4.3 Hyperparameter Tuning

I used the `wandb.config` method to define the hyperparameters for each run. This allows me to easily track and compare the hyperparameters used for different runs in the WandB dashboard.

2.4.4 Result Plot Generation

I used the `wandb.log` method to log the results of each run. This allows me to easily visualize the results in the WandB dashboard and compare the performance of different runs. This also allows me to easily export the plots and use them in the report.

3 Discussion

3.1 Task 1: A2C in `Pendulum-v1`

The A2C algorithm was implemented and tested in the `Pendulum-v1` environment. The training process involved collecting samples, computing the policy gradient, and updating the model parameters. The results indicate that A2C is capable of learning a policy that effectively balances the pendulum, achieving a reward over -150 after training for 1000 episodes.

3.1.1 Traning Command

The training command used for this task was:

```
python a2c_pendulum.py --device cpu --exp report
```

The hyperparameters used for training were:

- `device`: cpu
- `actor-lr`: 3×10^{-4}
- `critic-lr`: 3×10^{-3}
- `discount-factor`: 0.9
- `num-episodes`: 1000
- `eval-episodes`: 5
- `seed`: 42
- `entropy-weight`: 0.01

As the number of episodes are set to 1000, the model will be trained for at most 200k steps. Since it can achieve a reward over -150 after 1000 episodes that match the full grade requirement, I stop the training process at this point. But if you want to train the model for more episodes, you can set the `num-episodes` to a larger number.

3.1.2 Training Curve

The training curve for the A2C algorithm in the `Pendulum-v1` environment is shown in Figure 1. The x-axis represents the number of environment steps taken, while the y-axis shows the average reward obtained at evaluation.

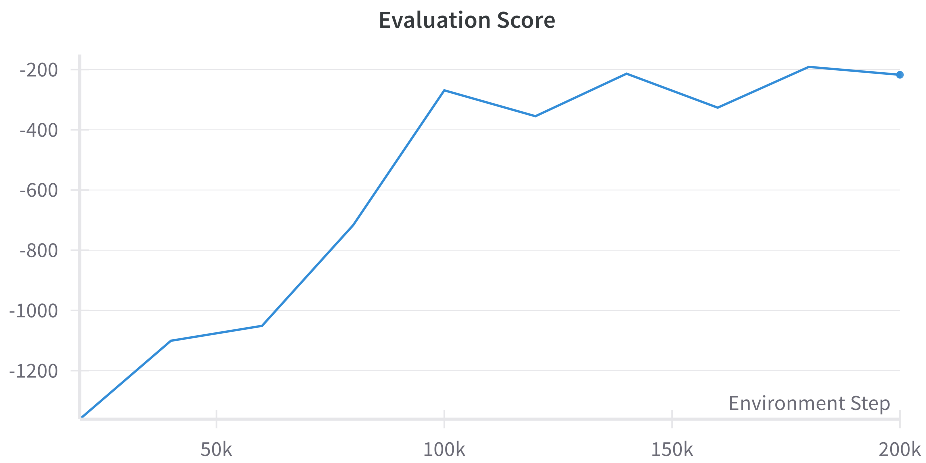

Figure 1: A2C training curve in the `Pendulum-v1` environment.

3.1.3 Testing Command

```
python3 a2c_pendulum.py --device cpu --mode test --ckpt
    results/task1/report/a2c_best.pth --seed 210114487
```

The seed set for testing is 210114487, which is just a random number to ensure the reproducibility of the results. The testing environment's seed are not picked by any specific rule, but just generated by adding from the specified seed. That is, the testing environment's seed is 210114487 + i, where i is the index of the testing environment.

For the first task, I can achieve reward -111.6 over 20 consecutive testing episodes using the best checkpoint trained in 200k environment steps.

3.2 Task 2: PPO-Clip with GAE on `Pendulum-v1`

The PPO-Clip algorithm was implemented and tested in the `Pendulum-v1` environment. The training process involved collecting samples, computing the policy gradient, and updating the model parameters. The results indicate that PPO-Clip is capable of learning a policy that effectively balances the pendulum, achieving a reward over -150 after training for 200k environment steps.

3.2.1 Training Command

The training command used for this task was:

```
python3 ppo_pendulum.py --device cpu --exp report
```

The hyperparameters used for training were:

- `device`: `cpu`
- `actor-lr`: 1×10^{-4}
- `critic-lr`: 3×10^{-4}
- `discount-factor`: 0.9
- `num-episodes`: 100
- `eval-episodes`: 5
- `seed`: 42
- `entropy-weight`: 0.01
- `batch-size`: 128
- `epsilon`: 0.2
- `rollout-len`: 2000

As the number of episodes are set to 100, the model will be trained for at most 200k steps. Since it can achieve a reward over -150 after 1000 episodes that match the full grade requirement, I stop the training process at this point. But if you want to train the model for more episodes, you can set the `num-episodes` to a larger number.

3.2.2 Training Curve

The training curve for the PPO-Clip algorithm in the `Pendulum-v1` environment is shown in Figure 2. The x-axis represents the number of environment steps taken, while the y-axis shows the average reward obtained at evaluation. The evaluation is performed every 20k environment steps, and the average reward is calculated over 5 episodes.

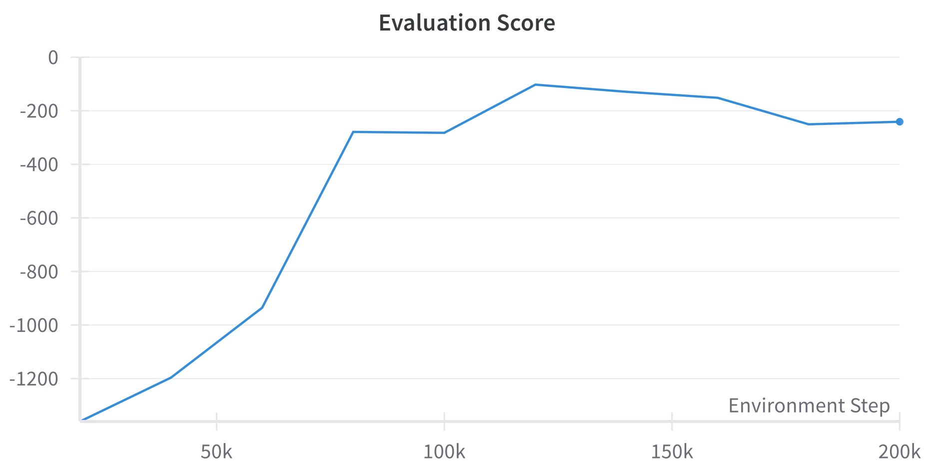

Figure 2: PPO-Clip training curve in the `Pendulum-v1` environment.

The training curve shows that the PPO-Clip algorithm is able to learn a policy that effectively balances the pendulum, achieving a reward over -150 after training for 200k environment steps. The training process was set to 200k environment steps, and the model was trained for at most 200k steps.

3.2.3 Testing Command

```
python3 ppo_pendulum.py --device cpu --mode test --ckpt
↪  results/task2/report/ppo_best.pth --seed 1708180417
```

Same as above, the seed set for testing is 1708180417, which is just a random number to ensure the reproducibility of the results. The testing environment's seed are not picked by any specific rule, but just generated by adding from the specified seed. That is, the testing environment's seed is 1708180417 + i, where i is the index of the testing environment.

For the second task, I can achieve reward -96.90 over 20 consecutive testing episodes using the best checkpoint trained in 200k environment steps.

3.3 Task 3: PPO-Clip with GAE on `Walker2d-v4`

The PPO algorithm was implemented and tested in the `Walker2d-v4` environment. The training process involved collecting samples, computing the policy gradient, and updating the model parameters. The results indicate that PPO is capable of learning a policy that effectively balances the walker, achieving a reward over 2500 after training for 1,000,000 environment steps.

3.3.1 Training Command

The training command used for this task was:

```
python3 ppo_walker.py --device cuda --exp report
```

The hyperparameters used for training were:

- `device`: cpu
- `actor-lr`: 1×10^{-4}
- `critic-lr`: 3×10^{-4}
- `discount-factor`: 0.99
- `num-episodes`: 400
- `eval-episodes`: 5
- `seed`: 42
- `entropy-weight`: 0.01
- `batch-size`: 64

- `epsilon`: 0.2
- `rollout-len`: 2500

As the number of episodes are set to 400, and the rollout length is set to 2500, the model will be trained for at most 1,000,000 steps. Since it can achieve a reward over 2500 after 1,000,000 steps that match the full grade requirement, I stop the training process at this point. But if you want to train the model for more episodes, you can set the `num-episodes` or `rollout-len` to a larger number.

3.3.2 Training Curve

The training curve for the PPO-Clip algorithm in the `Walker2d-v4` environment is shown in Figure 3. The x-axis represents the number of environment steps taken, while the y-axis shows the average reward obtained at evaluation. The evaluation is performed every 25 episodes, and the average reward is calculated over 5 episodes.

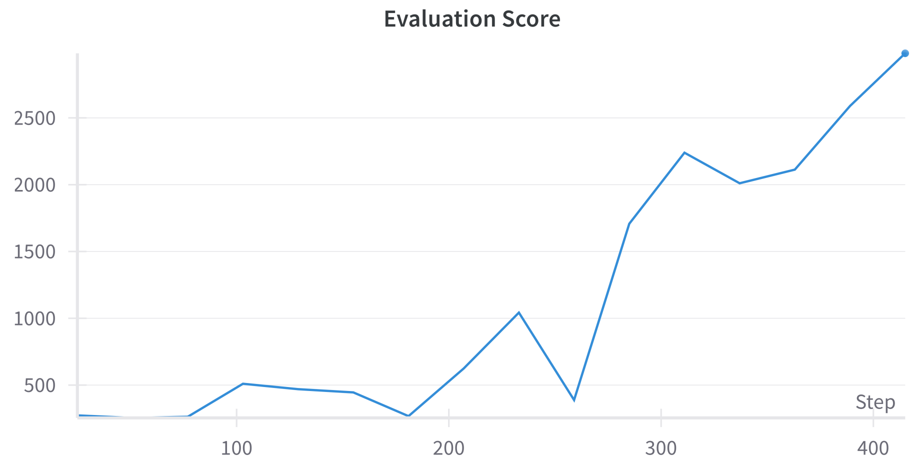

Figure 3: PPO-Clip training curve in the `Walker2d-v4` environment.

The training curve shows that the PPO-Clip algorithm is able to learn a policy that effectively balances the walker, achieving a reward over 2500 after training for 1,000,000 environment steps. The training process was set to 400 episodes, and the model was trained for at most 1,000,000 steps.

3.3.3 Testing Command

```
python3 ppo_walker.py --device cpu --mode test --ckpt
↪  results/task3/report/ppo_best.pth --seed 2139949224
```

Same as above, the seed set for testing is 2139949224, which is just a random number to ensure the reproducibility of the results. If you want to test the model in a different environment, you can set the `seed` to a different number. It is possible to lead to different results, but the model should still be able to achieve a reward over 3,500 for a very high probability. The testing environment's seed are not picked by any specific rule, but just generated by adding from the specified seed. That is, the testing environment's seed is 2139949224 + i, where i is the index of the testing environment.

For the third task, I can achieve reward 3655.76 over 20 consecutive testing episodes using the best checkpoint trained in 1,000,000 environment steps.

References

[1] M. Han, L. Zhang, J. Wang, and W. Pan, *Actor-critic reinforcement learning for control with stability guarantee*, 2020. arXiv: 2004.14288 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2004.14288.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[3] N.-C. Huang, P.-C. Hsieh, K.-H. Ho, and I.-C. Wu, *Ppo-clip attains global optimality: Towards deeper understandings of clipping*, 2024. arXiv: 2312.12065 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2312.12065.

[4] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, 2018. arXiv: 1506.02438 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1506.02438.

[5] J. Ansel, E. Yang, H. He, *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf.

[6] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[7] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[8] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016, pp. 2094–2100. [Online]. Available: https://arxiv.org/abs/1509.06461.

[9] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *4th International Conference on Learning Representations (ICLR)*, arXiv:1511.05952, 2016. [Online]. Available: https://arxiv.org/abs/1511.05952.

[10] B. Daley, M. White, and M. C. Machado, "Averaging *n*-step returns reduces variance in reinforcement learning," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, arXiv:2402.03903, 2024. [Online]. Available: https://arxiv.org/abs/2402.03903.

[11] V. Mnih, A. P. Badia, M. Mirza, *et al.*, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.