CSE 584, Machine Learning
The Pennsylvania State University
Fall 2024

Homework #2

Jeongwon Bae (945397461)

Abstract

This project implements the Asynchronous Advantage Actor-Critic (A3C) algorithm using PyTorch to solve the CartPole-v1 environment from OpenAI Gym. The Actor-Critic framework is a fundamental approach in reinforcement learning that combines both policy-based and value-based methods to achieve efficient and stable learning [1]. In this framework, the actor represents the policy $\pi(a_t|s_t;\theta)$, which selects actions a_t based on the current state s_t and policy parameters θ. The critic estimates the value function $V(s_t;\theta_v)$, evaluating the expected return from state s_t using value function parameters θ_v. The actor updates its policy in the direction suggested by the critic's feedback, enabling the agent to learn optimal behaviors.

Building upon this foundation, the Advantage Actor-Critic (A2C) algorithm introduces a synchronous approach where multiple agents interact with their own copies of the environment and update a shared global model in a coordinated manner [2]. In A2C, agents ollect experiences and wait for each other to complete their interactions before synchronously updating the global parameters. This method reduces variance in policy gradient estimates by utilizing the advantage function $A(s_t, a_t)$, defined as:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t),$$

where $Q(s_t, a_t)$ is the action-value function, and $V(s_t)$ is the state-value function. The advantage function measures how much better an action a_t is compared to the average expected return from state s_t.

The Asynchronous Advantage Actor-Critic (A3C) algorithm extends this approach by allowing agents to operate asynchronously [2]. In A3C, multiple worker processes interact with their own instances of the environment and update the global model independently without waiting for synchronization. This asynchrony offers several benefits: it decorrelates experiences between agents, reducing variance in updates and improving learning stability, and it maximizes computational resource utilization by eliminating idle times due to synchronization.

In the A3C algorithm, each worker maintains a local copy of the global model's parameters. Workers interact with their environments and periodically update the global model based on their own experiences. Each worker collects experience by interacting with its environment over a sequence of time steps. At each time step t, the worker observes the current state s_t and selects an action a_t according to its policy $\pi(a_t|s_t;\theta)$. After executing the action, the worker receives a reward r_t and transitions to the next state s_{t+1}. This process continues for a fixed number of steps or until a terminal state is reached.

After collecting sufficient experience, the worker computes the n-step return R_t for each time step, which is used to estimate the advantage function $A(s_t, a_t)$. The n-step return is calculated using the rewards and the value estimates from the critic:

$$R_t = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}; \theta_v),$$

where γ is the discount factor (typically $0 < \gamma < 1$), r_{t+k} is the reward received at time $t + k$, and $V(s_{t+n}; \theta_v)$ is the estimated value of the state s_{t+n}. If the episode terminates before n steps, the value function term is omitted, and R_t is simply the sum of discounted rewards.

The advantage for each time step is then computed as:

$$A(s_t, a_t) = R_t - V(s_t; \theta_v),$$

where $V(s_t; \theta_v)$ is the critic's estimate of the value of state s_t. The advantage function represents how much better or worse the action a_t performed compared to the critic's expected value of the state.

The worker then computes the gradients of the policy and value function losses. The policy loss at each time step is given by:

$$L_{\text{policy}} = -\log \pi(a_t | s_t; \theta) A(s_t, a_t),$$

which encourages the policy to assign higher probabilities to actions that yield positive advantages and lower probabilities to those with negative advantages. This effectively reinforces actions that lead to better-than-expected outcomes.

The value loss is defined as:

$$L_v = \left(R_t - V(s_t; \theta_v)\right)^2,$$

which trains the critic to minimize the difference between the estimated value and the actual return observed. An entropy regularization term is often included to encourage exploration by preventing the policy from becoming too deterministic too quickly:

$$L_{\text{entropy}} = -\beta \sum_a \pi(a|s_t; \theta) \log \pi(a|s_t; \theta),$$

where β is a hyperparameter controlling the strength of the entropy regularization.

The total loss for the worker combines these components:

$$L = L_{\text{policy}} + c_v L_v + L_{\text{entropy}},$$

where c_v is a hyperparameter weighting the value loss relative to the policy loss.

The worker computes gradients of the total loss with respect to its local parameters θ and θ_v. These gradients are then applied to update the global parameters using an optimizer such as RMSProp or Adam. The updates are performed asynchronously, meaning that each worker updates the global model independently without waiting for other workers. After updating the global model, the worker synchronizes its local model parameters with the updated global parameters.

The CartPole-v1 environment is a classic control problem where the objective is to prevent a pole, attached by an unactuated joint to a cart, from falling over by applying forces to move the cart left or right [3]. The environment provides a continuous four-dimensional state space representing the cart position (x), cart velocity ($\dot{x}$), pole angle (θ), and pole angular velocity ($\dot{\theta}$). The action space is discrete, consisting of two actions: pushing the cart to the left (0) or to the right (1). The agent receives a reward of +1 for every time step the pole remains upright. The episode terminates if the pole deviates more than 12° from vertical, the cart moves beyond certain boundaries, or the episode length reaches 500 time steps.

By leveraging the A3C algorithm to train an agent on the CartPole-v1 task, efficient exploration and faster convergence can be achieved compared to synchronous methods. Utilizing multiple asynchronous worker processes, each interacting with its own environment instance, allows for continuous updating of the shared global model based on the experiences of all workers. This approach enables the agent to quickly learn a robust policy that balances the pole for extended periods.

This implementation demonstrates the practical application of the A3C algorithm with clear formulations and parameter updates. It showcases the benefits of asynchronous updates in reducing training time and improving performance. Applying the A3C algorithm to the CartPole-v1 environment validates its effectiveness on a well-known control problem and highlights the advantages of using asynchronous, parallelized training methods in reinforcement learning.

Step-by-Step Code Explanation

We will now examine the core components of the reinforcement learning implementation, focusing specifically on the A3C algorithm applied to the CartPole-v1 environment [4]. The code is presented in logical blocks, with detailed explanations and line-by-line comments to demonstrate step-by-step understanding.

Firstly, the necessary libraries are imported, and the hyperparameters are set.

```
import gym                                      # OpenAI Gym for the environment
import torch                                    # PyTorch for neural networks
import torch.nn as nn                           # neural network modules
import torch.nn.functional as F                 # functional modules for activation functions
import torch.optim as optim                     # optimization algorithms
from torch.distributions import Categorical # Categorical distribution for action selection
import torch.multiprocessing as mp              # multiprocessing for parallel processing
import time                                     # time module for delays during testing

# Hyperparameters
n_train_processes = 3     # Number of parallel training processes
learning_rate = 0.0002    # Learning rate for the optimizer
update_interval = 5       # Interval for updating the global model
gamma = 0.98              # Discount factor for future rewards
max_train_ep = 300        # Maximum number of training episodes per worker
max_test_ep = 400         # Maximum number of testing episodes
```

In this block, we import essential modules such as `gym` for the environment, `torch` and its submodules for neural network implementation, and `torch.multiprocessing` for parallel processing. The hyperparameters are defined to control various aspects of the training process, such as the learning rate, the number of episodes, and the discount factor.

Next, the Actor-Critic neural network model is defined.

```
class ActorCritic(nn.Module):
  def __init__(self):
    super(ActorCritic, self).__init__()
    self.fc1 = nn.Linear(4, 256)            # Input layer with 4 inputs (state dimensions) and
        256 outputs
    self.fc_pi = nn.Linear(256, 2)          # Policy output layer with 2 actions (left or right)
    self.fc_v = nn.Linear(256, 1)           # Value output layer producing a single value
        estimate

  def pi(self, x, softmax_dim=0):
    x = F.relu(self.fc1(x))                 # Apply ReLU activation after first layer
```

```python
        x = self.fc_pi(x)                     # Compute action logits
        prob = F.softmax(x, dim=softmax_dim)   # Convert logits to probabilities using softmax
        return prob

    def v(self, x):
        x = F.relu(self.fc1(x))    # Apply ReLU activation after first layer
        v = self.fc_v(x)           # Compute state value estimate
        return v
```

Here, the `ActorCritic` class inherits from `nn.Module` and defines the neural network architecture. The network consists of a shared fully connected layer (`fc1`) followed by separate layers for the policy (`fc_pi`) and value (`fc_v`) outputs. The `pi` method computes the action probabilities using the softmax function, while the `v` method computes the state value estimate.

The `train` function is central to the training process of each worker in the A3C algorithm. Each worker executes this function independently, interacting with its own environment instance and contributing to the asynchronous updates of the global model. It begins by creating a local instance of the `ActorCritic` model, which is synchronized with the global model's parameters. This synchronization ensures that all workers start with the same initial parameters. Additionally, an optimizer is defined to update the global model's parameters, and each worker creates its own instance of the CartPole-v1 environment, allowing them to interact independently.

```python
  def train(global_model, rank):
    local_model = ActorCritic()        # Create a local model for the worker
    local_model.load_state_dict(global_model.state_dict()) # Synchronize local model with global
        model

    optimizer = optim.Adam(global_model.parameters(),
                lr=learning_rate)  # Define optimizer for global model

    env = gym.make('CartPole-v1')      # Create an instance of the environment
```

Once the environment is set up, the worker enters a loop where it will run for a specified number of episodes. At the start of each episode, the environment is reset to its initial state. The `done` flag is initialized to `False`, indicating that the episode has not yet terminated.

```python
    for n_epi in range(max_train_ep):    # Loop over episodes
      done = False
      s = env.reset()                  # Reset the environment to the initial state
```

As the episode progresses, the worker interacts with the environment by collecting experiences. For each time step, the current state is passed through the local policy network to obtain action probabilities. A categorical distribution is created from these probabilities, and an action is sampled. The worker then executes the action in the environment, receiving the next state, reward, and a `done` flag that indicates whether the episode has ended. The current state, action, and scaled reward are stored in lists for later use. This continues until the worker reaches the specified `update_interval` or the episode ends.

```python
    while not done:
      s_lst, a_lst, r_lst = [], [], []                 # Lists to store states, actions, and
          rewards
      for t in range(update_interval):
        prob = local_model.pi(torch.from_numpy(s).float()) # Get action probabilities from policy
            network
        m = Categorical(prob)     # Create a categorical distribution based on probabilities
        a = m.sample().item()     # Sample an action from the distribution
        s_prime, r, done, info = env.step(a)         # Execute the action in the environment

        s_lst.append(s)           # Store the current state
        a_lst.append([a])         # Store the action taken
        r_lst.append(r / 100.0)   # Store the scaled reward (divided by 100)

        s = s_prime                 # Update the current state
        if done:
          break                   # Exit the loop if the episode is done
```

Once the worker has collected a sequence of experiences, it proceeds to compute the necessary values for updating the global model. If the episode has ended, the bootstrap value `R` is set to `0.0`, since there are no future rewards. Otherwise, it is set to the estimated value of the final state, computed by the local critic network. The worker then computes the discounted returns in reverse order from the collected rewards and stores them in a list. This list is reversed to align the returns with the chronological order of the experiences.

```
      s_final = torch.tensor(s_prime, dtype=torch.float)
      R = 0.0 if done else local_model.v(s_final).item()   # Compute the bootstrap value
      td_target_lst = []
      for reward in r_lst[::-1]:
        R = reward + gamma * R     # Compute the discounted return
        td_target_lst.append([R])  # Store the return
      td_target_lst.reverse()        # Reverse to match the order of experiences
```

Next, the worker prepares the data for updating the model by converting the lists of states, actions, and target returns into PyTorch tensors. The advantage estimates are computed by subtracting the value estimates from the target returns. These advantages represent how much better or worse an action performed compared to the expected outcome.

```
      # Convert lists to tensors
      s_batch = torch.tensor(s_lst, dtype=torch.float)
      a_batch = torch.tensor(a_lst)
      td_target = torch.tensor(td_target_lst)

      advantage = td_target - local_model.v(s_batch)   # Compute the advantage estimates
```

The loss is then computed, which consists of both a policy loss and a value loss. The policy loss encourages the model to take actions that have higher advantages by minimizing the negative log probabilities of actions weighted by their advantages. The value loss minimizes the difference between the predicted state values and the computed returns using the smooth L1 loss. (We note that the entropy term is missing here, and to fully implement the algorithm as described in the abstract, it should be added.) Gradients are cleared, and the loss is backpropagated through the network to compute the gradients with respect to the model's parameters.

```
      pi = local_model.pi(s_batch, softmax_dim=1)      # Get action probabilities for the batch
          of states
      pi_a = pi.gather(1, a_batch)                     # Gather probabilities of the actions taken
      loss = -torch.log(pi_a) * advantage.detach() + \
        F.smooth_l1_loss(local_model.v(s_batch), td_target.detach())  # Compute the policy and
            value losses
      # entropy = - (pi * torch.log(pi + 1e-20)).sum(dim=1)  # Compute entropy (to be added)
      # loss -= beta * entropy                                 # Subtract entropy term from loss

      optimizer.zero_grad()                            # Clear gradients
      loss.mean().backward()                           # Backpropagate the mean loss
```

Once the gradients are computed, they are manually copied from the local model to the global model's parameters. This step is crucial for ensuring that the global model is updated with the gradients calculated from the local experiences. After copying the gradients, the optimizer is used to update the global model's parameters. Finally, the local model is synchronized again with the updated global model to ensure it has the latest parameters. At the end of the training loop, the environment is closed, and a message is printed to indicate that the worker has completed its training episodes.

```
      # Copy gradients from local model to global model
```

```
    for global_param, local_param in zip(global_model.parameters(), local_model.parameters()):
      global_param._grad = local_param.grad
    optimizer.step()                                  # Update global model parameters
    local_model.load_state_dict(global_model.state_dict())  # Synchronize local model with
        global model
  env.close()
  print(f"Training process {rank} reached maximum episode.")
```

This explanation outlines how the worker collects experiences from the environment, computes the necessary values for updating the model, and asynchronously updates the shared global model. Through this process, the worker contributes to the overall learning of the A3C algorithm, all while interacting with its own instance of the environment independently from other workers.

The `test` function is designed to evaluate the performance of the trained global model. It runs in a separate process and interacts with the environment without updating the model's parameters. This allows us to monitor how well the agent is performing over time.

```
def test(global_model):
  env = gym.make('CartPole-v1')
  score = 0.0
  print_interval = 20

  for n_epi in range(max_test_ep):  # Loop over episodes
    done = False
    s = env.reset()   # Reset the environment
    while not done:
      prob = global_model.pi(torch.from_numpy(s).float())   # Get action probabilities from
          global model
      a = Categorical(prob).sample().item() # Sample an action
      s_prime, r, done, info = env.step(a)  # Execute the action
      s = s_prime    # Update the current state
      score += r     # Accumulate the reward

    if n_epi % print_interval == 0 and n_epi != 0:
      print(f"# of episode :{n_epi}, avg score : {score / print_interval:.1f}")
      score = 0.0
      time.sleep(1) # Pause for readability
    env.close()
```

In this function, an instance of the CartPole-v1 environment is created using `gym.make('CartPole-v1')`. The variable `score` is initialized to accumulate rewards, and `print_interval` is set to determine how often the average score is displayed. The function runs a loop over a specified number of episodes (`max_test_ep`). At the start of each episode, the environment is reset using `env.reset()`, and the `done` flag is set to `False`.

Within each episode, the agent interacts with the environment until the episode ends. The current state `s` is converted to a PyTorch tensor and passed through the global policy network `global_model.pi` to obtain action probabilities. A categorical distribution is created from these probabilities, and an action `a` is sampled using `Categorical(prob).sample().item()`. The action is executed in the environment with `env.step(a)`, returning the next state `s_prime`, the reward `r`, and the `done` flag. The current state `s` is updated to `s_prime`, and the reward `r` is added to the cumulative `score`.

After each episode, the function checks if the episode number is a multiple of `print_interval` and not zero. If so, it prints the average score over the last `print_interval` episodes using:

```
print(f"# of episode :{n_epi}, avg score : {score / print_interval:.1f}")
```

The cumulative `score` is then reset to `0.0`, and a short pause is added with `time.sleep(1)` to improve readability of the output. Finally, the environment is closed with `env.close()` after all episodes are completed.

Moving on to the main execution block, this is where the global model is initialized, and the training and testing processes are started using multiprocessing.

```python
if __name__ == '__main__':
  global_model = ActorCritic()   # Initialize the global model
  global_model.share_memory()    # Share model parameters for multiprocessing

  processes = []
  for rank in range(n_train_processes + 1): # Create processes for training and testing
    if rank == 0:
      p = mp.Process(target=test, args=(global_model,))    # Create the testing process
    else:
      p = mp.Process(target=train, args=(global_model, rank,))  # Create training processes
    p.start()                  # Start the process
    processes.append(p)
  for p in processes:
    p.join()                   # Wait for all processes to finish
```

In this block, the global model is instantiated by creating an instance of the `ActorCritic` class. The `share_memory()` method is called to allow the model's parameters to be shared across multiple processes, which is essential for the multiprocessing setup. An empty list `processes` is initialized to keep track of all the processes.

A loop is used to create both training and testing processes. If the `rank` is zero, a testing process is created by targeting the `test` function and passing the `global_model` as an argument. For other ranks, training processes are created by targeting the `train` function and passing the `global_model` and the worker's `rank`. Each process is started with `p.start()`, and the process object is appended to the `processes` list.

After starting all processes, another loop is used to call `p.join()` on each process, causing the main program to wait for all child processes to complete before exiting. This ensures that all training and testing operations are finished properly.

By organizing the code in this way, the program leverages multiprocessing to train multiple agents in parallel while also evaluating the agent's performance simultaneously. This setup is fundamental to the A3C algorithm, where asynchronous updates from multiple workers contribute to learning a shared policy and value function.

Overall, the code implements an A3C agent that learns to solve the CartPole-v1 environment. The `ActorCritic` model defines the neural network architecture, combining both policy and value outputs. The `train` function handles experience collection, computation of advantages, and asynchronous updates to the global model. The `test` function evaluates the agent's performance by running it in the environment without learning. The main execution block sets up the multiprocessing environment, initializes the global model, and starts the training and testing processes.

This implementation demonstrates how asynchronous methods can effectively utilize computational resources and lead to efficient learning in reinforcement learning tasks. By having multiple worker processes interact with their own environments and update a shared model, the agent can learn robust policies that generalize well across different states and scenarios.

Complete Code Script

```
import gym                                  # Import OpenAI Gym for the environment
import torch                                # Import PyTorch library
import torch.nn as nn                       # Import neural network modules
import torch.nn.functional as F             # Import functional modules for activation functions
import torch.optim as optim                 # Import optimization algorithms
from torch.distributions import Categorical # Import Categorical distribution for probabilistic action selection
import torch.multiprocessing as mp          # Import multiprocessing for parallel execution
import time                                 # Import time module for delays during testing

# Hyperparameters
n_train_processes = 3    # Number of parallel training processes
learning_rate = 0.0002   # Learning rate for optimizer
update_interval = 5      # Interval for updating global model
gamma = 0.98             # Discount factor for future rewards
max_train_ep = 300       # Maximum number of training episodes per worker
max_test_ep = 400        # Maximum number of testing episodes

class ActorCritic(nn.Module):
  def __init__(self):
    super(ActorCritic, self).__init__()
    self.fc1 = nn.Linear(4, 256)           # Input layer to hidden layer with 256 units
    self.fc_pi = nn.Linear(256, 2)         # Output layer for policy (2 possible actions)
    self.fc_v = nn.Linear(256, 1)          # Output layer for value function

  def pi(self, x, softmax_dim=0):
    x = F.relu(self.fc1(x))                # Apply ReLU activation function
    x = self.fc_pi(x)                      # Pass through policy output layer
    prob = F.softmax(x, dim=softmax_dim)   # Convert logits to probabilities with softmax
    return prob

  def v(self, x):
    x = F.relu(self.fc1(x))                # Apply ReLU activation function
    v = self.fc_v(x)                       # Pass through value output layer
    return v

def train(global_model, rank):
  local_model = ActorCritic()                                # Create a local model
  local_model.load_state_dict(global_model.state_dict())    # Synchronize with global model

  optimizer = optim.Adam(global_model.parameters(), lr=learning_rate)  # Optimizer for global model

  env = gym.make('CartPole-v1')                              # Initialize environment

  for n_epi in range(max_train_ep):                          # Training loop over episodes
    done = False
    s = env.reset()                                        # Reset environment
    while not done:
      s_lst, a_lst, r_lst = [], [], []                   # Lists to store states, actions, rewards
      for t in range(update_interval):
        prob = local_model.pi(torch.from_numpy(s).float())  # Get action probabilities
        m = Categorical(prob)                            # Create categorical distribution
        a = m.sample().item()                            # Sample action
        s_prime, r, done, info = env.step(a)             # Perform action in environment

        s_lst.append(s)                                  # Append state to list
        a_lst.append([a])                                # Append action to list
        r_lst.append(r / 100.0)                          # Append scaled reward to list

        s = s_prime                                      # Update state
        if done:
          break                                        # Break if episode is done

      s_final = torch.tensor(s_prime, dtype=torch.float)
      R = 0.0 if done else local_model.v(s_final).item()   # Bootstrap value for last state
      td_target_lst = []
      for reward in r_lst[::-1]:
        R = reward + gamma * R                           # Calculate discounted return
        td_target_lst.append([R])                        # Append to target list
        td_target_lst.reverse()                          # Reverse to match order

        s_batch = torch.tensor(s_lst, dtype=torch.float) # Convert states to tensor
        a_batch = torch.tensor(a_lst)                    # Convert actions to tensor
        td_target = torch.tensor(td_target_lst)          # Convert targets to tensor

        advantage = td_target - local_model.v(s_batch)   # Compute advantage

        pi = local_model.pi(s_batch, softmax_dim=1)      # Get action probabilities
        pi_a = pi.gather(1, a_batch)                     # Get probabilities of taken actions
        loss = -torch.log(pi_a) * advantage.detach() + \
          F.smooth_l1_loss(local_model.v(s_batch), td_target.detach())  # Compute loss
        # entropy = - (pi * torch.log(pi + 1e-20)).sum(dim=1)  # Compute entropy (to be added)
        # loss -= beta * entropy                                 # Subtract entropy term from loss

        optimizer.zero_grad()                            # Clear gradients
        loss.mean().backward()                           # Backpropagate loss
        for global_param, local_param in zip(global_model.parameters(), local_model.parameters()):
          global_param._grad = local_param.grad        # Copy gradients to global model
        optimizer.step()                                 # Update global model
        local_model.load_state_dict(global_model.state_dict())  # Synchronize local model

    env.close()
    print(f"Training process {rank} reached maximum episode.")

  def test(global_model):
    env = gym.make('CartPole-v1')                              # Initialize environment
    score = 0.0
    print_interval = 20

    for n_epi in range(max_test_ep):                           # Testing loop over episodes
      done = False
      s = env.reset()                                        # Reset environment
      while not done:
        prob = global_model.pi(torch.from_numpy(s).float())    # Get action probabilities
        a = Categorical(prob).sample().item()              # Sample action
        s_prime, r, done, info = env.step(a)               # Perform action
        s = s_prime                                        # Update state
        score += r                                         # Accumulate reward

      if n_epi % print_interval == 0 and n_epi != 0:
        print(f"# of episode :{n_epi}, avg score : {score / print_interval:.1f}")
        score = 0.0
        time.sleep(1)                                      # Pause for readability
    env.close()

  if __name__ == '__main__':
    global_model = ActorCritic()                       # Initialize global model
    global_model.share_memory()                        # Share model parameters for multiprocessing

    processes = []
    for rank in range(n_train_processes + 1):          # Create processes
      if rank == 0:
        p = mp.Process(target=test, args=(global_model,))     # Testing process
      else:
        p = mp.Process(target=train, args=(global_model, rank,))  # Training process
      p.start()                                      # Start process
      processes.append(p)
    for p in processes:
      p.join()                                       # Wait for all processes to finish
```

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. The MIT Press, 2018.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937.

[3] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," in *Artificial Neural Networks: Concept Learning.* IEEE Press, 1990, pp. 81–93, ISBN: 0818620153.

[4] S. Rho. "Minimalrl-pytorch. "[Online]. Available: `https://github.com/seungeunrho/minimalRL`.