Deep Reinforcement Learning: REINFORCE vs A2C

Bryan Van Draanen
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle, WA 98195
bryanvd@cs.washington.edu
github.com/bryanvandraanen/deep-reinforcement-learning

Abstract

I present an implementation, investigation, and comparison into two policy-based reinforcement learning methods: REINFORCE and Advantage Actor Critic (A2C). The algorithms are evaluated using the OpenAI Gym CartPole-v1 environment and find both agents proficient at the task - capable of flawless performance in the classic control problem when acting on their respective optimal policies after training. I show that REINFORCE experiences more efficient policy learning and offer the explanation that the complexity of training an additional network in A2C for the relatively simple and discrete CartPole task necessitates additional training.

1 Motivation

This project presents an exploration and in-depth exercise into the understanding of two policy-based reinforcement learning methods: REINFORCE and Advantage Actor Critic (A2C). Note that A2C is the synchronous variant of the algorithm as opposed to Asynchronous Advantage Actor Critic (A3C) and was ultimately selected to minimize the noise introduced by asynchrony [2,6].

1.1 Policy Based Reinforcement Learning Methods

Policy-based methods present a different approach to classic reinforcement learning tasks. Rather than trying to learn a value function - one that maps an environment state-action pair to a value (i.e. Q-value) - policy-based methods attempt to directly optimize the policy via policy gradients [4]. Since these methods rely on calculating gradients from a trajectory (a sequence of actions taken in an episode), they are natural candidates for backpropagation and an underlying neural network.

REINFORCE represents the Monte-Carlo variant of policy-based reinforcement learning which relies on single "actor" which performs gradient *ascent* on the sum of rewards attained from a given episode [5]. On the other hand, A2C takes the same actor and incorporates the concept of a "critic" network which judges whether each individual action in an episode is effective resulting in a more nuanced evaluation of the trajectory [2]. The primary motivation for comparing these methods is to better understand how incorporating the aspect of a critic may offer superior training efficiency at the cost of potentially increased complexity learning weights for a network approximating a value function.

1.2 Personal

Deep reinforcement learning is a hot and fascinating topic in computer science and artificial intelligence. Current deep reinforcement learning initiatives by companies such as Google DeepMind pit agents against retro video games, classic board games, and other tasks originally designed to

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Table 1: CartPole-v1 observation space (input data)

Observation	Input Range
Cart Position	(-4.8, 4.8)
Cart Velocity	$(-\infty, \infty)$
Pole Angle	(-24°, 24°)
Pole Velocity at Tip	$(-\infty, \infty)$

challenge humans [1]. Having yet to receive a formal opportunity to create a reinforcement learning agent, the allure of implementing REINFORCE and A2C for classic control theory problems presents an excellent foray into the field.

2 Definition

This project focuses on the implementation correctness, generalization, and comparison of the REINFORCE and A2C algorithms. To accomplish this task, each reinforcement learning agent must conform to the OpenAI Gym environment abstraction and are evaluated using the classic "CartPole" control problem - an environment where the agent learns to balance a long pole attached to a horizontally-movable cart. Both the REINFORCE and A2C algorithms will be used to train an agent under the OpenAI Gym CartPole environment until proficient at the task.

2.1 OpenAI Gym: CartPole-v1

OpenAI Gym is a Python library that provides an abstraction applicable to nearly any reinforcement learning environment. Explicitly, the specification provides an input (observation) space of the environment, and takes an output (action) space which represents the agent's decision.

The "CartPole-v1" environment in OpenAI Gym provides an observation and action space analogous to the classic control cart-and-pole problem described previously [3]. The input data provided by the CartPole-v1 environment shown in Table 1 functions as the explored dataset for the REINFORCE and A2C agents.

Furthermore, the CartPole-v1 environment also provides a discrete output space which decides which direction to move the cart (in an attempt to balance the attached pole). This action space consists of two unique actions: push the cart to the left; push the cart to the right.

For every transition taken in this setting while the pole is still successfully balancing on the cart, the agent receives a reward of 1. Episodes terminate if 500 transitions have been performed, the pole angle exceeds 12°in either direction, or the center of the cart travels out-of-bounds (i.e. exceeds the cart position range of the environment).

2.2 Algorithm Generalization

OpenAI Gym provides the general state and action abstraction described previously. In addition to adhering to the OpenAI Gym environment abstraction, the REINFORCE and A2C algorithm implementations should generalize to any generic game-playing or reinforcement learning task. Specifically, the design of each agent supports a generic "get_action" specification with a "greedy" modifier which indicates whether to act on the optimal policy if true, and otherwise make probabilistic decisions based on the weight of each action.

3 Methodology

In order to compare the implementations of the REINFORCE and A2C algorithms, they must be implemented and evaluated under the same reinforcement learning environment after training and reaching proficiency. In this instance, both REINFORCE and A2C were implemented and trained in the CartPole environment until they reached proficiency.

3.1 Implementation

The implementation of the REINFORCE algorithm consists of an input layer, one hidden fully-connected layer, and one output layer corresponding to the action size. The actor completes an episode making exploratory actions weighted by the confidence of each action in the current policy and performs gradient ascent on the sum of total rewards attained from the episode.

The A2C implementation leverages the exact same actor as REINFORCE and incorporates an additional critic network. The critic network has a similar structure with one input layer, one hidden fully-connected layer, and an output layer converging on a single value approximating the value function. A2C follows a similar procedure as REINFORCE as it completes an entire episode then uses the mean-squared error of the critic values and the average of the policy and advantage (difference between action value and state value) as a loss function for gradient descent.

3.2 Training

For training, each agent attempts the CartPole environment and explores actions in its current policy until episode completion. The episode ends when the agent fails the task or performs a maximum of 500 transitions. When the episode completes the network weights are updated following each algorithm and this process repeats with a subsequent episode. Each agent continues to complete episodes in batches of 10 until they achieve proficiency or they complete 5000 total episodes.

3.3 Proficiency and Evaluation

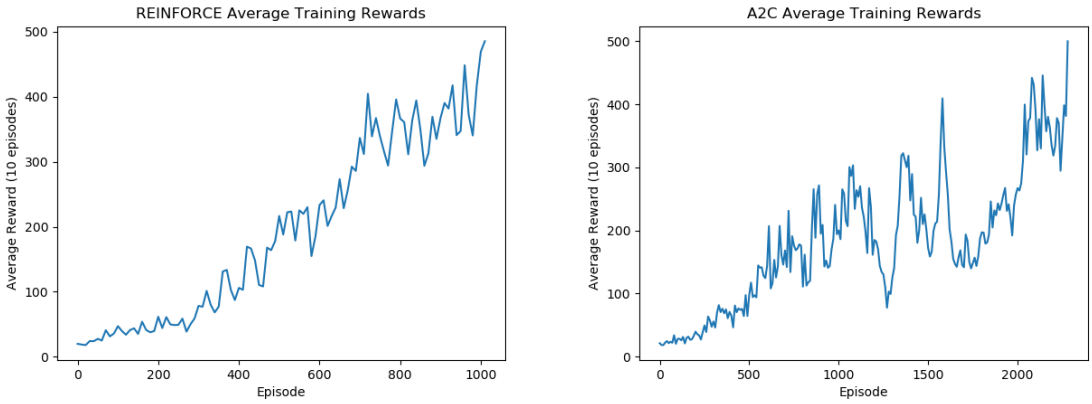
Each agent was deemed proficient when the average reward the algorithm attained over a 10 episode period exceeded the reward threshold of the environment. In the case of CartPole, once each agent attained an average reward greater than 475 (i.e. the agents successfully balance the pole for 475 transitions), they were considered proficient at the task. To evaluate and compare the policies learned by each algorithm, once proficient each agent was subjected to a 100-episode series where they acted greedily on policy actions to assess the average reward attained playing optimally.

4 Results

Comparing both algorithms, the number of episodes required to reach proficiency are plotted for each agent below. Once both REINFORCE and A2C agents were proficient at the CartPole task, a final evaluation was performed acting on the optimal policy and the average reward attained is shown.

4.1 Training Comparison

Plots 1, 2: *Average reward attained in 10-episode batches until each agent reached proficiency*



Both REINFORCE and A2C algorithms are able to train efficiently and effectively to improve their performance in the CartPole task. Note in plot 1, REINFORCE shows a somewhat noisy trend toward

Table 2: Average reward over 100 episodes of acting greedily on the policy for each proficient agent

Agent	Average Reward
REINFORCE	500.0
A2C	500.0

attaining the maximum reward of 500. On the other hand, A2C has a much more volatile and noisy path before reaching this same proficiency. At multiple times the average reward for A2C spiked to a local maximum, then degraded back to a sub-optimal policy. Lastly, it is important to note that A2C required roughly twice as many training episodes to reach proficiency compared to REINFORCE.

4.2 Optimal Policy Evaluation

Once both the REINFORCE and A2C agents achieved proficiency during training, the optimal policy was evaluated for 100 episodes with each agent exploiting their best predicted actions. Table 2 shows that each agent achieved the maximum possible reward during this evaluation period.

5 Analysis

The following provides potential reasoning for the observed results. In particular, explanations are provided justifying proficiency evaluation, training (in)efficiencies, and further work to explore to better understand both algorithms in alternative continuous action space settings.

5.1 Algorithm Proficiency and Optimal Policies

Both REINFORCE and A2C became proficient in the CartPole task and consistently achieve the maximum reward attainable in the environment when acting on optimal policies. This supports the notion that despite A2C experiencing a noisier trajectory to reach a proficient optimal policy, once each agent becomes proficient its greedy policy is sufficient to complete the task without error.

5.2 A2C Training Efficiency

The most notable observation between REINFORCE and A2C is the apparent training inefficiency experienced by A2C to reach proficiency. A2C showed a very volatile path to proficiency. Some possible explanations for this are inadequate hyperparameter tuning or thrashing weights due to a shared optimizer and learning rate incompatible for both the actor and the critic networks. Learning rates were kept consistent because of the shared actor between both algorithms, but in retrospect both agents likely learn different underlying functions and thus require unique configurations.

Additionally, A2C required more than twice as many training episodes to reach proficiency than REINFORCE. One potential explanation for this is the fact that A2C had a strictly more complex model to train compared to REINFORCE to complete the same task. Having to learn weights to approximate the value function for the CartPole environment, while useful for gauging whether individual actions made in an episode were beneficial, intuitively requires increased training time to become effective. With the relatively simplistic CartPole environment and the great performance of REINFORCE, learning only the policy is likely adequate for the task rather than incurring the overhead of learning the critic of the policy as well.

5.3 Discrete vs. Continuous Action Spaces

Given the increased training time required to reach proficiency for A2C and the researched and theoretical benefit of including a critic to improve training, this raises the obvious question of whether an environment like CartPole is sufficient to properly assess the effectiveness of each algorithm. Previous research suggests that policy gradient methods are more effective for environments that have continuous action spaces [4]. The CartPole environment presents two discrete actions is relatively simple in this regard. Therefore, planned future work to further compare the effectiveness of both algorithms will assess the performance of both in the context of a continuous action space.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013, December 19). Playing Atari with Deep Reinforcement Learning. Retrieved from https://arxiv.org/pdf/1312.5602.pdf

[2] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., ... Kavukcuoglu, K. (2016, June 16). Asynchronous Methods for Deep Reinforcement Learning. Retrieved from https://arxiv.org/pdf/1602.01783.pdf

[3] OpenAI. CartPole-v1. Retrieved from https://gym.openai.com/envs/CartPole-v1/

[4] Simonini, T. (2018, July 26). An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog! Retrieved from https://medium.com/free-code-camp/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d

[5] Sutton, R. S., McAllester, D., Singh, S., Mansour, Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. Retrieved from http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf

[6] Wu, Y., Mansimov, E., Liao, S., Radford, A., Schulman, J. (2017, August 18). OpenAI Baselines: ACKTR A2C. Retrieved from https://openai.com/blog/baselines-acktr-a2c/

Appendix

REINFORCE Implementation

```
from config import *
from util import *


class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim, dropout):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.dropout = nn.Dropout(p=dropout)
        self.linear2 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.dropout(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.softmax(x, dim=0)

        return x

    def get_action(self, state, greedy=False):
        probs = self(state)

        distribution = Categorical(probs)
        action = distribution.probs.max(-1)[1] if greedy else distribution.sample()

        log_prob = distribution.log_prob(action)

        return action.item(), log_prob.squeeze(0)


def reinforce_update(optimizer, policy, rewards, log_probs):
    returns = calculate_returns(rewards)

    log_probs = np.array(log_probs)
    returns = np.array(returns)

    policy_loss = -log_probs * returns

    optimizer.zero_grad()
    policy_loss = torch.stack(policy_loss.tolist()).sum()
    policy_loss.backward()
    optimizer.step()


def calculate_returns(rewards, normalize=True):
    returns = []

    for i in range(len(rewards)):
        reward_return = 0
        for i, r in enumerate(rewards[i:]):
            reward_return += GAMMA ** i * r

        returns.append(reward_return)

    returns = torch.Tensor(returns)
    if normalize:
        returns = (returns - returns.mean()) / (returns.std() + VERY_SMALL_NUMBER)

    return returns
```

```
def REINFORCE(policy, optimizer):
    episode_rewards = []

    for episode in range(1, EPISODES):
        state = env.reset()
        state = torch.Tensor(state)

        log_probs = []
        rewards = []

        for step in range(1, MAX_STEPS):
            if TRAIN_RENDER:
                env.render()

            action, log_prob = policy.get_action(state)

            new_state, reward, done, _ = env.step(action)
            new_state = torch.Tensor(new_state)

            log_probs.append(log_prob)
            rewards.append(reward)

            state = new_state

            if done:
                break

        episode_reward = np.sum(rewards)
        episode_rewards.append(episode_reward)

        reinforce_update(optimizer, policy, rewards, log_probs)

        if episode % LOG_INTERVAL == 0:
            print_episode_results(episode, episode_rewards, LOG_INTERVAL)

        if np.mean(episode_rewards[-PROFICIENCY:]) > env.spec.reward_threshold:
            print_solved_results(episode, episode_rewards, PROFICIENCY)
            break

    return episode_rewards
```

A2C Implementation

```
from config import *
from reinforce import Actor
from util import *


class Critic(nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(Critic, self).__init__()

        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)

        return x.detach().item()


def a2c_update(optimizer, rewards, log_probs, state_values):
```

```
    q_values = calculate_q_values(rewards)

    log_probs = torch.stack(log_probs)
    state_values = torch.Tensor(state_values)

    advantages = q_values - state_values

    value_loss = (advantages ** 2).mean()
    policy_loss = (-log_probs * advantages).mean()

    optimizer.zero_grad()
    (policy_loss + value_loss).backward()
    optimizer.step()


def calculate_q_values(rewards):
    q_value = 0
    q_values = [0] * len(rewards)
    for i in range(len(rewards) - 1, -1, -1):
        q_value = rewards[i] + GAMMA * q_value
        q_values[i] = q_value

    return torch.Tensor(q_values)


def A2C(actor, critic, optimizer):
    episode_rewards = []

    for episode in range(EPISODES):
        state = env.reset()
        state = torch.Tensor(state)

        log_probs = []
        rewards = []
        state_values = []

        for step in range(MAX_STEPS):
            if TRAIN_RENDER:
                env.render()

            value = critic(state)
            action, log_prob = actor.get_action(state)

            new_state, reward, done, _ = env.step(action)
            new_state = torch.Tensor(new_state)

            rewards.append(reward)
            state_values.append(value)
            log_probs.append(log_prob)

            state = new_state

            if done:
                break

        episode_reward = np.sum(rewards)
        episode_rewards.append(episode_reward)

        a2c_update(optimizer, rewards, log_probs, state_values)

        if episode % LOG_INTERVAL == 0:
            print_episode_results(episode, episode_rewards, LOG_INTERVAL)

        if np.mean(episode_rewards[-PROFICIENCY:]) > env.spec.reward_threshold:
            print_solved_results(episode, episode_rewards, PROFICIENCY)
```

```
            break

    return episode_rewards
```

Configuration

```
import gym
import numpy as np
import torch.multiprocessing as mp

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
import matplotlib.pyplot as plt

EPISODES = 5000
MAX_STEPS = 501
PROFICIENCY = 10
TEST_EPISODES = 10

GAMMA = 0.99
LOG_INTERVAL = 10
PLOT_COUNT = 1

TRAIN_RENDER = False
TEST_RENDER = True

VERY_SMALL_NUMBER = 1e-9

# Neural Network Constants
HIDDEN_DIM = 128
DROPOUT = 0.6
LEARNING_RATE = 1e-3
WEIGHT_DECAY = 1e-4

env = gym.make('CartPole-v1')
env.seed(0)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
```