Policy Gradient Methods in Deep Reinforcement Learning

A Master's Project
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mathematics

---

by
Sebastián Gracia Guzman
May 2021

Accepted by:
Dr. Yuyuan Ouyang, Committee Chair
Dr. Margaret Wiecek
Dr. Xin Liu

Abstract

Machine learning is a broad field aimed at the theoretical and practical study of constructing computer programs with the ability to independently learn and improve. In this framework, learning refers to improving an AI agent's performance for a given task. Within machine learning, there are three major distinct categories: supervised learning, unsupervised learning, and reinforcement learning. In reinforcement learning (RL), an agent is introduced to an environment and must determine which actions to take that will yield the greatest rewards. The agent is not given which actions incur a reward or penalty but instead must react to the environment and learn based on previous experiences. This project enters the diverse and growing field of RL with an overview of the mathematical theory and an implementation of an RL algorithm for the classic Atari game, Pong. The versatility of RL is also showcased in an optimal control task using a realistic physics simulation engine. The scope of this project covers RL algorithms that use the Policy Gradient Theorem. First, necessary background information is covered then, and important theoretical ideas are introduced. Afterwards, algorithms for RL are introduced and discussed. Then, applications of the algorithms are described along with performance measures. Results are then discussed and analyzed. This project assumes a basic understanding of probability, mathematical programming, and computer science.

Acknowledgments

I would like to thank Clemson University and the School of Mathematical and Statistical Sciences for the opportunity to pursue a masters as well as the financial support provided. Clemson University is acknowledged for generous allotment of computing time on the Palmetto cluster. I would like to thank my advisor, Dr. Ouyang, for his guidance through each stage of the process. I would also like to thank the other members of the committee, Dr. Wiecek and Dr. Liu. Both professors' expertise have been invaluable in my growth as a mathematician. A special thanks to my professors from University of Portland for challenging me and encouraging me to pursue graduate studies. Lastly, I would like to acknowledge the unwavering support from my friends and family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the field of reinforcement learning (RL), the goal of any feasible algorithm would be to produce an agent that efficiently takes in environmental states and takes appropriate actions based on said states. In a baseline reinforcement learning model there are several components including a *policy, reward signals*, and *value function*. An agent receives a reward signal from the environment after each action. The goal of the agent is to take actions that will yield the best rewards. There are numerous algorithms for reinforcement learning, and in this project, the policy-based methods which use the Policy Gradient Theorem are discussed. A mathematical theory provides a foundation for the formulation of RL. Then, implementation occurs using concepts from computer science. First, this section will cover the mathematical framework on which RL is built from.

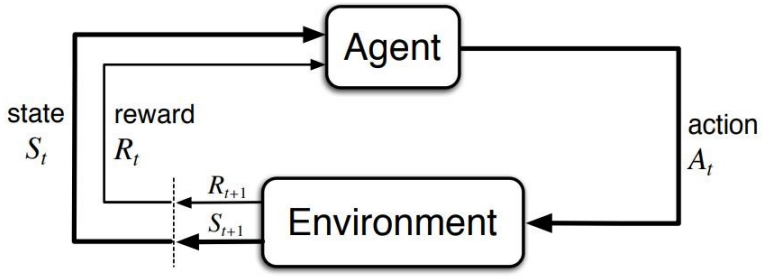


Figure 1.1: Agent environment interaction (figure from [12]).

1.1 Markov Decision Processes

It is useful to discuss Markov Decision Processes (MDP) as many reinforcement learning problems can be modeled using MDP. A *Markov Process* is a sequence of random states with the Markov property: a series of states $S_1, S_2, \ldots$ is Markov if and only if $\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$. That is, that the probability of a future state is only dependent on the previous state. An MDP can be described by a state space S, an action space A, a transition function $p(s'|s, a)$, and a reward function $R(s, a)$. At every time step t of an episode, an agent takes an action $a \in A$ and then receives information from the environment on their current state, $s \in S$, and reward, $r \in R$.

1.2 Environment

The environment is independent and separate from any algorithms or formulations that go into the agent. This project will assume that environments are fixed, and the goal remains to have an agent which acts optimally in the environment. In practice, when applying RL methods to a novel task, a user may have the ability to engineer the environment. How an environment should give rewards signals, referred to as reward engineering, is outside the scope of this project. For this project, the environments have already been designed and engineered by OpenAI Gym[1].

1.3 Policy Function

A *policy* is a way for an agent to determine actions based on a given state. In a deterministic policy, a state is mapped directly to an action. Alternatively, a stochastic policy maps a state to a probability distribution of actions.

$$\pi(s) = a \qquad \text{(Deterministic Policy)}$$

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \qquad \text{(Stochastic Policy)}$$

In most RL algorithms, a stochastic policy is used since often a deterministic policy can be exploited. For example, an agent acting under a deterministic policy playing rock-paper-scissors would not fare well. Another advantage is that a stochastic policy will also periodically explore other actions which could lead to greater reward. Exploration is a key component to reinforcement learning. Stochastic policies have exploration embedded into the policy so it is one less hyperparameter that requires tuning when training an agent. However if the true optimal policy is deterministic, policy gradient methods will converge to a deterministic policy. What we seek is to obtain an *optimal* policy in which actions taken by an agent acting under this policy maximizes expected rewards.

Now that our objective of obtaining an optimal policy is outlined, we can discuss how to achieve it. First it is important to be able to objectively decide when one policy is "better" than another.

1.4 Value Function

A value function $v(s)$ gives the expected value or return from a given state s. This introduces a notion for measuring how valuable or desirable a certain state is.

$$v(s) = \mathbb{E}[G_t|S_t = s]. \tag{1.1}$$

Where G_t is the discounted future reward going forward from time step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

Where we are summing the *discounted* return of R. We are applying a notion of diminishing return by setting our discount rate parameter, $0 \leq \gamma < 1$. It is important to incorporate this discount in order to determine the present value of future rewards. If $\gamma = 0$, we have an agent that behaves myopically [12] and only seeks to maximize immediate rewards. An agent might not pursue other actions if it lands in state with a positive reward,

despite greater rewards elsewhere. As γ approaches 1, the influence of future rewards are weighted more heavily.

Based on the above discussion, we can rewrite (1.1) as:

$$v(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} | S_t = s\right]. \tag{1.2}$$

When acting under a specific policy, π, we arrive at the *state value function for policy* π:

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} | S_t = s\right]. \tag{1.3}$$

Here $v_\pi(s)$ is the value at state s when acting under policy π. In other words, $v_\pi(s)$ represents the expected return starting from state s and taking actions according to policy π until terminal. Value functions allow us to define a partial ordering over all policies:

$$\pi_1 \succcurlyeq \pi_2 \iff v_{\pi_1}(s) \geq v_{\pi_2}(s) \quad \forall s \in S.$$

Finally, we can define the notion of an *optimal* policy that produces the largest reward.

$$v^*(s) = \max_\pi v_\pi(s) \quad \forall s \in S.$$

Here we are guaranteed to have an optimal policy π.

Theorem 1.4.1 *For any MDP, there exists an optimal policy,* π_* *such that* $\pi_* \succcurlyeq \pi \quad \forall \pi$.

The proof is omitted and can be found for example in [12]. Finding an optimal policy is what policy based reinforcement learning methods are after. In later chapters the methodology of obtaining the optimal policy will be explored.

1.4.1 State Action Value function

We can also formulate the expected reward from performing action a when in state s acting under policy π.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{1.4}$$

Similar to the value function, the state action value function gives how valuable an action at a specific state is when acting under a policy π. Since the state action values, or q-values, will give the expected reward for each action at a state s, it is possible to formulate a deterministic policy easily from the q function.

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, q(s,a)$$

Such a policy is referred to as a greedy policy because we are simply taking the best action based upon the q-function. There exists families of algorithms relying upon this methodology called q-learning methods. The focus of this project is on policy based methods in which the policy is formulated and updated directly instead of through the q-function, so we will not cover q-learning algorithms. The q-function, however, is still important in policy based methods as this project will later discuss.

1.5 Transition Probability Function

Define the transition probability of moving from current state s, to subsequent state s', with reward r, from action a, as follows:

$$p(s', r|a, s) \doteq \mathbb{P}\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1.5}$$

Then we can define the state transition probability as:

$$p(s'|a,s) \doteq \mathbb{P}\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r\in\mathcal{R}} p(s', r|a, s). \tag{1.6}$$

In most cases an agent will not have full access to the MDP and therefore will not have access to the transition probability. However the transition probability function has usefulness in proofs as will be shown in the later sections.

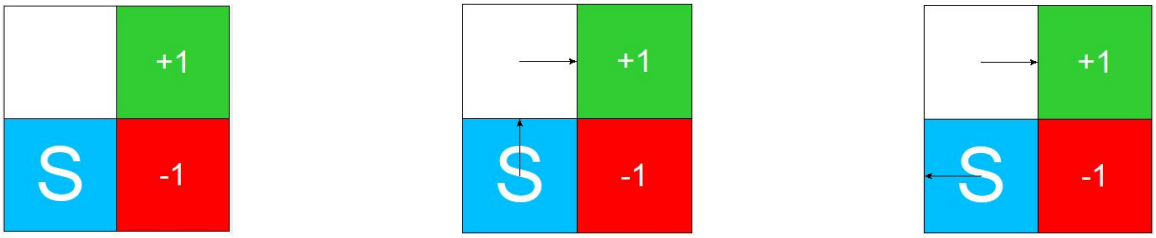
1.6 Exact Methods

In the case in which the agent does have full access to the MDP transition probability function, the environment can be solved with an optimal solution. Solving an environment refers to obtaining the optimal policy or plainly knowing how to act under every state to maximize rewards. We can rewrite the optimal value function with the transition probability as:

$$\begin{aligned}
v^*(s) &= \max_a q_{\pi^*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}[G_t|S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots|S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots)|S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma v^*(S_{t+1})|S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s')]
\end{aligned}$$

The last equation is the *Bellman optimality equation* for the value function. Since the transition probabilities are known, then we can extract a recursive formulation to obtain $v^*(s)$ and therefore obtain π^*. This can be represented as nonlinear equations with $|S|$, number of non-terminal states, unknowns.

1.6.1 Gridworld Example



(a) (b) (c)

Figure 1.2: (From left to right) Trivial Gridworld environment. Solved deterministic Gridworld environment. Solved Gridworld environment with .3 action noise.

Consider a simple finite MDP environment given by figure 1.2. Here an agent starts at state (1,1), the bottom left square labeled "S". The state (2,2) , top right, is a terminal state that awards the agent +1 reward and the state (1,2), bottom right, is also a terminal state that awards the agent -1. The white state (1,2), top left, gives 0 reward. At any given non-terminal state the agent can take actions: *north, south, east, west.* If an agent takes an action in a direction away from a possible state the agent will not move. For example, an agent selecting the west action in state (1,2) will remain in state (1,2). In this example consider $\gamma = .9$.

Terminal states are considered a state where the agent takes no action and receives the specified reward. That is for calculation purposes, the agent does not receive the reward when moving to a terminal state. The agent would receive the reward in the next action-reward pair. The Bellman optimality equations for this problem are the following:

$$v^*(2,2) = 1$$

$$v^*(2,1) = -1$$

$$v^*(1,2) = \max\{0.9v^*(1,2), 0.9v^*(1,1), 0.9v^*(2,2), 0.9v^*(1,2)\}$$

$$v^*(1,1) = \max\{0.9v^*(1,2), 0.9v^*(1,1), -0.9v^*(2,1), 0.9v^*(1,1)\}.$$

$$v^*(1,2) = \max\{0.9v^*(1,2), 0.9v^*(1,1), 0.9\}$$
$$v^*(1,1) = \max\{0.9v^*(1,2), 0.9v^*(1,1), -0.9\} = \max\{0.9v^*(1,2), 0.9v^*(1,1)\}.$$

The last equality above is since $v^*(1,2) \geq 0.9$ from the first equation. Note that the above system also implies that:

$$v^*(1,2) = \max\{v^*(1,1), 0.9\}$$
$$v^*(1,1) = \max\{0.9v^*(1,1), 0.81\},$$

and consequently $v^*(1,2) = 0.9$ and $v^*(1,1) = 0.81$. Solving the equation by hand is straightforward, since the transition probability term will always be 1 for the environment. Now consider a new environment in which there is 0.3 noise in the environment. Specifically, if an agent selects an action there is a 0.3 probability that any adjacent action is taken instead. For example if an agent selects action north there is 0.3 probability of taking actions east or west. The action opposite in direction of the selected action will not be selected. Now the system of equations becomes:

$$v^*(2,2) = 1$$

$$v^*(2,1) = -1$$

$$v^*(1,2) = \max\left\{\begin{array}{l} .85 * .9v^*(1,2) + .15 * .9v^*(2,2), \\ .7 * .9v^*(1,1) + .15 * .9v^*(2,2) + .15 * .9v^*(1,2), \\ .85 * .9v^*(2,2) + .15 * .9v^*(1,1), \\ .85 * .9v^*(1,2) + .15 * .9v^*(1,1) \end{array}\right\}$$

$$v^*(1,1) = \max \left\{ \begin{array}{l} .7 * .9v^*(1,2) + .15 * .9v^*(2,1) + .15 * .9v^*(1,1), \\ .85 * .9v^*(1,1) + .15 * .9v^*(2,1), \\ .85 * .9v^*(2,1) + .15 * .9v^*(1,1), \\ .85 * .9v^*(1,1) + .15 * .9v^*(1,2) \end{array} \right\}.$$

After solving the system of equations numerically, we can obtain the values for the unknowns:

$$v^*(1,2) \approx .8 \qquad v^*(1,1) \approx .45.$$

The system of equations is still with 2 unknowns but notice how introducing stochasticity increases the complexity of the system. But it is evident that increasing the dimensions of states and actions will result in large systems of equations to solve. Consider the world given by figure 1.3. With 14 non-terminal states the complexity of the system increases substantially and solving the problem using the above method quickly becomes infeasible.

1.6.2 Value Iteration

Solving the Bellman optimality equation is only feasible for smaller MDPs as in general there is no closed form solution [10]. Introduce the notion of iterations of the value function $v_k^*(s)$. Where $v_k^*(s)$ is total expected rewards starting from s, acting optimally for k steps. Thus trivially we have $v_0^*(s) = 0 \ \ \forall s \in S$ since the agent is already at a terminal

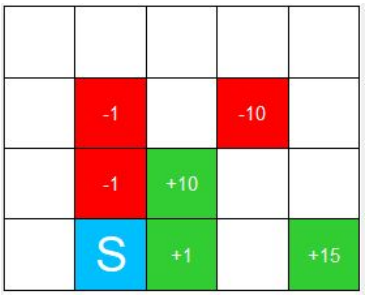
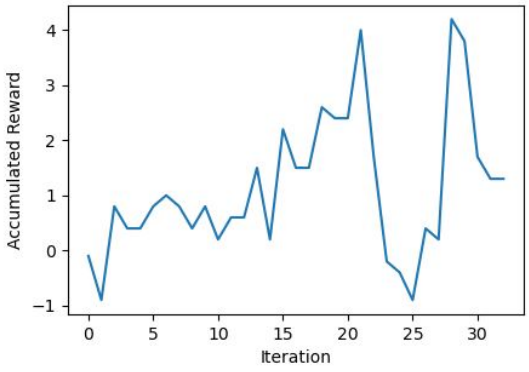
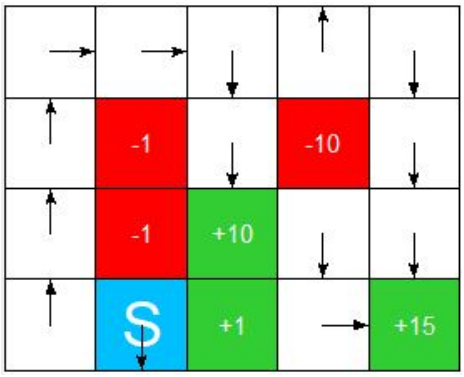


Figure 1.3: Non-trivial Gridworld environment.



(a) Rewards from value iteration algorithm.



(b) Solved Gridworld environment with .3 action noise and $\gamma = .9$.

Figure 1.4: The value iteration solved the more complicated Gridworld environment in 32 iterations. (Code to visualize results from [15]).

state and can't act further. Then:

$$v_1^*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_0^*(s')]$$

This can be extended up to a horizon H. Employing this iterative method of calculating

Algorithm 1 Value Iteration [4]

Initialize value function $v_0(s) = 0 \quad \forall s \in S$
for $k = 1, 2, \ldots H$ **do**
  for each state in $s \in S$ **do**
    $v_k^*(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{k-1}^*(s')]$.
    $\pi_k^*(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{k-1}^*(s')]$
return π_k^*

$v_k^*(s)$ leads to the value iteration algorithm. It is also simple to produce a deterministic policy from the algorithm. This algorithm has guaranteed convergence to $v(s)$ as $H \longrightarrow \infty$ [4]. In practice the loop is broken once successive value iterations fall within a predetermined tolerance. As discussed above, in practice the environment MDP is not always known and thus exact methods are not employed. Even if the probability function is known, state spaces dimensions can quickly become too large for value iterations. Consider the game of chess. Applying exact methods is infeasible for environments with large state

or action spaces. Thus there is a need for a system in which we can approximate the functions $\pi(a|s), v(s), q(s,a)$. These function approximations should be able to generalize over continuous states. This challenge will be discussed in the next section. The next section will also discuss methods for RL without access to the transition probabilities.

Chapter 2

Policy Based Methods

A class of methods that focus on optimizing the policy function are known as policy based methods. Unlike q-learning, the policy is directly improved instead of improving the q-function. Within policy based methods there are derivative methods and derivative free methods. Derivative free methods use black box optimization methods, such as genetic algorithms, while derivative methods use variants of gradient ascent. This section will discuss the latter methods.

2.1 Policy Function Approximation

As discussed in the previous section, a policy function, π, is the function that will map states to action probabilities. A policy function approximation is described by a vector of parameters θ. Neural networks are very good at approximating nonlinear functions and working with large amounts of data which make them prime candidates for policy functions. A policy parameterization can be achieved in multiple ways, but in practice θ usually represents the weights and biases of a deep neural network. Neural networks are beyond the scope of this project and only a brief overview will be given.

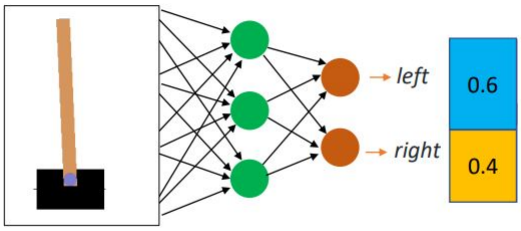


Figure 2.1: Neural Network taking in an image of the Cart-pole environment and mapping to an action probability distribution (figure from [3]). The goal of this environment is to keep the pole balanced by either going left or right.

2.1.1 Neural Networks

A neural network is a framework used in computer science modeled after neuron activation in biology. Neural networks are very flexible and can be modified in structure to approximate complex nonlinear functions. When applying deep learning to policy based methods for reinforcement learning, a neural network takes in a state as input. The formulation of the state can vary from either a discrete set of observation parameters to an entire frame or image in the case of deep learning. The input nodes are passed along a neural network's hidden layers to an output layer which has a probability distribution over actions. The weights and biases of each component of the neural network are encompassed in θ. A policy parameterized by θ is denoted by either $\pi_\theta(a|s)$ or $\pi(a|s;\theta)$. When applied in RL, the neural network can be used to model a policy function, value function, or state action value function. A neural network functions in the RL framework by mapping a state to a probability distribution, state value, or action values based on which function is being modeled. Neural networks map states through multiple hidden layers. More information on neural networks can be found at Stanford's open course on Convolutional Neural Networks [6], and the references within.

2.2 Objective function

The following discussion is based on Chapter 13 from [12]. In Policy Gradient methods, an agent is acting under a parameterized policy. A parameterized policy can sample action distributions without the need to evaluate the value function at each step. Even though value functions seem to be necessary to evaluate policies, we can instead introduce an objective function which depends solely on the policy parameter θ. This objective function, sometimes referred to as a score or loss function, $J(\theta)$, is a measure of how well our policy performs. Parameterizing our policy is advantageous theoretically as this allows us to compute the gradient of $J(\theta)$ which will lead to policy improvements. As previously discussed, policy based methods are variants of gradient ascent methods for solving optimization problems:

$$J(\theta) \doteq v_\pi(s_0).$$

That is, the expected rewards acting under policy π, starting from the initial state, s_0. Our goal becomes to maximize the objective function and henceforth find the optimal policy:

$$\max_\theta J(\theta).$$

We are looking to optimize policy π parameterized by θ. In the cases where the gradient might not be available, genetic algorithms or action space search methods might be used. Also given a simple parameterization, simplex methods can also be used. For the rest of this project it will be assumed θ is parameterized by the weights of a deep neural network. As will be shown, the Policy Gradient Theorem nicely provides the ability to use gradient ascent.

2.3 Policy Gradient

Given a parameterization θ, the gradient of $J(\theta)$ is the direction of greatest local increase in objective function value:

$$\nabla_\theta J(\theta) = \left[\frac{\partial J(\theta)}{\partial \theta_1}, \frac{\partial J(\theta)}{\partial \theta_2}, \ldots, \frac{\partial J(\theta)}{\partial \theta_n}\right]^T$$

As discussed above, θ represents parameters defining a deep neural networks in most cases.

Calculating the gradient of the objective function is expensive since it involves evaluating the value function which is the expectation of an infinite sum. It may not have a closed form description and might be computationally expensive to estimate. Fortunately the Policy Gradient Theorem provides the expression $\nabla J(\theta)$ as an expectation. While it may still be computationally expensive to estimate, it provides a possibility for us to implement stochastic gradient ascent.

2.4 Policy Gradient Theorem

Theorem 2.4.1 *Assume that the policy $\pi_\theta(s, a)$ is differentiable. Then for objective function $J(\theta)$, the gradient is proportional to the state value function and the gradient of the policy:*

$$\nabla J(\theta) \quad \propto \quad \sum_s \mu(s) \sum_a \nabla \pi_\theta(a|s) q(s, a).$$

In the above theorem statement, $\mu(s)$ is the state distribution under π of an episode. An episode refers to a single finite MDP. In the case of a continuous MDP, the policy gradient

can be further simplified since $\sum_s \mu(s) = 1$, and consequently:

$$\begin{aligned}\sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s;\theta) &= \sum_a q_\pi(s,a) \nabla \pi(a|s;\theta) \\ &= \sum_a q_\pi(s,a) \pi(a|s;\theta) \frac{\nabla \pi(a|s;\theta)}{\pi(s|a;\theta)} \\ &= \sum_a \pi(s|a;\theta) q_\pi(s,a) \nabla \log \pi(a|s;\theta) \\ &= \mathbb{E}_\pi \left[q_\pi(s,a) \nabla \log \pi(a|s;\theta) \right].\end{aligned}$$

Multiplying and dividing the first equality by $\pi(s|a;\theta)$ leads to the second equality. Then since $\nabla \log(f) = \frac{1}{f} \cdot \nabla f$ we can simplify the equation to instead use log probabilities. We describe the proof of the theorem below.

Proof of Theorem 2.4.1

Start with the gradient of the value function:

$$\begin{aligned}\nabla v(s) &= \nabla_\theta \left(\sum_{a \in A} \pi(a|s;\theta) q(s,a) \right) \\ &= \sum_{a \in A} \left(\nabla_\theta \pi(a|s;\theta) q(s,a) + \pi(a|s;\theta) \nabla_\theta q(s,a) \right) \\ &= \sum_{a \in A} \left(\nabla_\theta \pi(a|s;\theta) q(s,a) + \pi(a|s;\theta) \nabla_\theta \sum_{s',r} p(s',r|s,a)(r + v(s')) \right) \\ &= \sum_{a \in A} \left(\nabla_\theta \pi(a|s;\theta) q(s,a) + \pi(a|s;\theta) \sum_{s',r} p(s',r|s,a) \nabla_\theta v(s') \right) \\ &= \sum_{a \in A} \left(\nabla_\theta \pi(a|s;\theta) q(s,a) + \pi(a|s;\theta) \sum_{s'} p(s'|s,a) \nabla_\theta v(s') \right).\end{aligned}$$

From the first equality we use the product rule to expand the expression. Then from equality 3 to 4 we can simplify since $p(s', r|s, a)$ is not a function of θ. We can then simplify line 4 by equation (1.6). For better bookkeeping, let $\phi(s) = \sum_{a \in A} \nabla_\theta \pi(a|s;\theta) q(s,q)$. Continuing

from the prior equation we have:

$$\begin{aligned}
&\sum_{a\in A}\left(\nabla_\theta \pi(a|s;\theta)q(s,a) + \pi(a|s;\theta)\sum_{s'} p(s'|s,a)\nabla_\theta v(s')\right) \\
&= \phi(s) + \sum_{a\in A}\pi(a|s;\theta)\sum_{s'} p(s'|s,a)\nabla_\theta v(s') \\
&= \phi(s) + \sum_{a\in A}\sum_{s'} \pi(a|s;\theta)p(s'|s,a)\nabla_\theta v(s').
\end{aligned}$$

Define $\rho^\pi(s \to x, k)$ as the probability of transitioning from state s to state x in k steps under policy π. Thus $\rho^\pi(s \to s', k=1) = \sum_a \pi(a|s;\theta)p(s'|s,a)$. This gives a recursive formulation of all transition probabilities as $\rho^\pi(s \to x, k{+}1) = \sum_{s'} \rho^\pi(s \to s', k)\rho^\pi(s' \to x, 1)$ Expanding the right hand side yields,

$$\begin{aligned}
&\phi(s) + \sum_{a\in A}\sum_{s'} \pi_\theta(a|s)p(s'|s,a)\nabla_\theta v(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\nabla_\theta v(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\left[\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1)\nabla_\theta v(s'')\right] \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s'} \rho^\pi(s \to s', 1)\rho^\pi(s' \to s'', 1)\nabla_\theta v(s'') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s'} \rho^\pi(s \to s'', 2)\nabla_\theta v(s'') \\
&\vdots \\
&= \sum_{x\in S}\sum_{k=1}^{\infty} \rho^\pi(s \to x, k)\phi(s),
\end{aligned}$$

where the last equality is by unrolling all the gradients of the value function in the previous equality. Applying our formulation to the initial state value function we have:

$$\nabla_\theta v(s_0) = \sum_{x\in S}\sum_{k=1}^{\infty} \rho^\pi(s_0 \to x, k)\phi(s).$$

Let $\eta(s) = \sum_{k=1}^{\infty} \rho^{\pi}(s_0 \to x, k)$ thus we have:

$$\begin{aligned}
\sum_{x \in S} \sum_{k=1}^{\infty} \rho^{\pi}(s_0 \to x, k)\phi(s) &= \sum_{x \in S} \eta(s)\phi(s) \\
&= \left(\sum_{x \in S} \eta(s) \right) \sum_{s} \frac{\eta(s)}{\sum_{x \in S} \eta(s)} \phi(s) \\
&\propto \sum_{s} \frac{\eta(s)}{\sum_{x \in S} \eta(s)} \phi(s) \\
&= \sum_{s} \mu(s) \sum_{a \in A} \nabla_{\theta} \pi(a|s;\theta) q(s,a).
\end{aligned}$$

The second equality we are normalizing η so that it is a probability distribution. At the third line since $\sum_{x \in S} \eta(s)$ is a constant we can rewrite the equality as a proportion. Finally, we let $\mu(s) = \sum_{s} \frac{\eta(s)}{\sum_{x \in S} \eta(s)}$, which is a stationary distribution. Thus we arrive at our desired result,

$$\nabla J(\theta) \propto \sum_{s} \mu(s) \sum_{a \in A} \nabla_{\theta} \pi(a|s;\theta) q(s,a).$$

□

Having the gradient of the objective function allows us to implement the gradient ascent algorithm. This iterative algorithm is very effective at finding a critical point. Below is a simple step wise algorithm that will improve our policy.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla J(\theta_t)$$

Here α is a user-defined step size. In machine learning this is a hyperparameter also referred to as the learning rate. Since the Policy Gradient Theorem gives us a proportional relation, the constant of proportionality can be absorbed into the step size. Using gradient ascent we know that the policy will converge efficiently to a critical point of modest accuracy. Evaluating $\sum_{a \in A} \nabla_{\theta} \pi(a|s;\theta) q(s,a)$ is still expensive however. For this reason stochastic

gradients need to be used. Different algorithms implement different formulations of the gradient but all rely on the Policy Gradient Theorem. In the next sections brief outlines of algorithms will be given.

Chapter 3

Algorithms

3.1 REINFORCE

The REINFORCE algorithm[14] is a straightforward implementation of the Policy Gradient Theorem based on Monte-Carlo methods. Monte-Carlo methods are based on the episodic sampling of actions taken under policy π. That is to act under policy π until one trajectory is completed, i.e a terminal state is reached, then update parameters. From the experiences we can calculate rewards and adjust the gradient accordingly. This is due to the fact that the gradient is exactly equal to the expectation of the sample gradient [5]:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}[q(s,a)\nabla \ln \pi(a|s)] \\ &= \mathbb{E}[\mathbb{E}[G_t|s_t,a_t]\nabla \ln \pi(a_t|s_t)] \\ &= \mathbb{E}[G_t \nabla \ln \pi_\theta(a_t|s_t)].\end{aligned}$$

Using the fact that $q(s_t,a_t) = \mathbb{E}[G_t|s_t,a_t]$ we can simplify the first equality. Then by the law of total expectation we simplify the second equality. REINFORCE is a variant of the gradient ascent method in which improvements are made to the policy parameter, θ, by calculating the gradient of $J(\theta)$ and performing an update. Although the algorithm provides an unbiased estimate of the gradient, drawbacks of the REINFORCE algorithm

Algorithm 2 REINFORCE

Initialize policy parameter θ

Initialize learning rate and discount factor α, γ

loop forever:

Generate an episode following policy $\pi_\theta : s_1, a_1, r_2, s_2, \ldots S_T$

for step in episode $t = 1, \ldots, T-1$ **do**

$G_t \leftarrow$ expected return at step t.

$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \log \pi(a_t, |s_t; \theta)$.

are slow convergence rate and high variance. Both issues stem from the fact that an entire episode must be collected before an update can be performed. Variance can be reduced by introducing a baseline. This idea of a baseline will be expanded in the next algorithm.

While the REINFORCE algorithm is not widely used today, it was one of the first RL algorithms to use the Policy Gradient Theorem. The REINFORCE algorithm demonstrates how the Policy Gradient Theorem is readily implemented in policy optimization.

3.2 Actor-Critic Methods

As discussed above, Monte-Carlo methods have high variance. This is due to the fact that RL trajectories can be vastly different from one another. To reduce variance a *critic* can be introduced. The critic can be based on the computation of estimated state values $v(s)$ or action state values $q(s, a)$ depending on the algorithm. One other component, the *actor*, updates the policy, π_θ based on feedback from the critic. Actor-critic algorithms follow different variants of an approximate policy gradient given by [11].

$$\nabla J(\theta) \approx \mathbb{E}[\nabla \ln \pi_\theta(a|s) q_\phi(s|a)]$$

Here ϕ are the parameters for the neural network used for the q-function. Since actor critic methods are using approximation of the policy gradient, the resulting gradient is a biased estimator. This can be mitigated with a careful selection of the value function

approximation [11].

3.3 A2C

The influential algorithm published in [7] stands for Asynchronous Advantage Actor Critic (A3C). As the name suggests, A3C is an actor-critic algorithm. The advantage portion of the name is for how the gradient is approximated.

$$\nabla_\theta J(\theta) \approx \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s;\theta) A(s,a)]$$

Where $A(s,a)$, known as the advantage function, is computed from the state action value function and value function. The advantage function can be thought of as the difference between the action taken and the best action being taken. This is due to the fact that $v(s) = \max_a q(s,a)$. In practice the advantage is estimated using the reward returns and a value function parameterized by ϕ:

$$A(s,a) = q(s,a) - v(s) \approx G_t - v_\phi(s_t)$$

At every update of the policy function this algorithm also updates the value function. Thus we need to construct an objective function for evaluating the value function. One possible objective function will simply be the mean squared error. Thus compared to the score function for the policy parameter, we aim to minimize instead of maximize.

$$L(\phi) \doteq (G_t - v_\phi(s))^2$$

Since we are updating using gradient descent, we again need the gradient of the objective function. Calculating the gradient of the value objective function is much more straight forward compared to the policy objective function.

$$\nabla_\phi L(\phi) = 2(G_t - v_\phi(s)) \nabla_\phi (G_t - v_\phi(s))$$

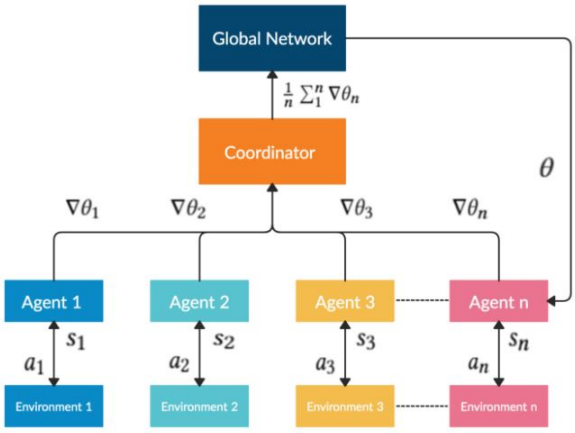


Figure 3.1: A2C Architecture overview (figure from [2]).

The last A in A3C stands for asynchronous meaning multiple agents are trained in parallel. Parallel training improves performance by decorrelating the experiences from one another. Since at any given time step, the agent will , most likely, be acting in different trajectories. This creates more stationarity in the experiences used for learning [7]. The A2C algorithm is a modification of A3C where the agents are trained synchronously, hence the third A is dropped to become A2C. Synchronous training has shown to have better convergence properties due to the fact that uniform policy updates are performed across all agents instead of at sporadic intervals [13].

Algorithm 3 Advantage Actor Critic

Initialize global policy network $\pi(a|s;\theta)$ and value function $v(s;\phi)$

Initialize n agents with access to π_θ and v_ϕ

Initialize batch size and discount factor $\mathcal{D}, \gamma$

repeat

for agent $i = 1, 2, \ldots, n$ in parallel **do**

Initialize step counter $t \leftarrow 0$

Reset environment and get initial state s_0

while $t < \mathcal{D}$ **do**

Perform action a_t according to policy $\pi_\theta(a_t|s_t)$

Receive reward r_t and new state s_{t+1}

$t \leftarrow t + 1$

if terminal state reached **then**

Reset environment and get initial state s

$$\mathrm{G} = \begin{cases} 0 & \text{for terminal } s_t \\ v_\phi(s_t) & \text{for non-terminal } s_t \end{cases}$$

Fit target values from experiences collected $(G \leftarrow r_i + \gamma G)$

Accumulate gradients w.r.t $\theta : \nabla\theta_i \leftarrow \theta + \nabla_\theta \log \pi_\theta(a_i|s_i)(G - v_\phi(s_i))$

Accumulate gradients w.r.t $\phi : \nabla\phi_i \leftarrow \phi + 2(G - v_\phi(s_i))\nabla_\phi(G - v_\phi(s_i))$

end for

Perform synchronous update using $\nabla\theta = \frac{1}{n}\sum_{i=1}^{n} \nabla\theta_i$ and $\nabla\phi = \frac{1}{n}\sum_{i=1}^{n} \nabla\phi_i$

Set new global policy π_θ and value function v_ϕ

Chapter 4

Implementation

The work in this project was completed using Python due to its extensive and readily available ML and RL packages. The implementation of these algorithms came from the Stable Baselines3 library(SB3). SB3 is a set of reliable implementations of reinforcement learning algorithms in PyTorch [8]. In SB3, A2C is already implemented with a deep neural network taking the game state as a vector of pixels as an input. Stable Baselines3 also includes environment preprocessing and neural network architecture. While it is possible to implement algorithms "from scratch", given the time constraints for this project, replicating results from the benchmarks was the aim of this implementation. The results obtained from training were both able to achieve established benchmarks for both tasks respectively.

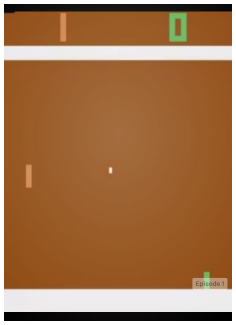


Figure 4.1: Pong Environment given by OpenAI Gym [1]

4.1 Pong

Pong is an Atari 2600 game in which a player must maneuver a paddle (either up or down) in order to beat a computer controlled opponent in virtual ping pong. The environment is created by Open AI's gym library. This allows the environment to be easily called and used.

4.1.1 Pong Environment Details

SB3 also works well with the gym environments to handle all prepossessing and image transformations in order to readily implement RL algorithms. Stable baselines implemented the ideas used by Google DeepMind [9] to stack frames and preprocess environments to optimize memory usage for all Atari 2600 games. Instead of having the entire video of the environment, preprocessing takes every 4 frames and superimposes them so that even though the input is a still image, motion is still preserved. Pong is an environment with a discrete action space with actions of *up, down,* or *no action*. The agent receives a positive 1 reward for scoring against the computer controlled opponent and a negative 1 reward for the computer scoring against the agent. All other rewards are zero.

4.1.2 Pong Results

From the training results we see that the agent had a difficult time getting past -20 reward spending around 1 million time steps without improvement. This makes sense intuitively since at first an agent will perform randomly only receiving a positive reward when it scores a point against its opponent. Since the agent is performing random actions this would take many iterations before occurring. But as we can see, after receiving positive rewards, the agent quickly begins to learn how to score points. After an initial rapid increase the agent's learning curve becomes more noisy. Ultimately the agent reaches maximum rewards in around 2.5 million time steps. Evaluating the agent at the end of training shows that the optimal policy was obtained as the agent received the maximum possible

Figure 4.2: Pong A2C Learning curve with 16 parallel workers

Mean Reward	Standard Deviation
20.00	± 0.00

Table 4.1: Evaluation of trained A2C agent across 10 episodes

reward of 20 with 0 standard deviation across 10 rollouts. Training was accomplished with 16 environments running in parallel. Total training time was around six hours using the Palmetto Cluster.

4.2 Pybullet

Pybullet is a physical simulation environment in which several different dynamical bodies can be used for optimal control tasks. For this project the A2C was implemented on the Half Cheetah Bullet Environment.

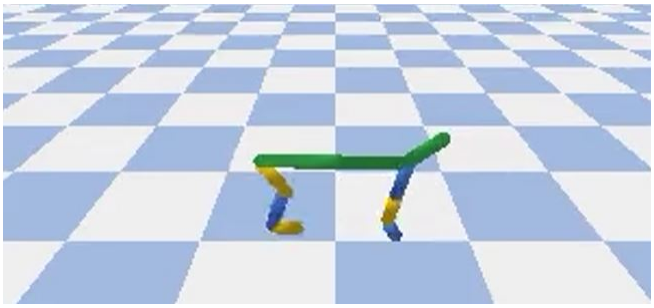


Figure 4.3: Half Cheetah Pybullet Environment [1]

Mean Reward	Standard Deviation
2001	± 67.3

Table 4.2: Evaluation of trained A2C agent across 10 episodes

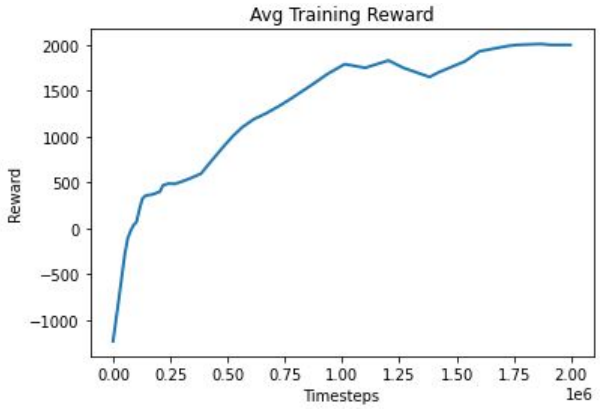


Figure 4.4: Half Cheetah A2C Learning curve with 4 parallel workers

4.2.1 Environment Details

The goal of this environment is for an agent to control a Half-Cheetah's limbs in order for the agent to move forward. The agent receives a positive reward for advancing forward. The reward signal is higher the faster the agent is moving forward thus encouraging the agent to learn the fastest way to run. The episode is terminated if the Half Cheetah's limbs are touching the ground for too long receiving a negative reward.

The agent has access to control all the limbs in Figure 4.3 which are not green. The agent receives 26 observations describing the state. The observations include continuous information about center of mass, orientation, velocity, etc. as well as booleans indicating which limbs are touching the ground. The action space is continuous since the agent must indicate how much torque to apply to each limb. The expected range for each of the six sub-actions is between [-1,1].

4.2.2 Results

After two million time steps the A2C agent reaches a reward of around 2000 matching the benchmarks set by SB3. The training time was approximately three hours. The learning curve shows a steady improvement with a peak reward of 2010. As with the A2C Pong results, rapid improvement is made at the onset. The learning curve is also much less noisy than the Pong implementation. This could be partly due to the fact that rewards from Pong can be sparse while this is not the case in the Half Cheetah environment.

4.3 Analysis of Results

Through both implementations it is made clear that reinforcement learning can be very sample inefficient. Especially in the Pong environment we can see that training takes a considerable number of iterations and total time. The different performance between algorithms also shows how fickle reinforcement learning can be. Machine Learning in general, and especially RL, is very sensitive to hyperparameter and algorithm choice. Certain algorithms can perform much better than others on certain environments but not on others. For the Half Cheetah environment we can see that the A2C environment was not able to reach an optimal policy. The best performing algorithm, Proximal Policy Optimization (PPO), reaches a peak reward of 2800 on the Half Cheetah environment. However, benchmarks for A2C have shown that high performance policies can be achieved in other Pybullet environments. Pong achieves perfect performance in around ten million time steps while the Half Cheetah reaches peak performance in around two million time steps.

Reinforcement learning depends on trial and error thus the need to collect many experiences is unavoidable. This downside will be mitigated as access and availability to more powerful CPU/GPU increases.

Chapter 5

Conclusion

5.1 Conclusion

With the help of the Policy Gradient Theorem, there is a whole class of methods in reinforcement learning focusing on policy optimization involving the gradient. An advantage of policy gradient methods is that it is fairly straightforward to implement and grasp. Where other methods might seek to maximize the value function v or action value function, which would then in turn maximize policy, the aim is to optimize the policy directly. Policy based methods are also better equipped to achieve truly random optimal policies. Policy based methods can also handle both continuous and discrete action spaces. The Policy Gradient Theorem very conveniently allows the gradient of the parameterized objective function be analytically calculated. Policy gradient methods also fall short precisely where its main strength lies. Because of its reliance on gradient ascent, stochastic gradient ascent methods are not guaranteed to converge to global maximum. Convergence is to a critical point which may or may not be global max. However this is a problem not unique to policy based methods and many machine learning methods share this shortcoming.

Reinforcement learning is currently going through a period of rapid interest and expansion. This is due to the advances in deep learning which has rapidly opened the field of machine learning in general. Deep learning is a framework in which very large

quantities of data are taken in as input to a neural network. Deep learning has expanded the applications of RL to many novel tasks. Despite the nature of designing algorithms to play trivial video games the possible applications are countless. The ability to consistently create agents which can learn without the need of supervision is incredibly powerful. Recent applications of reinforcement learning include automated stock trading [2], autonomous driving, and robotics. The vast applications of reinforcement learning are all possible due to the mathematical framework which describe complex agent-environment interactions.

Appendices

Appendix A Value Iteration

The following code is an implementation of the value iteration algorithm. The following code was produced to solve the grid-world problem posed in Project 1 in CPSC 8810. Starter code is given by Pei Xu (peix@g.clemson.edu) and Ioannis Karamouzas (ioannis@g.clemson.edu)

```
# grid_world.py
# ----------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to Clemson University and the authors.
#
# Authors: Pei Xu (peix@g.clemson.edu) and
Ioannis Karamouzas (ioannis@g.clemson.edu)
"""
The package 'matplotlib' is needed for the program to run.

The Grid World environment has discrete state and action spaces
and allows for both model-based and model-free access.

It has the following properties:
    env.observation_space.n     # the number of states
    env.action_space.n          # the number of actions
    env.trans_model             # the transition/dynamics model

In value_iteration and policy_iteration, you can access the transition model
at a given state s and action by calling
```

```
    t = env.trans_model[s][a]
where s is an integer in the range [0, env.observation_space.n),
      a is an integer in the range [0, env.action_space.n), and
      t is a list of four-element tuples in the form of
        (p, s_, r, terminal)
where s_ is a new state reachable from the state s by taking the action a,
      p is the probability to reach s_ from s by a, i.e. p(s_|s, a),
      r is the reward of reaching s_ from s by a, and
      terminal is a boolean flag to indicate if s_ is a terminal state.

In mc_control, sarsa, q_learning, and double q-learning once
a terminal state is reached,
the environment should be (re)initialized by
    s = env.reset()
where s is the initial state.
An experience (sample) can be collected from s by taking an
action a as follows:
    s_, r, terminal, info = env.step(a)
where s_ is the resulted state by taking the action a,
      r is the reward achieved by taking the action a,
      terminal is a boolean flag to indicate if s_ is a terminal state, and
      info is just used to keep compatible with openAI gym library.


A Logger instance is provided for each function, through which you can
visualize the process of the algorithm.
You can visualize the value, v, and policy, pi, for the i-th iteration by
    logger.log(i, v, pi)
```

```
You can also only update the visualization of the v values by
    logger.log(i, v)
"""


# use random library if needed
import random

import numpy as np


def value_iteration(env, gamma, max_iterations, logger):
    """
    Implement value iteration to return a deterministic policy for all states.
    See lines 20-30 for details.

    Parameters
    ----------
    env: GridWorld
        the environment
    gamma: float
        the reward discount factor
    max_iterations: integer
        the maximum number of value iterations that should be performed;
        the algorithm should terminate when max_iterations is exceeded.
        Hint: The value iteration may converge before reaching max_iterations.
        In this case, you want to exit the algorithm earlier. A way to check
        if value iteration has already converged is to check whether
```

```
    the infinity norm between the values before and after
    an iteration is small enough.
    In the gridworld environments, 1e-4 (theta parameter in the pseudocode)
    is an acceptable tolerance.
logger: app.grid_world.App.Logger
    a logger instance to perform test and record the iteration process

Returns
-------
pi: list or dict
    pi[s] should give a valid action,
    i.e. an integer in [0, env.action_space.n),
    as the optimal policy found by the algorithm for the state s.
"""
NUM_STATES = env.observation_space.n
NUM_ACTIONS = env.action_space.n
TRANSITION_MODEL = env.trans_model

v = [0] * NUM_STATES
pi = [0] * NUM_STATES
# Visualize the value and policy
logger.log(0, v, pi)
# At each iteration, you may need to keep track of pi to perform logging

# Initialize tolerance for convergence
theta = 1e-4

# Function to calculate q values for a state
```

```
def helper(state, v):
    sum_states = [0] * NUM_ACTIONS
    for action in range(NUM_ACTIONS):
        for p, s_, r, terminal in TRANSITION_MODEL[state][action]:
            sum_states[action] += p * (r + gamma * v[s_]*(not terminal))
    return sum_states


for k in range(max_iterations):
    # Copy v_k-1 values into v1
    v1 = v.copy()
    for state in range(NUM_STATES):
        # Update v values based on max q values
        q = helper(state, v1)
        # Update policy based on argmax q values
        v[state] = max(q)
        pi[state] = np.argmax(q)
    # Convert to np array in order to compute norm
    v1_np = np.array(v1)
    v_np = np.array(v)
    # Convergence check
    if np.linalg.norm(v_np - v1_np, np.inf) < theta:
        break
    # Log the current iteration
    logger.log(k+1, v, pi)

return pi
```

Appendix B A2C Pong Code

The following code was used to produce the results from section 4.1.1.

```
!pip install stable-baselines3[extra]
import os
from stable_baselines3 import A2C
from stable_baselines3.common.env_util import make_atari_env
from stable_baselines3.common.vec_env import VecFrameStack
# There already exists an environment generator that will make
and wrap atari environments correctly.
env = make_atari_env('PongNoFrameskip-v4', n_envs=16, seed=0)
# Stack 4 frames
env = VecFrameStack(env, n_stack=4)

# Hyperparameters from SB3
model = A2C('CnnPolicy', env, verbose=1, ent_coef = .01, vf_coef = .25)
model.learn(total_timesteps=1e7)

from google.colab import files
model.save("a2c_pong_trained_2")
files.download("a2c_pong_trained_2.zip")

# From SB3
# Record a video
from stable_baselines3.common.vec_env import VecVideoRecorder,
DummyVecEnv
env = make_atari_env('PongNoFrameskip-v4', n_envs=4, seed=0)
env = VecFrameStack(env, n_stack=4)
```

```
env_id = 'PongNoFrameskip-v4'
video_folder = 'logs/videos/'
video_length = 3000

obs = env.reset()

# Record the video starting at the first step
env = VecVideoRecorder(env, video_folder,
                       record_video_trigger=lambda x: x == 0,
                       video_length=video_length,
                       name_prefix=" 1e7 Pong Trained Agent 4 ENVS
                       2-{}".format(env_id))

obs = env.reset()
for _ in range(video_length + 1):
  action, _ = model.predict(obs)
  obs, _, _, _ = env.step(action)
# Save the video
env.close()

# Evaluate trained agent
from stable_baselines3.common.evaluation import evaluate_policy
mean_reward, std_reward = evaluate_policy(trained_model, env)

print(f"Mean reward = {mean_reward:.2f} +/- {std_reward:.2f}")

# Plot results of training
import matplotlib.pyplot as plt
```

```
timesteps = [(2*i) * 1e5 for i in range(51)]
rewards = [-20.7, -20.6, -20.4, -20.5, -19.9, -18.7, -15.7, -11.9,
           -6.26, -4.29, -4.75, -1.04, 14.6, 18.5, 17.9, 19.3,
           16.5, 13.7, 19.4, 19.3, 12.6, 19.3, 19.2, 19.3, 17.2,
           18.4, 18.9, 13.9, 18.8, 19.2, 9.03, 19.6,
          19.4 ,19.3, 14.4, 19.6, 19.2, 18.9, 19.1, 19.4,
          17.1, 13.7, 17.1, 9.09, 19.2, 5.5, 15.4, 19.1, 19.5,
          19.6, 19.3]

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
plt.plot(timesteps, rewards, linewidth=2)
plt.ylabel('Reward')
plt.xlabel('Episode #')
ax1.set_title('Avg Training Reward')
plt.show()
```

Appendix C A2C HalfCheetah Code

The following code was used to produce the results from section 4.2.2.

```
!pip install stable-baselines3[extra] pybullet

import os

import pybullet_envs

from stable_baselines3 import A2C
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.vec_env import VecNormalize
# From SB3 documentation
def func(progress):
        """
        Progress will decrease from 1 (beginning) to 0
        :param progress: (float)
        :return: (float)
        """
        return progress * .002
env = make_vec_env("HalfCheetahBulletEnv-v0", n_envs=4)

env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.)

model = A2C('MlpPolicy', env, learning_rate = func, n_steps =32, gamma = 0.99,
vf_coef = 0.5,ent_coef= 0.0, verbose=1 )
model.learn(total_timesteps=2e6)
```

```
from stable_baselines3.common.evaluation import evaluate_policy
env = make_vec_env("HalfCheetahBulletEnv-v0", n_envs=1)
env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.)
#  do not update them at test time
env.training = False
# reward normalization is not needed at test time
env.norm_reward = False
mean_reward, std_reward = evaluate_policy(model, env)

print(f"Mean reward = {mean_reward:.2f} +/- {std_reward:.2f}")

rom stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv

env_id = 'HalfCheetahBulletEnv-v2'
video_folder = 'logs/videos/'
video_length = 3000

#env = DummyVecEnv([lambda: gym.make(env_id)])

obs = env.reset()
# From SB3
# Record the video starting at the first step
env = VecVideoRecorder(env, video_folder,
                       record_video_trigger=lambda x: x == 0,
                       video_length=video_length,
                       name_prefix=
                       " 2e6 CORRECT HYPERPARAMETERS Trained Agent 1
                       ENVS-{}".format(env_id))
```

```
obs = env.reset()
for _ in range(video_length + 1):
  action, _ = model.predict(obs)
  obs, _, _, _ = env.step(action)
# Save the video
env.close()

# Plot results of training
import matplotlib.pyplot as plt

timesteps = [0, 25600, 38400, 51200, 64000, 76800,
             89600, 102400, 115200, 128000,
              140800, 153600, 166400, 179200, 192000,
              204800, 217600, 243200, 268800, 307200,
              345600, 384000, 422400, 486400, 524800,
              563200, 614400, 665600, 729600,
              780800, 883200, 934400, 1011200, 1100800,
              1203200, 1267200, 1382400,
              1420800, 1536000, 1600000, 1728000, 1766400,
              1868800, 1907200, 1996800]
rewards = [-1.23e3, -765,-522,-283,-110,-26.6, 35, 69.6, 209, 323, 355, 362,
           367, 376, 388, 400, 469, 489, 484, 516, 554, 596, 714, 903, 1010,
           1100, 1190, 1250, 1340, 1420, 1590, 1680, 1790, 1750, 1830, 1750,
           1650, 1700, 1820, 1930, 1990, 2000, 2010, 2000, 2000]

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
```

```
plt.plot(timesteps, rewards, linewidth=2)
plt.ylabel('Reward')
plt.xlabel('Timesteps')
ax1.set_title('Avg Training Reward')
plt.show()
```

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Shan Zhong Hongyang Yang, Xiao-Yang Liu and Anwar Walid. Deep reinforcement learning for automated stock trading: An ensemble strategy. *ICAIF*, 2020.

[3] Ioannis Karamouzas. Derivative free methods. `https://sites.google.com/a/g.clemson.edu/cpsc-drl/schedule`, 2021.

[4] Ioannis Karamouzas. Markov decision process. `https://sites.google.com/a/g.clemson.edu/cpsc-drl/schedule`, 2021.

[5] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`, 2016.

[6] Fei Fei Li and Andrej Karpathy. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.stanford.edu/`, 2021.

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[8] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[9] Daniel Seita. Frame skipping and pre-processing for deep q-networks on atari 2600 games. *danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/*, 2016.

[10] David Silver. Lecture 2: Markov decisioun process. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[11] David Silver. Lecture 7: Policy gradient methods. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[12] Barto A. Sutton, R. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[13] Lilian Weng. Policy gradient algorithms. *lilianweng.github.io/lil-log*, 2018.

[14] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. machine learning. 1992.

[15] Pei Xu and Ionnis Karamouzas. Clemson university cpsc 8810 project 1: Gridworld, 2020.