Reinforcement Learning - Algorithms and Environments

Project Report

Eklavya Mentorship Program

Author: *Viraj Shah*

Mentors: *Himanshu Chougule, Aniruddha Thakre*
from the Society of Robotics and Automation, VJTI, Mumbai

Date: October 14, 2022

Contents

1 Project Overview

Reinforcement learning proves to be a novel approach to solving an increasing amount of increasingly complex problems, while simultaneously having a strong mathematical foundation whose first stones were laid as early as the 1960s. Modern RL agents are capable of achieving superhuman performance in environments as complex as the popular MMORPG title StarCraft II. To explore this subject, we have solved, as well as created, various RL environments.

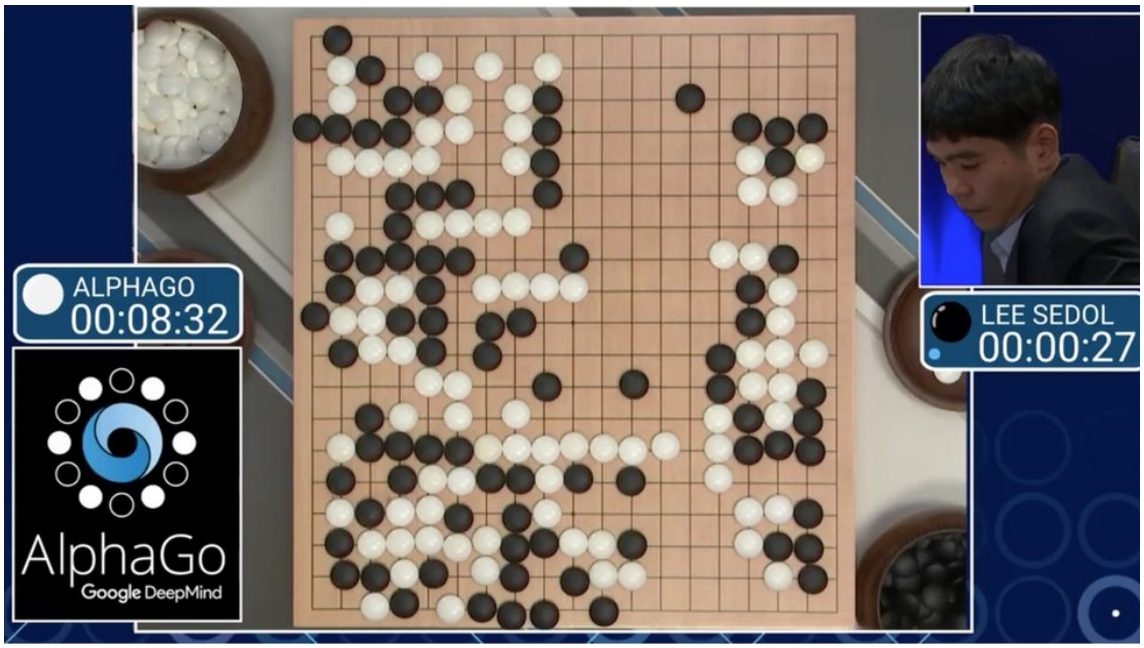

Figure 1: One of the more recent triumphs of Reinforcement Learning - Google DeepMind's AlphaGo defeats the Go world champion.

2 Applications

2.1 Overview

Due to its generality, reinforcement learning is studied in many disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, and statistics. In the operations research and control literature, reinforcement learning is called approximate dynamic programming, or neuro-dynamic programming. The problems of interest in reinforcement learning have also been studied in the theory of optimal control, which is concerned mostly with the existence and characterization of optimal solutions, and algorithms for their exact computation, and less with learning or approximation, particularly in the absence of a mathematical model of the environment.

2.2 Specific Applications

The use of deep learning and reinforcement learning can train robots that have the ability to grasp various objects-even those unseen during training. This can, for example, be used in building products in an assembly line. This is achieved by combining large-scale distributed optimization and a variant of deep Q-Learning called QT-Opt. QT-Opt support for continuous action spaces makes it suitable for robotics problems. A model is first trained offline and then deployed and fine-tuned on the real robot. Google AI applied this approach to robotics grasping where 7 real-world robots ran for 800 robot hours in a 4-month period.

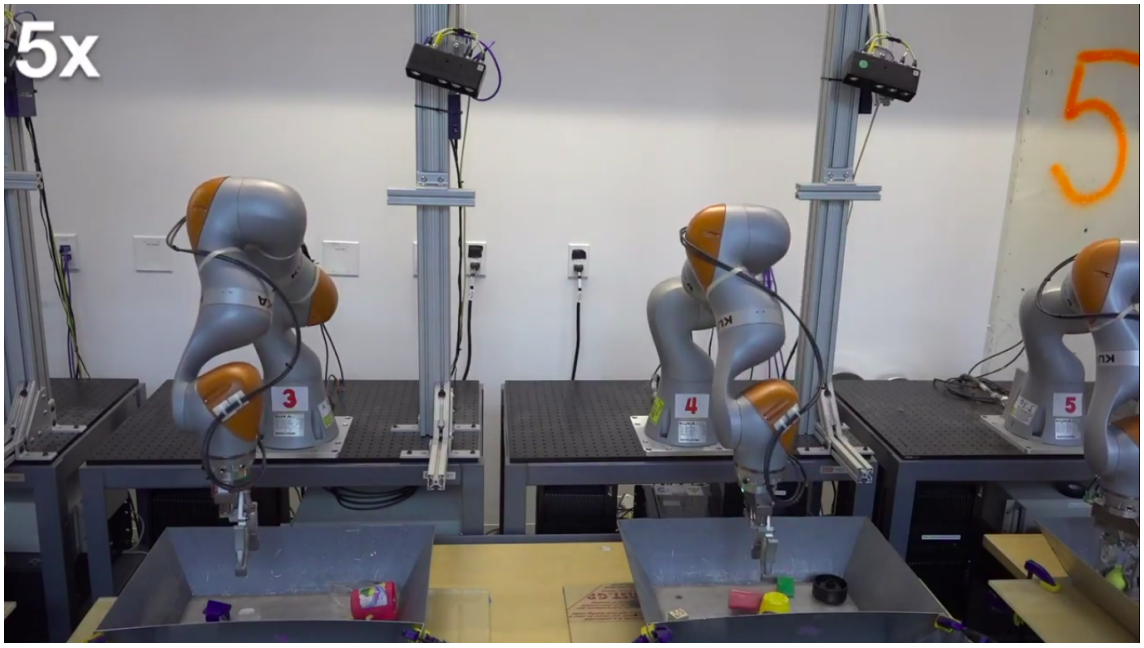

Figure 2: Google AI's RL based pick and place robots.

In this experiment, the QT-Opt approach succeeds in 96% of the grasp attempts across 700 trials grasps on objects that were previously unseen. Google AI's previous method had a 78

Various papers have proposed Deep Reinforcement Learning for autonomous driving. In self-driving cars, there are various aspects to consider, such as speed limits at various places, drivable zones, avoiding collisions-just to mention a few. AWS DeepRacer is an autonomous racing car that has been designed to test out RL in a physical track. It uses cameras to visualize the runway and a reinforcement learning model to control the throttle and direction.

In the industry, learning-based robots are used to perform various tasks. Apart from the fact that these robots are more efficient than human beings, they can also perform tasks that would be dangerous for people. A great example is the use of AI agents by Deepmind to cool Google Data Centers. This led to a 40% reduction in energy spending. The centers are now fully controlled with the AI system without the need for human intervention. There is obviously still supervision from data center experts.

3 Introduction

Reinforcement learning (RL) can be described as a computational method to approach learning from interaction. RL is a goal-directed, interaction driven approach to learning and automated decision making. It separates itself from different techniques by using direct interaction with the environment as a source of information.

3.1 The Reinforcement Learning Problem

Reinforcement learning is different from other machine learning problems, in the way that it focuses on learning, not through evaluating the end result obtained by an agent (or model in case of other subcategories of machine learning), but rather by taking into consideration whatever the agent observes, the actions it takes, and their effects on the environment.

3.2 Towards a Solution

The idea is to come up with a way to find a mapping between situations and actions that maximises a numerical reward signal over a complete episode. This involves a certain level of planning - the agent must learn to take take actions which may not provide the best immediate reward, but are likely to pay off over the rest of the episode.

An important consideration is the balance between exploration and exploitation, that is the split between the set of actions that are known to be rewarding, versus the set of actions that may be even more rewarding, but have not been well explored by our agent.

3.3 The Structure of the RL Problem

The interactions between the agent and the environment can be characterised as a loop, with each iteration corresponding to an exchange of information between them, resulting in a state transition where this can repeat.

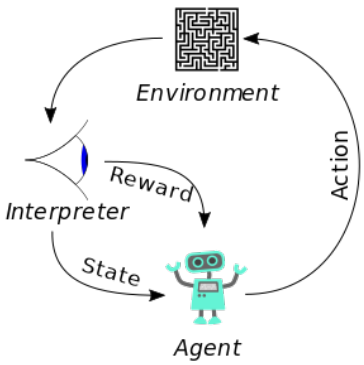

Figure 3: The typical framing of a RL scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent.

Apart from the agent being trained and the environment it is being trained to deal with, reinforcement learning systems typically are composed of four major elements:

1. The policy followed by the agent.
2. A reward function determined by the specifics of the problem.
3. A value function learned by the agent.
4. A model of the environment, also learned by the agent.

3.4 Implementing Environments and their Solutions

In order to practically implement and manage the many RL algorithms in a uniform manner, it is only natural that we require all kinds of environments and agents share a uniform, standardized interface such that the internal workings of the environment are abstracted away and agents, regardless of their policies can be made to interact with any environment with minimal adjustment (whether the agent will perform well or not perform at all is a separate consideration). To achieve this level of standardization, agents are implemented as to interact with environments through the OpenAI Gym API, and environments are written as to conform to its specifications.

4 A Deeper Dive

4.1 k-Armed Bandits

A good way to be introduced to some of the key aspects of RL is through a simplified version of the RL problem - one where there is a single state, which means that the environment has no dynamics with regards to how it reacts to the actions taken by the agent, and rewards are simply sampled from a stationary distribution based on the action taken.

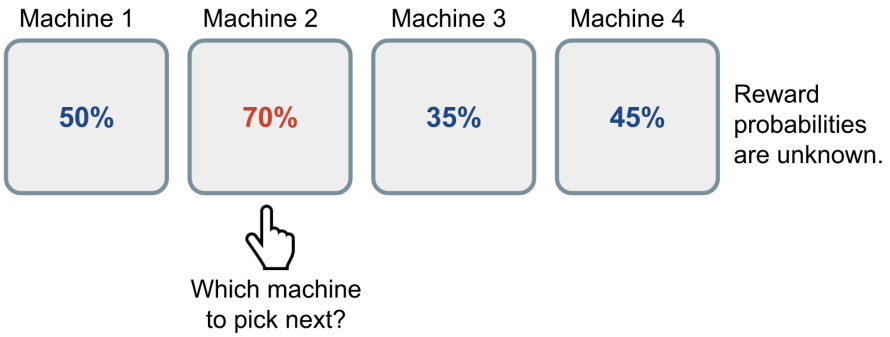

Figure 4: A representation of the k-Armed bandit problem: The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits"), who has to decide which machines to play, how many times to play each machine and in which order to play them, and whether to continue with the current machine or try a different machine.

The subset of the RL problem as just described is commonly known as the k-Armed bandits program and is where we chose to begin work on this project. As above, the k-armed bandit

problem allows the agent to choose one of k available actions, to each of which can be attributed a stationary probability distribution determining the reward signal upon taking that action. The agent chooses any of the given actions for each time step, in attempt to maximise the total reward. In this problem, the value function can easily be learnt as the expected reward for any given action, by setting it to be the mean reward obtained by the agent by taking the action.

The problem then becomes one of simply finding an optimal strategy to balance exploration and exploitation, and also happens to be free from a lot of the other complications present in more whole RL problems, making it ideal for testing several strategies to tackle this:

1. The greedy policy - always exploits the best available as per the agent's experience.
2. The ϵ-greedy policy - explores with probability ϵ, otherwise takes the greedy action.
3. The optimistic greedy policy - acts greedily but starts out with high predicted action-values.
4. Upper confidence bounds - acts greedily according to a preference value that considers both the expected reward and its uncertainty.
5. Gradient bandits - uses a parameterized policy and applies gradient descent to update parameters.
6. Bayesian bandits - uses Bayesian inference to update the estimated distributions of each action.

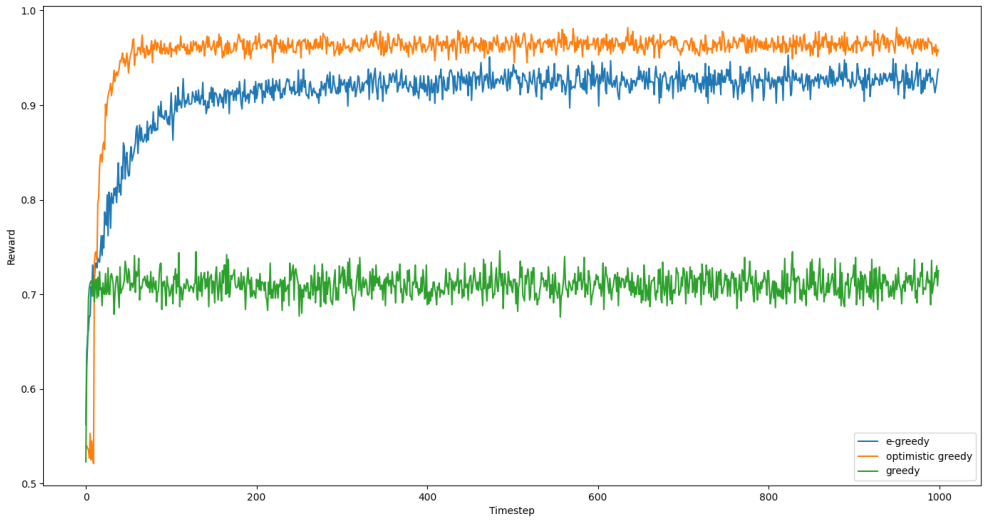

Figure 5: A plot comparing the learning rates of some different policies.

To summarize our findings: there are several simple ways of balancing exploration and exploitation. The ϵ-greedy methods choose randomly a small fraction of the time, whereas UCB

methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing estimates optimistically causes even greedy methods to explore significantly.

4.2 Dynamic Programming

Now armed with solutions to the exploration vs. exploitation problem, we continue to expand our horizons beyond the single state case. With multiple states, we must use a robust mathematics framework to define how our agent interacts with the environment and what those interactions result in. This mathematical framework comes in the form of a Markov Decision Process (MDP).

MDPs describe the state space, action space, reward function, transition kernel, and discount factor and tell us all we need to know about the environment to figure out the optimal policy without having to actually have the agent sample any uncertainties through training. The Bellman equations relate the various components of an MDP.

The Bellman expectation equations are:

$$v_\pi(s) = \sum_a \pi(s,a)[r(s,a) + \gamma \sum_{s'} p(s'|a,s) v_\pi(s')] \tag{1}$$

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|a,s) \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \tag{2}$$

The Bellman optimality equations are:

$$v^*(s) = \max_a [r(s,a) + \gamma \sum_{s'} p(s'|a,s) v^*(s')] \tag{3}$$

$$q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|a,s) \max_{a' \in A} q^*(s',a') \tag{4}$$

The set of techniques that uses information provided by MDPs to solve the problems of prediction and control are all bundled into Dynamic Programming (DP). These use the Bellman equations as a rule to update the learnt state-values until the optimal state-value function is learnt in policy evaluation, and creates a new policy that acts greedily upon these in policy improvement, which in turn can be repeatedly done each time the value function converges in policy iteration, or after a single iteration of policy iteration in value iteration.

Classical DP methods operate in sweeps through the state set, performing an expected update operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements.

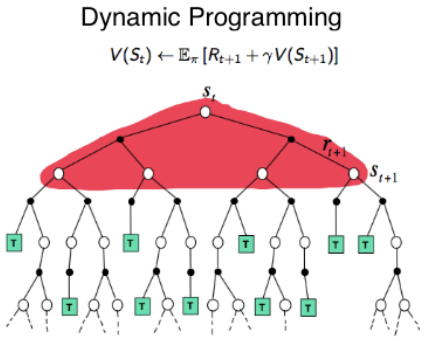

Figure 6: A tree showing possible state sequences highlighting states considered while using DP.

To elaborate on these techniques one at a time:

1. Policy evaluation refers to the (typically) iterative computation of the value function for a given policy.

2. Policy improvement refers to the computation of an improved policy given the value function for that policy.

3. Putting these two computations together, we obtain policy iteration and value iteration, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

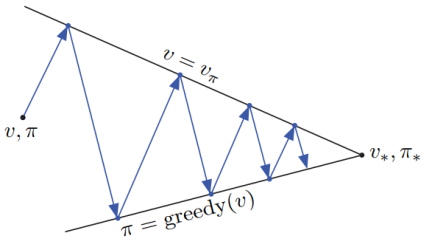

Figure 7: A diagram showing how policy iteration causes the state-value function to converge.

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. In the worst case, the time that DP methods take to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k. A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n.

DP is sometimes thought to be of limited applicability because of the curse of dimensionality, the fact that the number of states often grows exponentially with the number of state variables. On problems with large state spaces, asynchronous DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state.

4.3 Monte Carlo Methods

DP methods fail to work completely when sufficient information about the relevant MDP is not available. In some cases, even when we have a perfect model of the environment, the state-space is too big to efficiently sweep over the entirety of. To work with such problems, we use Monte Carlo (MC) methods. Here we do not assume complete knowledge of the environment. MC methods require only experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from actual experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior.

Monte Carlo methods sample and average returns for each state–action pair much like the bandit methods we explored sample and average rewards for each action. The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode.

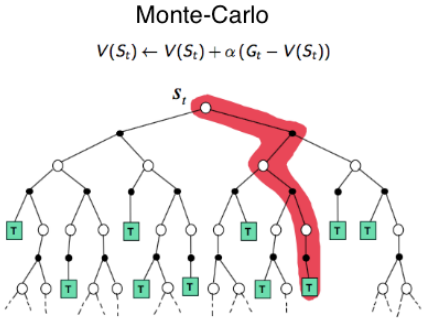

Figure 8: A tree showing possible state sequences highlighting states considered while using MC.

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of sample episodes. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or sample models. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to focus Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set

4.4 Temporal Difference Methods

Temporal Difference (TD) learning is a combination of Monte Carlo ideas and Dynamic Programming ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome.

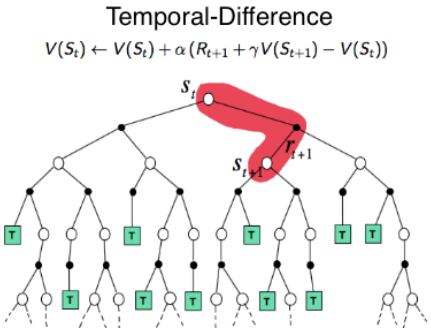

Figure 9: A tree showing possible state sequences highlighting states considered while using TD.

Carefully combining the update rules given by both Monte Carlo methods and Dynamic Programming allows us to reap the benefits of both while also managing to avoid their respective weaknesses. In this project, we focus on one major TD method: Q learning.

Q learning involves creation of a Q table which stores state-action values for all possible combinations. In a Monte Carlo like fashion, the agent is made to sample values and update the table accordingly. However, instead of waiting for the episode to be completed before integrating its new experiences into its decision process, it takes a page from dynamic programming's book and uses bootstrapping to turn it into an on-policy technique. The hybrid update is given by:

$$q(s,a) \longleftarrow q(s,a) + \alpha[r + \gamma \max_a q(s',a) - q(s,a)] \tag{5}$$

Approaches like Q learning are widely applicable, but invariably struggle when the state-action space become so large that it is no longer feasible to represent and update the learnt Q function tabularly, which forces us to replace our Q function with an approximation, such as a simple linear approximator or more effectively, a neural network.

Such uses of neural networks in RL opens up a whole new field - Deep Reinforcement Learning.

4.5 Deep Reinforcement Learning

The simplest Deep RL algorithms would be Deep Q Learning - also called Deep Q Networks (DQN). As mentioned above, DQN uses a neural network to approximate the Q function, but in order to facilitate the use of Deep Learning in this form, various tweaks and improvements are often made in attempt to sidestep common pitfalls, such as overfitting due to recency bias, catastrophic forgetfulness, and the learning of undesirable cues from correlations in the observation sequence.

To address these, DQN is often paired with experience replay, where state transitions are stored in a memory buffer, from which batches of random experiences are sampled and shown to the agent. Another common improvement is using a second neural network to evaluate the best action while the other trains. Every set number of iterations, the learnt information is copied over this network. In a variant known as double DQN, the two networks are trained in a mutually symmetric fashion using separate experiences with update rules that incorporate information from the other network, leading to accelerated learning.

In addition to these, many improvements have been made to Deep Q Learning, and it is an active topic of research. Rainbow DQN combines the classic DQN algorithms with six key improvements and can be considered an optimal starting point for further work in this field.

While working on this project, most problems faced were closely related to the training time vs. efficiency of various configurations of implementational details and hyperparameters. Both the learning rate and performance (and not to mention whether the agent was able to learn at all) proved to be highly sensitive to the aforementioned. More importantly, the amount of time that would have to be spent in training before any conclusions could be made about the effectiveness of any changes was a major hinderance.

4.6 Custom Environments

As mentioned in the introductory section, environments should be written in a standardized fashion - having the same interface for the agent to interact with, and being easy to integrate with existing code, for any kind of agents or wrappers.

Aside from that implementational detail, it is important to consider what kind of information is being communicated between it and the environment while designing one. Such considerations boil down to fixing the boundary between the agent and the environment, which suggests what actions at what levels should be possible and what kind of observations would those be based on. For example, consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out—say, where the rubber meets the road, considering your actions to be tire torques. Or you could define them farther in—say, where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of where to drive. What is the right level, the right place to draw the line between agent and environment?

5 Implementations

All implementations are done in Python 3.10 and use NumPy to handle data. Environments from OpenAI Gym have also been used, and custom environments have been written to conform to the same's API.

5.1 Project Structure

All scripts are stored in src/, with them using saves/ to read and write learnt values or weights. Custom environments live in envs/.

```
README.md
envs
    gridworldenv.py
    pickplaceenv.py
    snakeenv.py
notes
    introduction.md
    multi-armed-bandits.md
requirements.txt
saves
    blackjack_q.npy
    cartpole_q.npy
    mountaincar_q.npy
    pendulum_q.npy
src
    blackjack.ipynb
    control.ipynb
    dqn.py
    gridworld.ipynb
    multiarmbandits.py
    preprocess.py
```

5.2 multiarmedbandits.py

This script is designed to test out the exporation vs. exploitation balance of the various strategies as previously discussed. It contains functions that set-up a basic k-Armed Bandits testbench and functions to measure learning rates, along with measures of how well the learnt distribution matches the actual distribution ie a measure of exploration and how much reward the agent actually ended up receiving ie exploitation

5.3 gridworld.ipynb

This notebook contains an implementation of the gridworld environment, applies Dynamic Programming techniques to solve the environment, and analyzes the results.

5.4 blackjack.ipynb

This notebook solves the blackjack environment provided by OpenAI Gym, using Monte Carlo methods, and analyzes the results.

5.5 control.ipynb

This notebook solves three of the Classic Control environments provided by OpenAI Gym. It uses Q learning, and a single solution adapts effectively to the variation in action and observation spaces.

5.6 dqn.py and preprocess.py

preprocess.py implements the Wrapper interface provided by OpenAI Gym to preprocess the observations passed by the Atari environments. It resizes frames, converts them to grayscale, and stacks them to create the new observation which doubles as the state of an agent intending to solve any of those environments.

dqn.py implements Delayed Deep Q Learning with Experience Replay to solve the Atari 2600 Pong environment, using preprocess.py's observation wrapper. It includes functionality to track training progress and create checkpoints from which training can be paused and resumed.

5.7 gridworldenv.py, pickplaceenv.py, and snakeenv.py

These each implement an environment that adheres to the OpenAI Gym API, and use pygame to create visuals.

1. gridworldenv.py implements a Grid World like environment where the agent it to travel to, and open a chest for which a key, present somewhere else on the grid, is required.
2. pickplaceenv.py implements an 2D environment with a manipulator that is pick up an object from one location and place it at another.
3. snakeenv.py implements an environment representing the classic game - Snake.

6 Conclusion and Future Work

6.1 Conclusion

Reinforcement Learning problems have many possible solution methods, each more suitable for a given type of problem, with more state of the art methods generally coming ever so close to effectively solving the entire reinforcement learning problem with its broadest definition. This project is a journey towards such a method, with each technique bringing new improvements over the last.

6.2 Future Work

The next steps would be to continue working towards current bleeding edge research work, with a goal to eventually even contribute to the field of reinforcement learning.

References

1. Introduction to Linear Algebra (Gilbert Strang)
2. Reinforcement Learning: An Introduction
3. Algorithms for Reinforcement Learning
4. DeepMind x UCL Deep Learning Lecture Series
5. Foundations of Deep RL (Pieter Abbeel)
6. OpenAI Gym Documentation