Intro to the Algorithms of Reinforcement Learning

Harrison Jansma

Abstract

This document details an overview of the field of reinforcement learning as well as my personal implementation of policy evaluation, iterative policy improvement, and Q-Learning.

1 Introduction

Reinforcement Learning is a subfield of machine learning that has received increasing attention over the last five years. This is largely due to the impressive feats that Reinforcement Learning methods have been able to achieve. Research institutes like OpenAI and DeepMind have been able to use reinforcement learning to create things like robotic hands that can manipulate objects (OpenAI 2018) or computer-controlled teams that can play the 5v5 competitive video game DOTA at the human-level. (OpenAI 2018)

Beyond these astounding accomplishments, expectations of future technologies like self-driving cars are set to establish reinforcement learning techniques as the foundation of a new wave of technological advancement.

In this paper, I seek to provide an overview of the fundamental reinforcement learning concepts that permeate the literature. Section 1 through Section 6 will be devoted to the understanding of the agent-environment interaction, Markov Decision Processes, and Policy Selection. Sections 7 through 10 will be devoted to the implementation of various reinforcement learning algorithms towards simple tasks.

2 Reinforcement Learning

The fundamental problem that reinforcement learning seeks to address is "How do we learn to interact with an environment to accomplish a goal?" Reinforcement learning solutions break this problem down into an interactive relationship between two entities, an environment and an agent.

An **agent** can sense some state of its environment and perform actions that affect said environment. The agent must also have a goal that it can achieve by means of its actions. How the agent goes comes to understand the affect of its actions on its goal is the process of learning. Often the agent must learn to accomplish its goal despite a limited perception of its environment and complex relations between the agent's actions and its overall goal. To solve these complicated tasks, agents must learn complicated planning skills by means of predicting the consequences of actions on the goal in question.

The agent's task of learning to maximize its goal by interaction raises many problems not faced in other fields of machine learning. One of these challenges is the tradeoff between **exploration and exploitation**. To accomplish a goal, the agent must perform actions that it knows are the most beneficial, but to discover said actions the agent must test new actions that have not been selected before. Thus, the agent's pursuit of its goal can become a hinderance to its own learning.

One other challenge faced by the agent is the difficulty in learning from an complex environment. The task of planning actions that generate future reward requires a complete understanding of past experiences. If the agent isn't capable of generalizing from past experiences, then the agent must resort to trial and error for every possible action in every state of its environment.

3 Markov Decision Processes

Reinforcement learning borrows from control theory and dynamic programming in the uses of **Markov Decision Processes** (MDPs). MDPs are used in the context of reinforcement learning to

formalize the task of sequential decision making under the assumption of delayed rewards. Without this assumption, an agent's actions are independent, implying that in any state the action that achieves the highest immediate reward should be chosen by the agent. This is called the **Multi-armed bandit** problem, and though it is a subset of reinforcement learning it is not useful towards game-playing and thus is out of the scope of this paper.

MDPs frame the sequential decision-making process of an **agent** with respect to everything outside its control, namely the **environment**. The delayed reward component MDPs allow a present action to affect the agent's environment and therefore affect rewards for future actions. These **rewards** are numerical values that the agent seeks to maximize over time. The entire state-agent interaction is described by the following pattern. Given a perceived **state** of the environment, the agent chooses an action that maximizes expected future rewards. The environment changes as a result of this action and returns to the agent a reward signal and a new state. This interaction is cyclical and reoccurs either indefinitely or until a specific terminal state is reached.

Markov decision processes are defined as a set of states, actions, and rewards with a function that determines next state and reward given the current state and action. More explicitly, the random variables for reward and state **R_t and S_t** have well defined probability distributions dependent on the previous state and action.

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}.$$

Sutton and Barto

From the deterministic function p, we can derive models of how the environment changes with respect to an agent's actions (state-transition probabilities) or even expectations of reward given state and action.

4 Rewards and Returns

The goal of the agent is encoded into a numeric signal (reward) that is passed from the environment to the agent as a result of an action. Given a sequence of rewards, $R_t, R_{t+1}, R_{t+2}, \ldots$, the agent seeks to maximize the **expected return**, or some cumulative sum of future rewards. Most commonly this is expressed as a discounted sum of future rewards G_t given below.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

Sutton and Barlow

The **discounting factor** γ $(0 \leq \gamma \leq 1)$, ensures that the expected return is finite, even for agent-environment interactions of indefinite length.

5 Policies and Value Functions

I have defined a system of interaction between the agent and environment, as well as an expected reward function that the agent seeks to maximize. Now we must consider how an agent can learn goal maximizing decisions from a sequence of states, actions, and rewards.

When faced with a decision (choice of multiple actions given a state), the agent must estimate an expected reward for each choice of action. This **value function** estimates how good it is for an agent to be in each state. This value is determined by the rewards (and actions) of all future timesteps starting at that particular state. Since future rewards depend on the agent's future choices, the value function is defined with respect to a **policy**.

A policy is simply a mapping from states to actions. This decision-making function can be stochastic (given state, choices are partially random) or deterministic (given state, choice is fixed). Either way, having a policy allows the value function to be calculated by ensuring that the agent will act in a predictable manner. With a policy in hand, and a complete model of the environment, the agent's maximization of reward is a guaranteed task that can be completed via dynamic programming.

6 Optimal Policies and Planning

Since value functions are defined with respect to a policy, they can also be used to determine which policies will maximize the reward for the agent. These policies are called **optimal policies.**

The optimal policy for an environment has a few defining features. Firstly, the value function attained by following the optimal policy expects higher cumulative future returns than any other policy would. More simply, an agent acting in accordance with the optimal policy will choose the most beneficial action in every state it finds itself in.

The task of finding the optimal policy is the primary goal of reinforcement learning. For a known environment this is called **planning** and it is achieved through algorithms within the field of dynamic programming. In situations where a model of the environment is not available, methods from control theory and machine learning are deployed to learn from an agent's trial and error.

7 Policy Evaluation

Policy evaluation is the task of determining how beneficial following a policy will be. In practice, this amounts to approximating the value function $V_\pi(s)$ for every state s. With a known environment, this can be achieved via dynamic programming.

In this project I implemented one such algorithm of policy evaluation detailed in **Sutton and Barlow.**

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Sutton and Barto

The algorithm starts by initializing a random value for each state. Then we iterate through each state, getting the probabilities of next action determined by the policy and subsequently updating the value function for the current state to the expected return of all future states. By iterating many times, the algorithm effectively "backs-up" rewards from future states to earlier states, converging to the true value function of the policy.

The environment that this algorithm was applied to was the gridworld problem. In this simple task the agent is placed on a checkerboard and tasked with moving to one corner of the board as quickly as possible. It is a rather uninteresting task but was chosen because of this simplicity.

Code and notes can be found in the project directory, but the results are summarized here. In the code, a random policy was evaluated on a 4x4 grid. The policy chose up, left, right, or down with an equal probability of 0.25. In turn, the agent received a reward of -1 for all positions except two corners on the grid where the agent "stops playing".

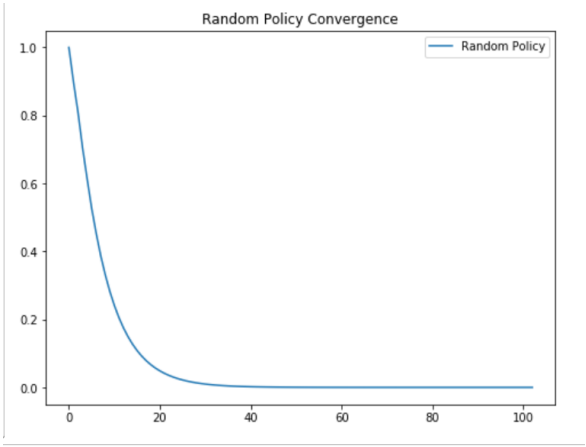
The grids below shows the estimated value function results for each state of the 4x4 grid. At the first iteration we start with zero values and end with values of -1 everywhere except the poles. After the second iteration, we start to see the absence of reward at the corners is being "backed-up" into the values of inner squares. The final grid shows the converged results of the random value function. Notice that the expected rewards is smaller as the squares get closer to the corner. This is caused by the agent being more likely to move to the corner given a random policy.

```
Iteration 1:
Delta: 1.0
[[ 0. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1.  0.]]

Iteration 2:
Delta: 0.9
[[ 0.    -1.675 -1.9   -1.9  ]
 [-1.675 -1.9   -1.9   -1.9  ]
 [-1.9   -1.9   -1.9   -1.675]
 [-1.9   -1.9   -1.675  0.   ]]

Iteration 103:
Delta: 0.0
[[ 0.   -5.28 -7.13 -7.65]
 [-5.28 -6.61 -7.18 -7.13]
 [-7.13 -7.18 -6.61 -5.28]
 [-7.65 -7.13 -5.28  0.  ]]
```

As the program iterates, the estimated value of the value function of the random policy converges to the true value function. This convergence is demonstrated in the following figure.



8 Policy Improvement

Policy evaluation allows us to estimate the policy's value function. This in turn tells us how

much reward the agent can expect to accumulate by being in a state s and thereafter following the given policy.

With the estimated value function computed we can determine whether following a different policy would be more beneficial to the agent. Given two policies $\pi \; and \; \pi'$, a value function $V_\pi(s)$, and model of the environment $p(s'|s,a)$ in any state we should switch to the new policy π' if the following.

$$q_\pi\big(s,\pi'(s)\big) \geq V_\pi(s)$$

The above equation means that given a current state s, it is beneficial to switch policies to π' if the expected return for taking a single action under policy π' and therafter reverting to π is greater than the expected return from always following the policy π. If in every state it is better to switch to policy π', then we can deduce that π' is at least as good if not better that π.

When applied to the above gridworld example, we can compare the random policy to a policy that chooses actions greedily based on $V_{random}(s)$. Checking at each position on the board whether it is beneficial to switch to the greedy policy.

```
Is greedy selection better
in the current state?

[True, True, True, True]
[True, True, True, True]
[True, True, True, True]
[True, True, True, True]
```

In fact it can be proven that it is always better to switch to a greedy policy based on the value function of the current policy. This implies that with a known environment, we can find the optimal policy by starting with a random policy and iteratively evaluating the policy then updating the policy by greedily selecting from the evaluated value function. This algorithm is called **policy improvement**.

In the gridworld example, an optimal policy was found by starting with the random policy value function we previously approximated, then iteratively creating a new greedy policy and evaluating. The resulting optimal value function and policy after one thousand iterations are as follows.

```
Optimal Value Function:
[[ 0.   -1.   -1.9  -2.71]
 [-1.   -1.9  -2.71 -1.9 ]
 [-1.9  -2.71 -1.9  -1.  ]
 [-2.71 -1.9  -1.    0.  ]]

 Optimal Policy:
 ['UDRL', 'L', 'L', 'DL']
 ['U', 'UL', 'UDRL', 'D']
 ['U', 'UDRL', 'DR', 'D']
 ['UR', 'R', 'R', 'UDRL']
```

Policy iteration and improvement are exhaustive searches of environment states. They also require a complete model of the world, which can be an unreasonable assumption in a unknowable or dynamic environment.

9 Q - Learning

Policy iteration allowed us to use a model of the environment to find better and better policies for the agent. This convergence to an optimal policy is fueled by the idea of iteratively "backing-up" information about rewards from future time-steps to current estimates of expected rewards.

Back-ups are applied in many reinforcement learning algorithms. One such algorithm is called **Q-Learning**. Q-learning is an iterative algorithm that is very similar to iterative policy iteration.

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Sutton and Barto

This algorithm does not assume a model of the environment is present, instead learning by trial and error. By iterating over many environment-agent iteractions, a table of (state, action) pairs can be constructed that can accurately predict the expected future reward of taking an action in a particular state.

One new parameter that is introduced in this algorithm α is the learning rate. A high learning rate causes Q-values to be overwritten quickly by current rewards.

In this project I implement Q-Learning on a slightly more challenging variant of gridworld. In this task we have an agent on a 5x5 grid. The grid represents an ice skating rink. Before learning begins, three random holes are

placed on this 5x5 grid. The agent is simply tasked with skating as long as possible. If the agent moves into the hole, it receives a reward of -1000. If the agent moves without falling in the hole, it receives a reward of +100. The game ends when either the agent has skated for 1000 turns, or the agent has fallen in the hole.

One other complication that was added to make the task more challenging was that the agent had a 30% chance of "slipping". If an agent performs an action and slips, they will move twice as far as they intended.

On one run, the rink looked like this.

```
Rink:
[' ', ' ', ' ', 'X', ' ']
[' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', 'X', ' ']
[' ', 'X', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ']
```

Here the X's represent the three holes in the skating rink. After 5000 iterations the policy that the agent learned is as follows.

```
Optimal Policy:
['D', 'L', 'L', 'X', 'D']
['U', 'L', 'L', 'L', 'D']
['U', 'L', 'U', 'X', 'D']
['U', 'X', 'D', 'R', 'D']
['U', 'L', 'L', 'L', 'L']
```

Note the agent not only avoids the holes, but also avoids any movement that might cause the agent to slip into a hole. Neat.

Q-learning is definitely an improvement over the dynamic programming algorithms that we have previously discussed. However, Q-Learning is still dreadfully inefficient for tasks that have many states and actions. In high complexity environments, it becomes impossible to search the state-action space for high-reward actions. Furthermore, storing and referencing Q-values for a game with billions of states is too cumbersome or down-right impossible when real-time decisions are expected.

The solution to the problem of high complexity environments is creating agents that learn to generalize. That is, faced with similar states, the agent expects similar outcomes (even if the currrent state is completely novel).

10 Deep Q – Learning

Deep Q – Learning improves on its predecessor by allowing an agent to generalize information about the state to other, similar states. This is achieved by implementing a deep neural network to approximate the Q function. By using a parametric representation instead of a lookup-table, deep Q-learning allows real-time decision making in environments with high-dimensional state-space.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Tan F. et al (2017)

The use of machine learning in Deep Q-Learning allows generalization that results in faster, more efficient learning and inference on tasks within unknown environments. This algorithm was developed by OpenAI and introduced as a method to effectively teach reinforcement learning agents to play complicated games like Atari with only visual stimulus. OpenAI (2018)

11 Conclusion

In this paper we introduced some of the more basic algorithms developed to teach an autonomous agent to act in an unknown environment. We discussed the environment-agent interaction, as well as important concepts like value functions, policies, and iterative policy improvement.

Though we have covered much, Reinforcement Learning is an agglomeration of several disciplines, and as such has mountains of topics to discuss. Hopefully this paper serves as a brief primer towards more advanced algorithms.

For more information, check out Sutton and Barto's *Reinforcement Learning, an Introduction.*

.

References

Sutton, Richard S.., and Andrew G.. Barto. *Reinforcement Learning: an Introduction*. The MIT Press., 2018.

OpenAI,. (2018). Learning Dexterous In-Hand Manipulation.

OpenAI (2018) OpenAIFive https://openai.com/blog/openai-five/

Mnih, Volodymyr, et al. "Playing Atari with Deep Reinforcement Learning." ArXiv:1312.5602 [Cs], Dec. 2013. arXiv.org, http://arxiv.org/abs/1312.5602.

Tan F., Yan P., Guan X. (2017) Deep Reinforcement Learning: From Q-Learning to Deep Q-Learning. In: Liu D., Xie S., Li Y., Zhao D., El-Alfy ES. (eds) Neural Information Processing. ICONIP 2017. Lecture Notes in Computer Science, vol 10637. Springer, ChamAssociation for Computing Machinery. 1983. *Computing Reviews*, 24(11):503-512.