Getting into Reinforcement Learning

Cem Subakan

Mila

March 26, 2020

Goal of this presentation

- To spell out the fundamentals of Reinforcement Learning Setup (MDP), using a simple example.
- In very high level, talk about several solutions.
  - Value iteration, Policy Evaluation, Monte Carlo RL, SARSA, Q-learning, DQN, Reinforce, Reinforce with baseline, A2C.

Goal of this presentation

- To spell out the fundamentals of Reinforcement Learning Setup (MDP), using a simple example.
- In very high level, talk about several solutions.
  - Value iteration, Policy Evaluation, Monte Carlo RL, SARSA, Q-learning, DQN, Reinforce, Reinforce with baseline, A2C.
- So, no advanced Reinforcement Learning!!!

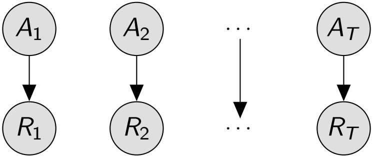
Table of Contents

The multi-armed bandit problem

- We have K choices at a given time t. We denote this with $A_t \in \{1, \ldots, K\}$.
- Each choice has an associated reward. That is, the choice A_t has an associated reward R_t.
- The ultimate goal is to maximize the sum of rewards:

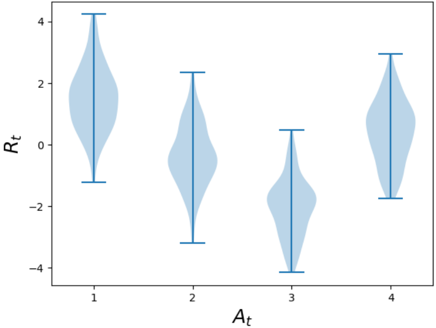
$$\max_{A_{1:T}} \sum_{t=1}^{T} R_t(A_t)$$

Dependency modeling of the bandit problem



An example bandit problem

- Below is $p(R_t|A_t)\ \forall t$:



- $p(R_t|A_t) = \mathcal{N}(\mu(A_t), \sigma^2)$

How do we solve the bandit problem

- We can just count:

$$Q_T(a) := \frac{\sum_{t=1}^{T-1} R_t \mathbf{1}_{[A_t=a]}}{\sum_{t=1}^{T-1} \mathbf{1}_{[A_t=a]}}$$

and then set $A_t = \arg\max_a Q_T(a)$.

How do we solve the bandit problem

- We can just count:

$$Q_T(a) := \frac{\sum_{t=1}^{T-1} R_t \mathbf{1}_{[A_t=a]}}{\sum_{t=1}^{T-1} \mathbf{1}_{[A_t=a]}}$$

  and then set $A_t = \arg\max_a Q_T(a)$.
- The Incremental Version:

$$Q_{t+1} = Q_t + \frac{1}{t}[R_t - Q_t]$$

How do we solve the bandit problem

- We can just count:

$$Q_T(a) := \frac{\sum_{t=1}^{T-1} R_t \mathbf{1}_{[A_t=a]}}{\sum_{t=1}^{T-1} \mathbf{1}_{[A_t=a]}}$$

  and then set $A_t = \arg\max_a Q_T(a)$.
- The Incremental Version:

$$Q_{t+1} = Q_t + \frac{1}{t}[R_t - Q_t]$$

- Notice the form:

$$NewEst. \leftarrow OldEst. + StepSize[Target - OldEstimate]$$

An Incremental Solution for the bandit problem

A simple bandit algorithm

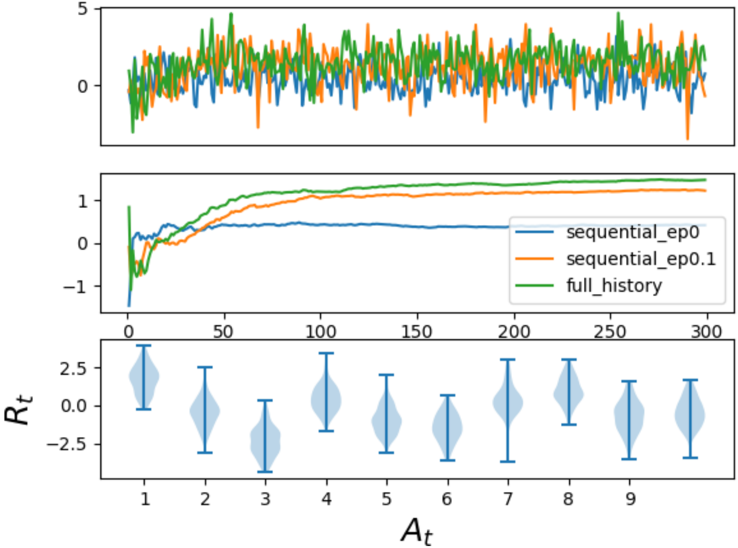
Initialize, for $a = 1$ to k:
$\quad Q(a) \leftarrow 0$
$\quad N(a) \leftarrow 0$

Loop forever:

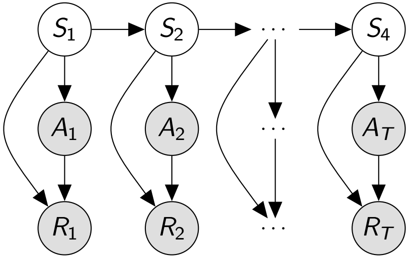
$$A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$\quad R \leftarrow bandit(A)$
$\quad N(A) \leftarrow N(A) + 1$
$\quad Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

Testing the simple bandit



Contextual bandits



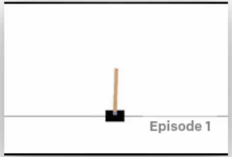
- The context changes, and consequently the reward distribution changes also.
- We have an additional challenge of associating the rewards with the context. This setup is known as associative search also.

Table of Contents

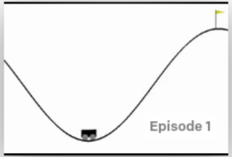
Table of Contents

Example Tasks

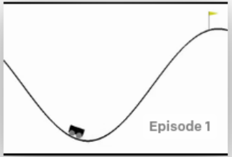
Acrobot-v1
Swing up a two-link robot.



CartPole-v1
Balance a pole on a cart.



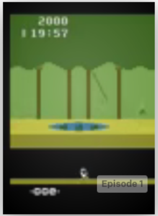
MountainCar-v0
Drive up a big hill.



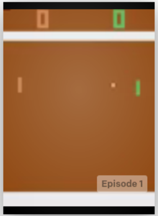
MountainCarContinuous-v0
Drive up a big hill with continuous control.

Pendulum-v0
Swing up a pendulum.

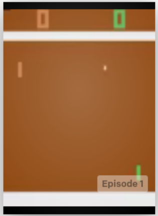
Example Tasks (Games)



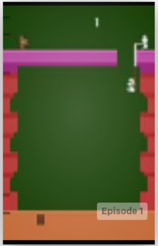
Pitfall-v0
Maximize score in the game Pitfall, with screen images as input



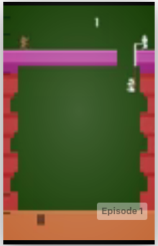
Pong-ram-v0
Maximize score in the game Pong, with RAM as input



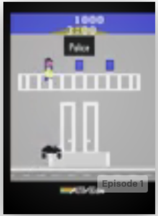
Pong-v0
Maximize score in the game Pong, with screen images as input



Pooyan-ram-v0
Maximize score in the game Pooyan, with RAM as input



Pooyan-v0
Maximize score in the game Pooyan, with screen images as input



PrivateEye-ram-v0
Maximize score in the game PrivateEye, with RAM as input

Another Example

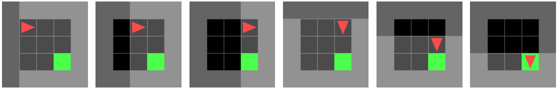
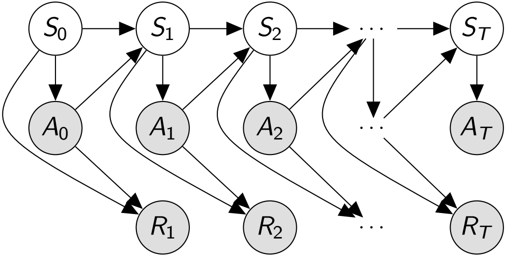


Table of Contents

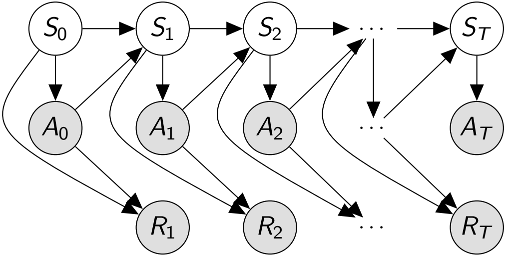
Full Reinforcement Learning



- $S_{1:T}$ – states
- $A_{1:T}$ – actions
- $R_{1:T}$ – rewards

$$
\begin{aligned}
S_{t+1}|S_t, A_t &\sim p(S_{t+1}|S_t, A_t) \\
R_{t+1}|A_t, S_t &\sim p(R_{t+1}|S_t, A_t) \\
A_t|S_t &\sim \pi(A_t|S_t),
\end{aligned}
$$

Full Reinforcement Learning



- $S_{1:T}$ – states
- $A_{1:T}$ – actions
- $R_{1:T}$ – rewards

$$\begin{aligned} S_{t+1}|S_t, A_t &\sim p(S_{t+1}|S_t, A_t) \\ R_{t+1}|A_t, S_t &\sim p(R_{t+1}|S_t, A_t) \\ A_t|S_t &\sim \pi(A_t|S_t), \end{aligned}$$

Note the Markovian assumption! Furthermore:

$$\begin{aligned} S_{t+1}|S_t, A_t &\sim p(S'|S, A), \forall t \\ R_{t+1}|A_t, S_t &\sim p(R'|S, A), \forall t \\ A_t|S_t &\sim \pi(A|S), \forall t \end{aligned}$$

What do we learn in Reinforcement Learning

- The main goal is to learn a policy $\pi(A|S)$, so as to maximize future rewards.

What do we learn in Reinforcement Learning

- The main goal is to learn a policy $\pi(A|S)$, so as to maximize future rewards.
- We usually don't know the environment dynamics $p(S_{t+1}, R_{t+1}|S_t, A_t) = p(S_{t+1}|S_t, A_t)p(R_{t+1}|S_t, A_t)$. But we are typically able to interact with the environment to sample episodes: $(S_0, A_0, R_0), (S_1, A_1, R_1), \ldots, (S_T, A_T, R_T)$. (That is, if we have access to a simulator)

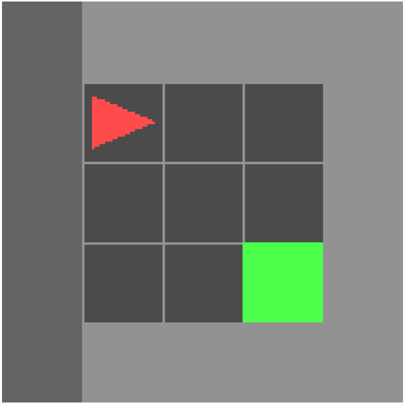
What do we learn in Reinforcement Learning

- The main goal is to learn a policy $\pi(A|S)$, so as to maximize future rewards.
- We usually don't know the environment dynamics $p(S_{t+1}, R_{t+1}|S_t, A_t) = p(S_{t+1}|S_t, A_t)p(R_{t+1}|S_t, A_t)$. But we are typically able to interact with the environment to sample episodes: $(S_0, A_0, R_0), (S_1, A_1, R_1), \ldots, (S_T, A_T, R_T)$. (That is, if we have access to a simulator)
- Some approaches first learn the environment dynamics and then do RL. (Model Based RL)

What do we learn in Reinforcement Learning

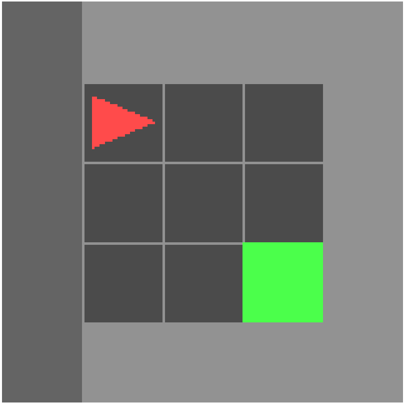
- The main goal is to learn a policy $\pi(A|S)$, so as to maximize future rewards.
- We usually don't know the environment dynamics $p(S_{t+1}, R_{t+1}|S_t, A_t) = p(S_{t+1}|S_t, A_t)p(R_{t+1}|S_t, A_t)$. But we are typically able to interact with the environment to sample episodes: $(S_0, A_0, R_0), (S_1, A_1, R_1), \ldots, (S_T, A_T, R_T)$. (That is, if we have access to a simulator)
- Some approaches first learn the environment dynamics and then do RL. (Model Based RL)
- Another family of approaches don't learn the environment, but rather interact with it. (Model Free RL)

Example Reinforcement Learning Setup



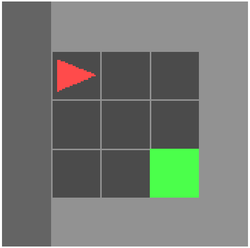
- States: $S \in \{((x, y), d), \text{where } x, y \in \{1, 2, 3\}, d \in \{W, N, E, S\}\}$. Examples: ((1, 1), E), ((3, 3), S)
- Actions: $A \in \{turnleft, turnright, goforward\}$.
- Rewards: $1 - t * c$ each time we get to green square.

Example Reinforcement Learning Setup



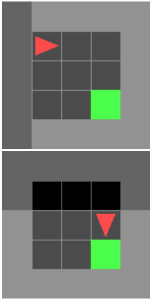
- States: $S \in \{((x, y), d), \text{where } x, y \in \{1, 2, 3\}, d \in \{W, N, E, S\}\}$. Examples: ((1, 1), E), ((3, 3), S)
- Actions: $A \in \{turnleft, turnright, goforward\}$.
- Rewards: $1 - t * c$ each time we get to green square.
- Interactive Demo

Action-State Transition tensor



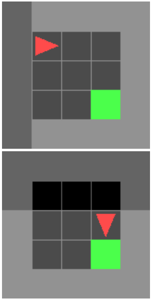
- State transition:
  $p(S' = (1,1,N)|S = (1,1,E), A = \mathit{turnleft}) = 1$
  $p(S' = (2,1,E)|S = (1,1,E), A = \mathit{forward}) = 1$
  $p(S' = (2,1,N)|S = (1,1,E), A = \mathit{forward}) = 0$
  $p(S' = (3,3,E)|S = (1,1,E), A) = 0,\ \forall A$

Action-State Transition tensor



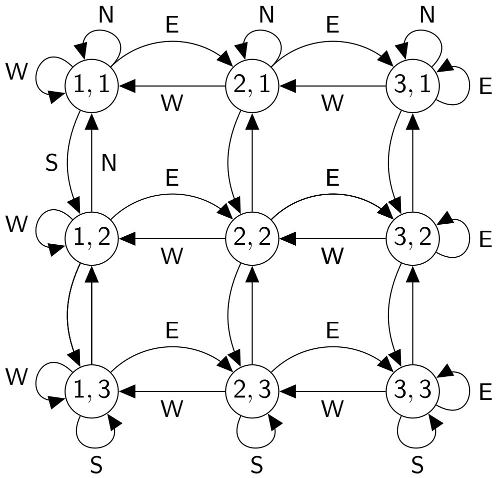
- State transition:
  $p(S' = (1,1,N)|S = (1,1,E), A = turnleft) = 1$
  $p(S' = (2,1,E)|S = (1,1,E), A = forward) = 1$
  $p(S' = (2,1,N)|S = (1,1,E), A = forward) = 0$
  $p(S' = (3,3,E)|S = (1,1,E), A) = 0, \ \forall A$
- Reward distribution:
  $p(R' = 0|S = (1,1,E), A) = 1, \ \forall A$
  $p(R' = 1 - c * t|S = (3,2,S), A = goforward) = 1$

Action-State Transition tensor



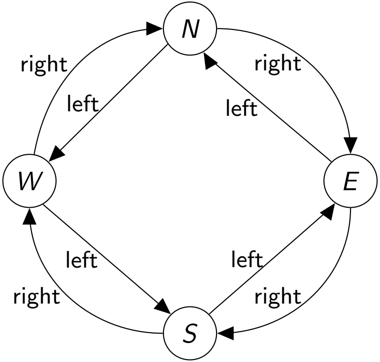
- State transition:
  $p(S' = (1,1,N)|S = (1,1,E), A = turnleft) = 1$
  $p(S' = (2,1,E)|S = (1,1,E), A = forward) = 1$
  $p(S' = (2,1,N)|S = (1,1,E), A = forward) = 0$
  $p(S' = (3,3,E)|S = (1,1,E), A) = 0, \ \forall A$
- Reward distribution:
  $p(R' = 0|S = (1,1,E), A) = 1, \ \forall A$
  $p(R' = 1 - c * t|S = (3,2,S), A = goforward) = 1$
- Policy:
  $\pi(A_t|S_t)$, we want to learn one so that the rewards are maximized!

State Transition-Action Tensor (Action=Forward)



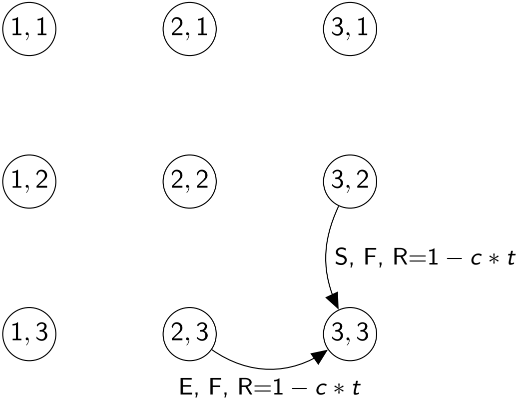
If there is an arrow, that means the corresponding entry is 1.

State Transition-Action Tensor (Action=$\{$Left,Right$\}$)



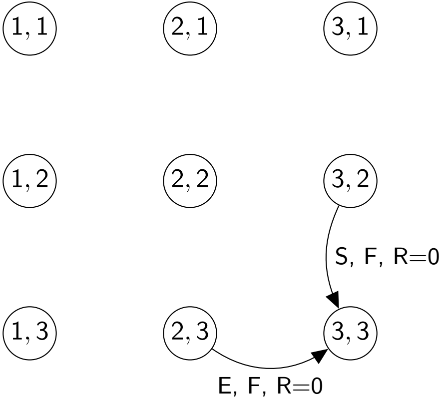
The coordinates stay the same, but the orientation (N, E, S, W) change.

Reward-State-Action Tensor



The reward is 0 if there is no arrow.

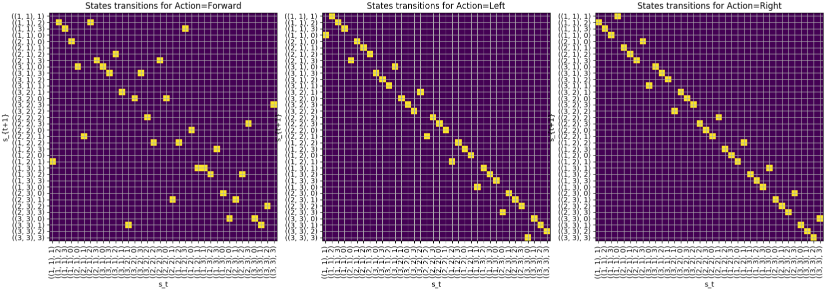
Reward-State-Action Tensor v2



The reward is -1 if there is no arrow.

The Actual State Transition-Action Tensor

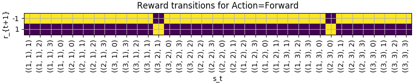
$$p(S_{t+1}|S_t, A_t)$$

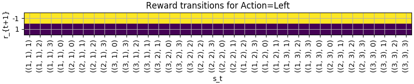


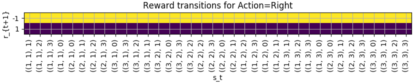
Notice that each states are sparsely connected. Each state **at max** connected **3 other** state. (Also note, 0=east, 1=south, 2=west, 3=north)

The Actual Reward Transition-Action Tensor

$$p(R_{t+1}|S_t, A_t)$$



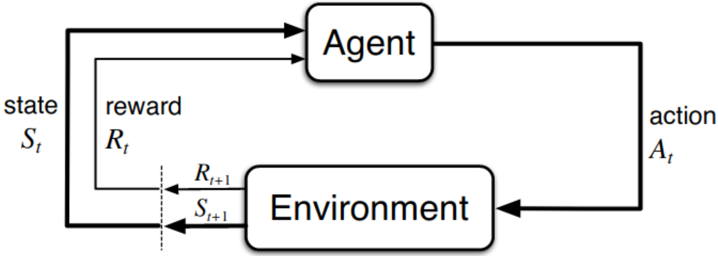




Notice how sparse are the rewards. (Also note, 0=east, 1=south, 2=west, 3=north)

Some more terminology

- Environment: The environment has a state S_t, and transitions according to $p(S_{t+1}|S_t, A_t)$ yields rewards according to $p(R_t|S_t, A_t)$.
- Agent: The agent chooses actions A_t according to the policy $\pi(A_t|S_t)$.



(* diagram taken From Sutton, Barto)

MDP Recap

- **To Recap:** States, Rewards, Policy, State Transition-Action Tensor, Reward-State-Action Tensor, Environment, Agent
- **Difference between MDP and PO-MDP:** In MDP we can fully observe the states of the environment. In PO-MDP, we either observe noisy observations regarding the state, or we observe a related representation. (The coordinates fully describe the MDP for the ongoing gridworld example, but the example below considers the case where the agent only sees what in front)

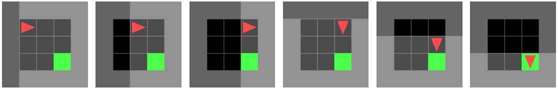


Table of Contents

But what do we optimize?

Discounted sum of future rewards

$$G_t := \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The value function (expected future returns)

$$V_\pi(s) := \mathbb{E}_\pi[G_t | S_t = s]$$

- Value Function gives the expected value of the random variable G_t given S_t for policy π.

But what do we optimize?

Discounted sum of future rewards

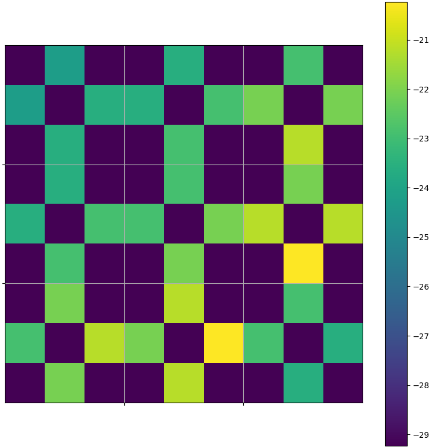
$$G_t := \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The value function (expected future returns)

$$V_\pi(s) := \mathbb{E}_\pi[G_t | S_t = s]$$

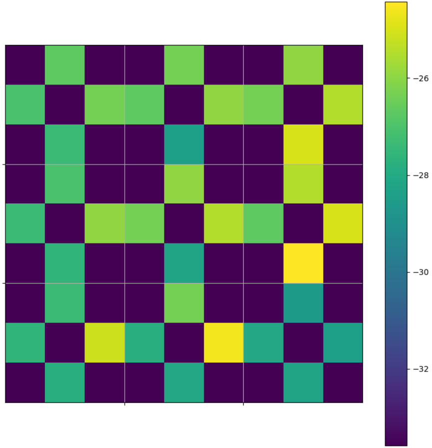
- Value Function gives the expected value of the random variable G_t given S_t for policy π.
- That is, for a given state, and a given policy what is the expected sum of future rewards.

Visualizing the Value Function for our minigridworld



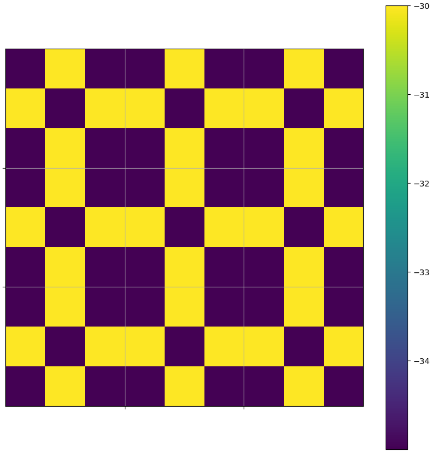
We show the value function for the optimal policy. (each square contains the values for 4 different directions).

Visualizing the Value Function for our minigridworld



We show the value function for a **learned policy**. (each square contains the values for 4 different directions).

Visualizing the Value Function for our minigridworld



We show the value function for a **random policy**. (each square contains the values for 4 different directions).

Value Function

- The value function:

$$
\begin{aligned}
V_\pi(S_t) &:= \mathbb{E}_\pi[G_t|S_t] \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k|S_{k-1}, A_{k-1}) p(R_k|A_{k-1}, S_{k-1}) \pi(A_{k-1}|S_{k-1}) G_k \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k|S_{k-1}, A_{k-1}) p(R_k|A_{k-1}, S_{k-1}) \pi(A_k|S_k)(R_{k+1} + \gamma G_{k+1})
\end{aligned}
$$

Value Function

- The value function:

$$
\begin{aligned}
V_\pi(S_t) &:= \mathbb{E}_\pi[G_t | S_t] \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k | S_{k-1}, A_{k-1}) p(R_k | A_{k-1}, S_{k-1}) \pi(A_{k-1} | S_{k-1}) G_k \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k | S_{k-1}, A_{k-1}) p(R_k | A_{k-1}, S_{k-1}) \pi(A_k | S_k)(R_{k+1} + \gamma G_{k+1}) \\
&= \sum_{A_t, S_{t+1}, R_{t+1}} \cancel{p(S_{t+1} | S_t, A_t)} p(R_{t+1} | A_t, S_t) \pi(A_t | S_t) R_{t+1} + \\
&\quad \sum_{\substack{A_t \\ S_{t+1} \\ R_{t+1}}} \pi(A_t | S_t) p(S_{t+1}, R_{t+1} | S_t, A_t) \underbrace{\sum_{\substack{A_{t+1:T} \\ S_{t+2:T} \\ R_{t+2:T}}} \prod_{k=t+2}^{\infty} \pi(A_{k-1} | S_{k-1}) p(R_k | A_{k-1}, S_{k-1}) p(S_k | S_{k-1}, A_{k-1}) \gamma G_{t+1}}_{\gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1}] = \gamma V_\pi(S_{t+1})}
\end{aligned}
$$

Value Function

- The value function:

$$
\begin{aligned}
V_\pi(S_t) &:= \mathbb{E}_\pi[G_t|S_t] \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k|S_{k-1}, A_{k-1}) p(R_k|A_{k-1}, S_{k-1}) \pi(A_{k-1}|S_{k-1}) G_k \\
&= \sum_{A_{t:T}, S_{t+1:T}, R_{t+1:T}} \prod_{k=t+1}^{\infty} p(S_k|S_{k-1}, A_{k-1}) p(R_k|A_{k-1}, S_{k-1}) \pi(A_k|S_k)(R_{k+1} + \gamma G_{k+1}) \\
&= \sum_{A_t, S_{t+1}, R_{t+1}} \cancel{p(S_{t+1}|S_t, A_t)} p(R_{t+1}|A_t, S_t) \pi(A_t|S_t) R_{t+1} + \\
&\sum_{\substack{A_t \\ S_{t+1} \\ R_{t+1}}} \pi(A_t|S_t) p(S_{t+1}, R_{t+1}|S_t, A_t) \underbrace{\sum_{\substack{A_{t+1:T} \\ S_{t+2:T} \\ R_{t+2:T}}} \prod_{k=t+2}^{\infty} \pi(A_{k-1}|S_{k-1}) p(R_k|A_{k-1}, S_{k-1}) p(S_k|S_{k-1}, A_{k-1}) \gamma G_{t+1}}_{\gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1}] = \gamma V_\pi(S_{t+1})}
\end{aligned}
$$

- $V_\pi(S_t) = \sum_{A_t} \pi(A_t|S_t) \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_\pi(S_{t+1})]$

Value Recursions

Value Function

$$
\begin{aligned}
V_\pi(S_t) =& \mathbb{E}_\pi[G_t|S_t] \\
=& \sum_{A_t} \pi(A_t|S_t) \sum_{R_{t+1},S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_\pi(S_{t+1})]
\end{aligned}
$$

Value Recursions

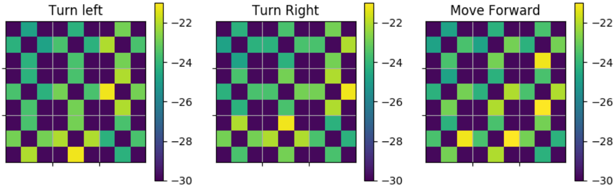
Value Function

$$\begin{aligned} V_\pi(S_t) =& \mathbb{E}_\pi[G_t|S_t] \\ =& \sum_{A_t} \pi(A_t|S_t) \sum_{R_{t+1},S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_\pi(S_{t+1})] \end{aligned}$$

Action-Value Function

$$\begin{aligned} Q_\pi(S_t, A_t) =& \mathbb{E}_\pi[G_t|S_t, A_t] \\ =& \sum_{R_{t+1},S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + V_\pi(S_{t+1})] \\ =& \sum_{R_{t+1},S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \sum_{A_{t+1}} \pi(A_{t+1}|S_{t+1}) Q_\pi(S_{t+1}, A_{t+1})] \end{aligned}$$

Visualizing the Action-Value Function



Above images are: $Q(S, A = \mathit{left})$, $Q(S, A = \mathit{Right})$, $Q(S, A = \mathit{Forward})$

Bellman-Optimality Conditions

- The optimal value functions:

$$Q_*(S, A) := \max_{\pi} Q_\pi(S, A)$$
$$V_*(S) := \max_{\pi} V_\pi(S)$$

- The optimal policy:

$$\pi_*(A|S) := \arg\max_{A'} Q_*(S, A')$$

Bellman-Optimality Conditions

- The optimal value functions:

$$Q_*(S, A) := \max_\pi Q_\pi(S, A)$$
$$V_*(S) := \max_\pi V_\pi(S)$$

- The optimal policy:

$$\pi_*(A|S) := \arg\max_{A'} Q_*(S, A')$$

- Also,

$$\begin{aligned} V_*(S) &:= \sum_A \pi_*(A|S) Q_*(S, A) \\ &= \sum_A \delta(A - \arg\max_{A'} Q_*(S, A')) Q_*(S, A) \\ &= Q_*(S, \arg\max_{A'} Q_*(S, A')) \\ &= \max_A Q_*(S, A) \end{aligned}$$

Bellman-Optimality Recursion

Bellman Value recursion:

$$
\begin{aligned}
V_*(S_t) &= \sum_{A_t} \pi(A_t|S_t) Q_*(S_t, A_t) \\
&= \max_{A} Q_*(S, A) \\
&= \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_*(S_{t+1})] \\
&= \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma \max_{A_{t+1}} Q_*(S_{t+1}, A_{t+1})]
\end{aligned}
$$

Bellman-Optimality Recursion

Bellman Value recursion:

$$\begin{aligned} V_*(S_t) &= \sum_{A_t} \pi(A_t|S_t) Q_*(S_t, A_t) \\ &= \max_{A} Q_*(S, A) \\ &= \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_*(S_{t+1})] \\ &= \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma \max_{A_{t+1}} Q_*(S_{t+1}, A_{t+1})] \end{aligned}$$

Interpretation: We make the decision that yields largest R_T at time T and then make the best decision at $T-1$, and go back until t.

Value Iteration

- If we know the tables $p(S_{t+1}|S_t, A_t)$, and $p(R_{t+1}|S_t, A_t)$ then this recursion converges:

$$V_\pi(S_t) = \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_\pi(S_{t+1})]$$

Value Iteration

- If we know the tables $p(S_{t+1}|S_t, A_t)$, and $p(R_{t+1}|S_t, A_t)$ then this recursion converges:

$$V_\pi(S_t) = \max_{A_t} \sum_{R_{t+1}, S_{t+1}} p(S_{t+1}, R_{t+1}|A_t, S_t)[R_{t+1} + \gamma V_\pi(S_{t+1})]$$

Value Iteration, for estimating $\pi \approx \pi_*$

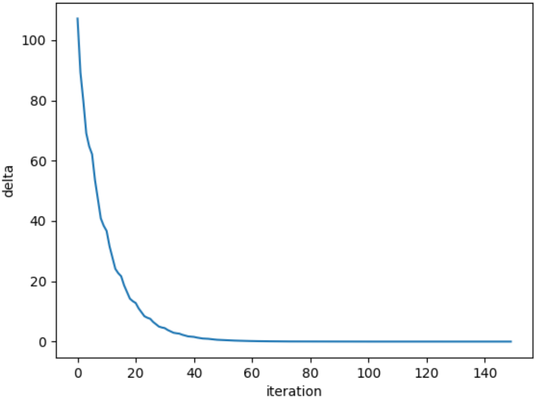
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

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

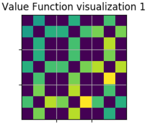
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

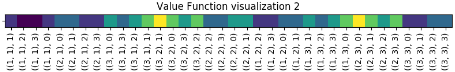
The argmax makes the policy improve:
$V_{\pi'}(s) \geq Q_\pi(s, \pi'(s)) = \max_a Q_\pi(s, a) \geq V_\pi(s), \ \forall\ s$

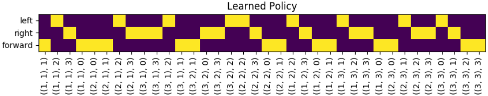
Value Iteration to recover the optimal policy



Result of the Value Iteration

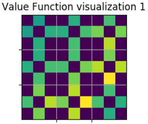


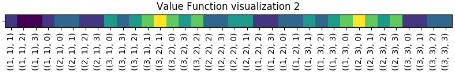


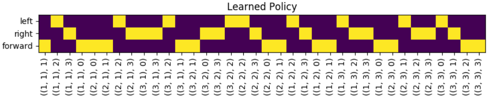


(Note, 0=east, 1=south, 2=west, 3=north)

Result of the Value Iteration







(Note, 0=east, 1=south, 2=west, 3=north)

`Click for Policy Replay`

Table of Contents

Monte Carlo Methods

- In general, we do not have the transition tables. We can however create random episodes using the current policy. And then update our policy according to the returns.

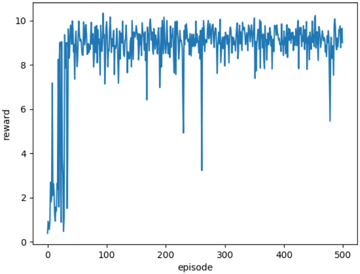
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

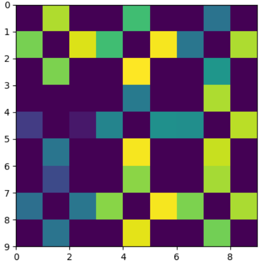
Initialize:
  $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
  Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
  Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

- The argmax makes the policy improve:
$V_{\pi'}(s) \geq Q_\pi(s, \pi'(s)) = \max_a Q_\pi(s,a) \geq Q_\pi(s, \pi(s)) \geq V_\pi(s)$

Basic MC Algo. on our GridWorld Example





TD Methods (n) step Monte Carlo Methods

- Instead of sampling whole sequences, we can make updates on one step updates of the form:

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha[G_t - V(S_t)] \\ &= V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \end{aligned}$$

- This also holds for the action value function:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\textit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Off policy TD - Q learning

- We change the updates:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

- With epochs the TD error goes to zero. Notice that the TD error is the Bellman optimality condition.

Off policy TD - Q learning

- We change the updates:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

- With epochs the TD error goes to zero. Notice that the TD error is the Bellman optimality condition.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop 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

Table of Contents

Table of Contents

Deep Q-Network, DQN (Playing Atari w DRL)

- The fixed point algorithm earlier, updates discrete tables.
- In DQN, We instead do function approximation such that $Q_*(S_t, A_t) \approx Q(S_t, A_t; \theta)$. We can then apply this on large, continuous state spaces.
- We then minimize the TD error.

$$L(\theta_i) = \mathbb{E}[Q(s, a; \theta_i) - (r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}))]$$

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

DQN intuition

- We have two networks. Target network gets us to compute: $(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}))$
- Policy network outputs $Q(s, a; \theta_i)$.
- Then we try to minimize the absolute difference between the two networks.
- The network parameters are transferred after a certain number of iterations.
- We evaluate the loss function using sampled transitions.

Sampling

```
all_rewards = []
for i_episode in range(args.num_episodes):
    # Initialize the environment and state
    obs = env.reset()

    rewards_ep = []
    for t in range(args.frames_per_proc):
        # Select and perform an action
        action = algo.select_action(preprocess_obss([obs], device=device))
        next_obs, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device).float()
        rewards_ep.append(reward.item())

        # Store the transition in memory
        algo.memory.push(obs, action, next_obs, reward)

        # Move to the next state
        obs = next_obs

        # Perform one step of the optimization (on the target network)
        algo.optimize_model()
    # Update the target network, copying all weights and biases in DQN
    if (i_episode % algo.target_update) == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

Updates

```
        batch = self.memory.transition(*zip(*transitions))

        # Compute a mask of non-final states and concatenate the batch elements
        # (a final state would've been the one after which simulation ended)
        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                              batch.next_state)), device=self.device) #, dt
ype=torch.uint8)
        non_final_next_states = self.preprocess_obss(batch.next_state, device=self.device)

        #non_final_next_states = torch.cat([self.preprocess_obss([s], device=self.device).i
mage for s in batch.next_state if s is not None])
        state_batch = self.preprocess_obss(batch.state, device=self.device)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
        # columns of actions taken. These are the actions which would've been taken
        # for each batch state according to policy_net
        state_action_values = self.policy_net(state_batch)[0].logits.gather(1, action_batch
)

        # Compute V(s_{t+1}) for all next states.
        # Expected values of actions for non_final_next_states are computed based
        # on the "older" target_net; selecting their best reward with max(1)[0].
        # This is merged based on the mask, such that we'll have either the expected
        # state value or 0 in case the state was final.
        next_state_values = torch.zeros(self.batch_size, device=self.device)
        next_state_values[non_final_mask] = self.target_net(non_final_next_states)[0].logit
s.max(1)[0].detach()
        # Compute the expected Q values
        expected_state_action_values = (next_state_values * self.gamma) + reward_batch

        # Compute Huber loss
        loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze
(1))
```

Table of Contents

Policy Gradient Methods

- So far we have only worked with updating Value Functions. This time we compute a gradient of expected returns.

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) =:& \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \\
=& \nabla_\theta \int p(\tau|\theta) r(\tau) d\tau \\
=& \int \nabla_\theta p(\tau|\theta) r(\tau) d\tau \\
=& \int p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) r(\tau) d\tau \\
=& \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log p(\tau|\theta) r(\tau)] \\
\approx& \sum_t \nabla_\theta \pi(a_t|s_t;\theta) r(s_t, a_t, s_{t+1})
\end{aligned}
$$

Policy Gradient Methods

- So far we have only worked with updating Value Functions. This time we compute a gradient of expected returns.

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) =: & \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \\
= & \nabla_\theta \int p(\tau|\theta) r(\tau) d\tau \\
= & \int \nabla_\theta p(\tau|\theta) r(\tau) d\tau \\
= & \int p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) r(\tau) d\tau \\
= & \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log p(\tau|\theta) r(\tau)] \\
\approx & \sum_t \nabla_\theta \pi(a_t|s_t;\theta) r(s_t, a_t, s_{t+1})
\end{aligned}
$$

- We have a Monte Carlo estimate for the gradient of the expected returns. The gradient is amplified if $r(.)$ is large.

Policy Gradient Methods

- So far we have only worked with updating Value Functions. This time we compute a gradient of expected returns.

$$\begin{aligned}\nabla_\theta J(\pi_\theta) =& :\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \\ =& \nabla_\theta \int p(\tau|\theta) r(\tau) d\tau \\ =& \int \nabla_\theta p(\tau|\theta) r(\tau) d\tau \\ =& \int p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) r(\tau) d\tau \\ =& \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log p(\tau|\theta) r(\tau)] \\ \approx& \sum_t \nabla_\theta \pi(a_t|s_t;\theta) r(s_t, a_t, s_{t+1})\end{aligned}$$

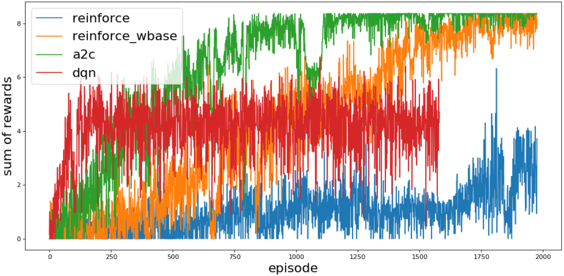
- We have a Monte Carlo estimate for the gradient of the expected returns. The gradient is amplified if $r(.)$ is large.
- This framework is the basis for several algorithms:
  - $r_t(s_t, a_t, s_{t+1}) = G_t \rightarrow$ REINFORCE
  - $r_t(s_t, a_t, s_{t+1}) = G_t - \hat{v}(s_t) \rightarrow$ REINFORCE with baseline
  - $r_t(s_t, a_t, s_{t+1}) = R_t + \gamma \hat{v}(s_{t+1}) - \hat{v}(s_t)$ One step actor critic

Policy Gradient Methods Workflow

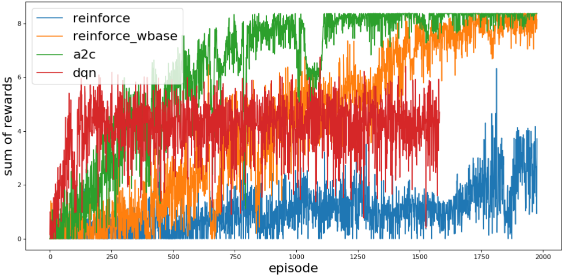
```
for i in range(args.num_episodes):
    print('episode {}'.format(i))

    update_start_time = time.time()
    exps = algo.collect_experiences_parallelfor()

    algo.update_parameters(exps, preprocess_obss)
```

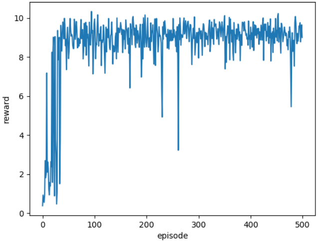
```
def update_parameters(self, exps):

    gam = self.discount

    T = exps['rewards'].shape[0]
    all_Gs = []
    for t in range(T):
        gams = ((torch.ones(T - t)*gam) ** torch.arange(T-t).float()).to(self.device)
        all_Gs.append((exps['rewards'][t:T] * gams).sum().to(self.device))

    all_Gs = torch.tensor(all_Gs).to(self.device)

    self.optimizer.zero_grad()
    log_p = (- (all_Gs*exps['log_probs'][:T]).sum())
    print(log_p.item())
    log_p.backward(retain_graph=True)

    self.optimizer.step()
```

Comparing, recapping 4 DRL algorithms (and MC)



Comparing, recapping 4 DRL algorithms (and MC)

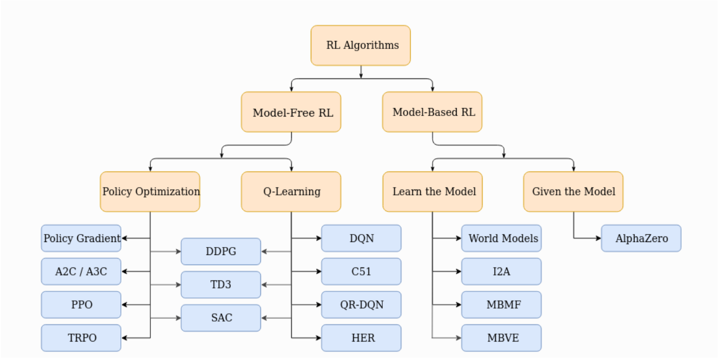




Some conclusions after playing with RL things

- Deep RL is Difficult
- We compared DQN, REINFORCE, RF. w. baseline, a2c on 3x3 Minigridworld environment.
- We used the same policy network, and same hyper parameters for all models.
- Changing the Reinforce coefficient improves the policy gradient algorithms.
- DQN converges faster, but gets stuck in a local optimum. ϵ is very important.
- Recap: Policy Gradient Algos. are on-policy. DQN is off-policy. Both do function approximations. In a2c and Reinforce with base, we also learn an approximation for the value function.

Taxonomy of RL Algorithms



We covered DQN, Policy Gradient (Reinforce), a2c (a3c is the asynchronous version), and looked at what to do given the model in the discrete case.

Conclusions / Things I couldn't get to

- We did a very basic introduction.
- Code is available on my github.
- Resources: Sutton, Barto book; DQN paper; A3C paper; Minigrid world environment; open ai spinning up page; torch-ac package; rl-starting-files pytorch repo.
- Model based (practical) RL, multiagent RL, off policy without exploration learning.