RL WORKSHOP

Eyob Dagnachew

DATA SCIENCE CLUB

Schedule

1. RL intro
2. MDP’s
3. Policy Iteration
4. MC learning
5. Function approx
6. Policy gradient methods
7. Self play
8. Env

RL Intro

- What is RL?
  - At a very high level, RL is the process of learning to act and accomplish goals in **dynamic** environments
  - This is inspired by some ideas in psychology that behavior is reinforced by through the idea that behaviors that result in praise/pleasure tend to repeat and behaviors that bring “pain” go extinct
  - This leads to RL being a “trial and error” learning paradigm
- Agents receive rewards for working towards a goal
- Different representations of rewards and goals can alter the agents ability to perform well in specific scenarios found in RL (online learning, stochastic policies, etc)

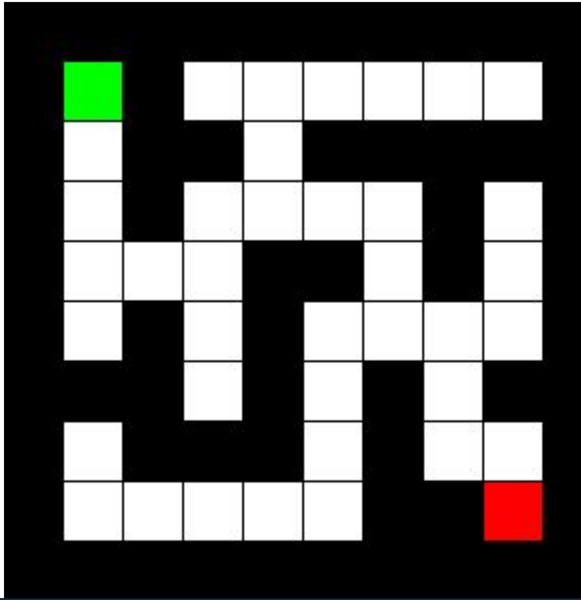
MDP'S: Formalizing RL problem spaces

- The MDP representation of the RL process is built on the Markov assumption: that all information needed to make an a acting given your current state is all encoded within your current state, no explicit access to previous states is needed

$$\mathbb{P}\left[R_{t+1} = r, S_{t+1} = s' \,|\, S_0, A_0, R_1, \ldots, A_{t-1}, R_t, S_t, A_t\right] = \mathbb{P}\left[R_{t+1} = r, S_{t+1} = s' \,|\, S_t, A_t\right]$$

MDP'S: Formalizing RL problem spaces

- Represented often times by 5 main items
  - State Space (S): Representation of all possible states to be in
  - Action Space (A): Representations of all possible actions to take
  - Transition Function (T:SxSxA → Real): Given state and action, take action
    - Represents P(s'|s, a): Probability of ending up in new state given state action pair
  - Reward Function (R:SxA → Real): Reward for specific state action pair
  - The last two can be mixed
    - P(s', r|s, a) - Transition function can include reward
  - Policy (denoted by π) (π: S → A): Maps states to actions to take, what the agent uses to make decisions



MDP'S: Cont

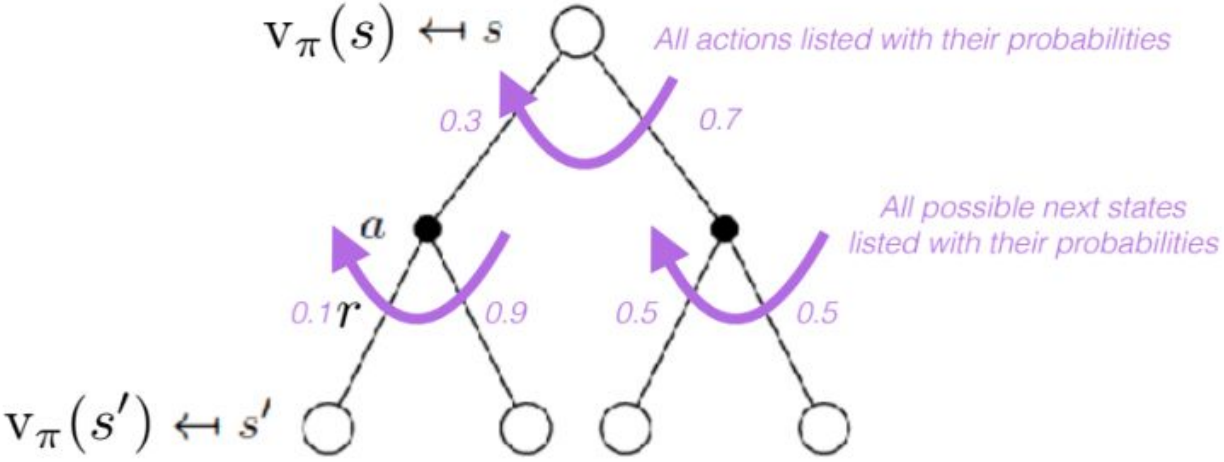
- So how can we act on this representation?
- Goal - Learn optimal policy: π* S → A
  - Maximize discounted reward
  - Gain best possible performance in an environment

$$v_\pi(s) = E_{a\sim\pi}[q_\pi(s,a)|S_t = s, A_t = a] = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

- How can we judge the quality of a policy?
  - Value/ Q-value functions
- Value function (V:S → Real):
  - Represents the estimated value of being in a state while following policy
- Q-Value function (Q:SxA → Real):
  - Represents the estimated value of being in a state and taking an action, then following policy

$$q(s,a) = \mathbb{E}[R_{t+1} + \gamma v(s')|S_t = s, A_t = a] = \sum_{r,s'} (r + \gamma v(s'))p(s',r|s,a)\ .$$

MDP'S: Value function visualization



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

Policy Iteration: Optimizing policy with the value function

- Policy Iteration
  - Evaluate policy by estimating the values of each step till convergence
  - Improve your policy using reward and estimated values
- Major assumption: we know the *reward* and *transition* functions

Policy Iteration: Pseudocode

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\quad \Delta \leftarrow 0$
   $\quad$ Loop for each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s))\big[r + \gamma V(s')\big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 1: Policy iteration, taken from Section 4.3 of Sutton & Barto's RL book (2018).

MC methods: Dynamics free learning

- Policy iteration was built using the assumption of environmental dynamics (transition function, reward functions,etc)
    - How can we still learn a policy assuming we don't have those dynamics?
- One method is plausible: Monte carlo learning
    - Core idea: instead of using the dynamics of the function to determine what state we end up and the rewards we get from that, we use our own experiences in the environment
    - We update the value of a state after simulation an entire episode starting from that state

MC methods: Dynamics free learning

- Goal: learn $v_\pi(s)$ from episodes of experience under policy π:

$$S_1, A_1, R_2, ..., S_k \sim \pi$$

- Remember that the return is the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

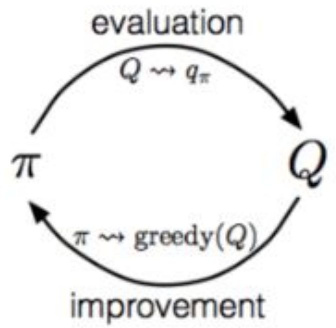
- Remember that the value function is the expected return:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

- Monte-Carlo policy evaluation **uses empirical mean return instead of expected return**

MC Learning Cont

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*$$



- MC policy iteration step: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: greedify with respect to value (or action-value) function

MC Learning: Pseudocode

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow \text{average}(Returns(S_t))$

FA: moving on from tabular methods

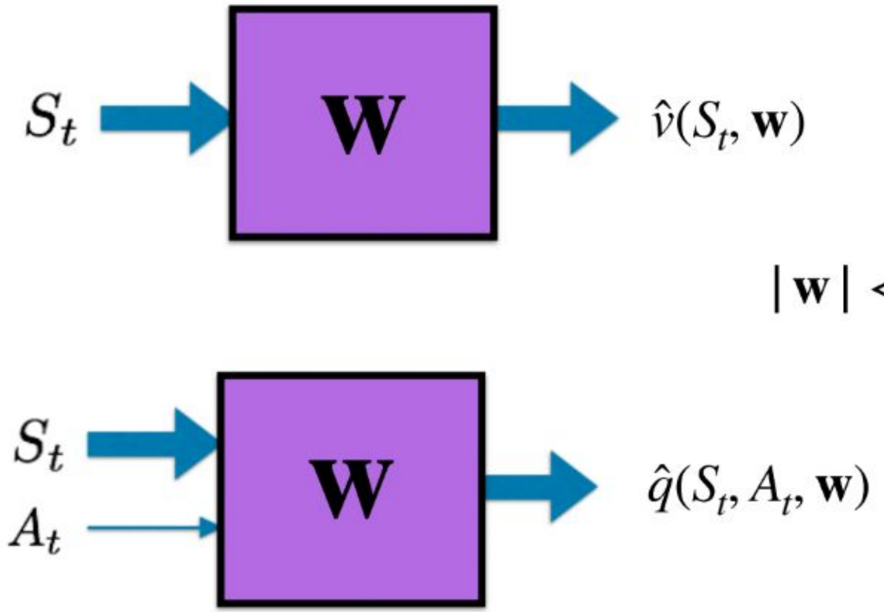
- In both previous RL methods we continuously update each states value function through various methods
  - Policy iteration: we update by sampling every possible next state given an action
  - MC-learning: update state value with bootstrapped estimate return
- But with FA we instead parameterize the value function with specific weights instead to better generalize learning of policy
- From there we can use any linear function approximator to utilize GD to improve the weights performance in accuracy in relation to an objective function

FA: moving on from tabular methods

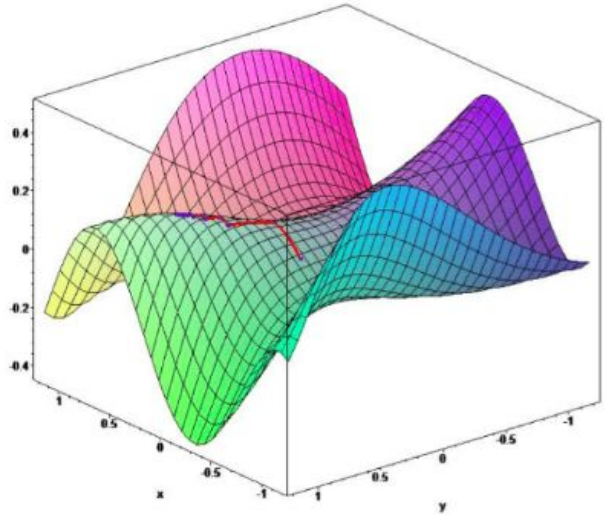
- Goal: find parameter vector w minimizing mean-squared error between the true value function $v_\pi(S)$ and its approximation $\hat{v}(S, \mathbf{w})$:

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[\left(v_\pi(S) - \hat{v}(S, \mathbf{w})\right)^2\right]$$

FA: moving on from tabular methods



$$|\mathbf{w}| << |\mathcal{S}|$$



Gradient descent

- To find a local minimum of J(w), adjust w in direction of the negative gradient:

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

- Starting from a guess $\mathbf{w}_0$
- We consider the sequence $\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \ldots$ s.t. : $\mathbf{w}_{n+1} = \mathbf{w}_n - \frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}_n)$
- We then have $J(\mathbf{w}_0) \geq J(\mathbf{w}_1) \geq J(\mathbf{w}_2) \geq \ldots$

Gradient descent

- Let J(w) be a differentiable function of parameter vector w
- Define the gradient of J(w) to be:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

FA: example: TD learning w/O FA

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

FA: TD learning with FA

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^n \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\boldsymbol{\theta}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\big[R + \gamma\hat{v}(S',\boldsymbol{\theta}) - \hat{v}(S,\boldsymbol{\theta})\big]\nabla\hat{v}(S,\boldsymbol{\theta})$
    $S \leftarrow S'$
  until S' is terminal

Policy gradient methods: A new paradigm

- Instead of parameterizing the value function of the state, parameterize the state and as a larger result the policy itself, removing the need for value functions
- This means we can parameterize the policy along whatever lines we want, including, including having the policy parameters be the parameters of a probability distribution, allowing for probabilistic optimal policies, ideal for situations like poker where actions are sampled from this distribution

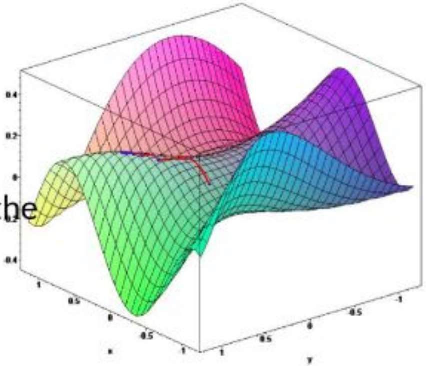
Policy optimization

- Let $U(\Theta)$ be a policy objective (loss) function
- General policy optimization pseudocode goes as such
    - Initialize policy parameters (Θ)
    - Sample trajectories by deploying policy with current parameterization
    - Compute gradient vector $\nabla U(\Theta)$
    - Update policy parameters as such: $\Theta + \alpha\nabla U(\Theta)$ (where α is step size)

Policy optimization: an alternate view

Policy Gradient

- Let U(θ) be any policy objective function
- Policy gradient algorithms search for a local maximum in U(θ) by ascending the gradient of the policy, w.r.t. parameters θ



$$\theta_{new} = \theta_{old} + \Delta\theta$$

$$\Delta\theta = \alpha \nabla_\theta U(\theta)$$

α is a step-size parameter (learning rate)

is the policy gradient

$$\nabla_\theta U(\theta) = \begin{pmatrix} \frac{\partial U(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial U(\theta)}{\partial \theta_n} \end{pmatrix}$$

Policy gradient: the gradient of the policy objective w.r.t. the parameters of the policy

Types of policy parameterizations

Policy functions

deterministic continuous policy

$a = \pi_\theta(s)$

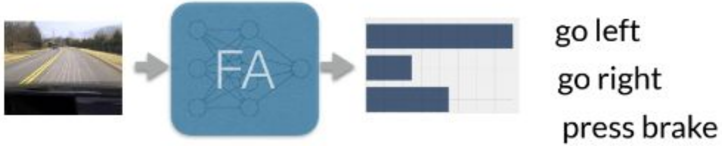
e.g. outputs a steering angle directly

stochastic continuous policy

$a \sim \mathcal{N}(\mu_\theta(s), \sigma^2_\theta(s))$

FA for stochastic multimodal continuous policies is an active area of research

(stochastic) policy over discrete actions



Outputs a distribution over a discrete set of actions

REINFORCE: taking advantage of policy gradients

From this method, as well as deriving a specific gradient we can state the final algorithm for this workshop using Policy gradients

0. Initialize policy parameters θ

1. Sample trajectories $\{\tau_i = \{s_t^i, a_t^i\}_{t=0}^{T}\}$ by deploying the current policy $\pi_\theta(a_t \mid s_t)$.

2. Compute gradient vector $\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) G_t^{(i)}$

3. $\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$

REINFORCE: taking advantage of policy gradients

Algorithm 1 REINFORCE

procedure REINFORCE
    Start with policy model π_θ
    repeat:
        Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
        for t *from* $T-1$ *to* 0:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
        $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
        Optimize π_θ *using* $\nabla L(\theta)$
end procedure

MISC Topics

Gymnasium

- Environment Collection: Gym is a Python library offering a range of environments for testing reinforcement learning algorithms.

- OpenAI Support: Developed by OpenAI, Gym ensures quality environments compatible with various reinforcement learning techniques.

- Simplified Usage: Gym provides a user-friendly interface, streamlining the process of experimenting with reinforcement learning algorithms.

Gymnasium: env

- Environment Interface: The env class in Gym defines a common interface for interacting with reinforcement learning environments.

- Unified Interaction: It abstracts the interaction between agents and environments, providing consistent methods for actions, observations, and rewards.
  - Some methods include, env.step, env.display
  - Some fields include: action_space, observation_space
- Good to look through documentation

Self Play

- Self-Play Concept: Self-play is a reinforcement learning technique where an agent learns by playing against itself rather than against fixed opponents or a pre-existing dataset.
- Dynamic Opponent: In self-play, the agent's opponent evolves alongside its own learning process, adapting to the agent's current capabilities and strategies.
- Continuous Improvement: Through iterative self-play, agents can continuously improve their strategies, leading to robust and adaptive behavior in a variety of environments or games.

Similar(ish) example: Generative adversarial Imitation learning(GAIL)

Activity

Complete (and correct) and implementation of the REINFORCE Algorithm

Collab link:
https://tinyurl.com/2s4fskcj