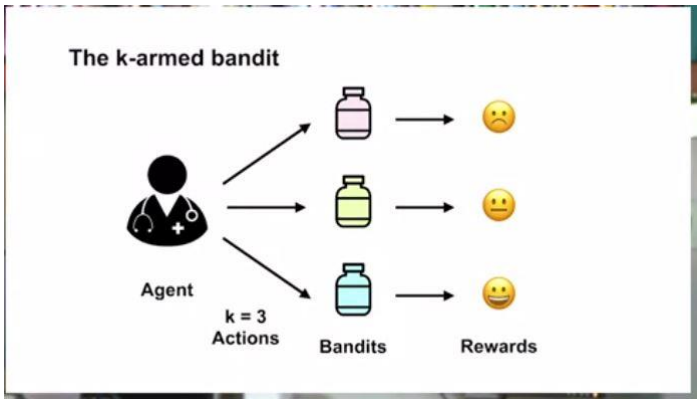
COURSE 1: Fundamentals of Reinforcement learning.

This is a example of sequential decision making.

K-armed bandit problem



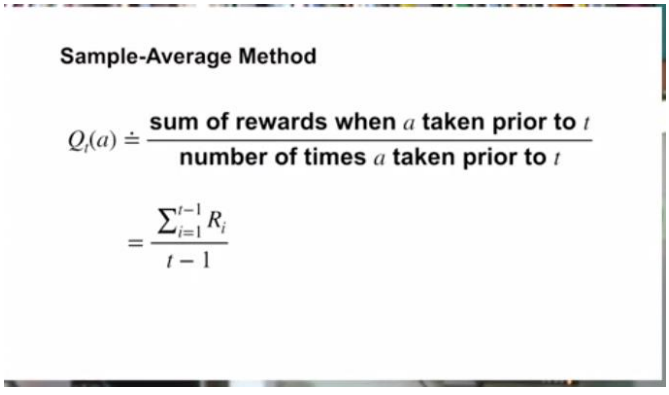
Action-Values

- The value is the expected reward

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a] \quad \forall a \in \{1, \ldots, k\}$$
$$= \sum_r p(r \mid a)\, r$$

- The goal is to maximize the expected reward

The value function q*(q-star) is unknown to the agent so we use sample average method for estimating q*.



We expand the equation using incremental update rule :

Incremental update rule

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_t(a)}\left(R_t - Q_t(a)\right)$$

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Exploration-Exploitation

Exploration versus Exploitation

- Exploration - improve knowledge for long-term benefit
- Exploitation - exploit knowledge for short-term benefit

The agent cannot explore and exploit continuously so it uses Epsilon-greedy action selection to

See when to explore and when to exploit.

Epsilon-Greedy Action Selection

$$A_t \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ Q_t(a) & \text{with probability } 1-\epsilon \\ a \sim Uniform(\{a_1 \ldots a_k\}) & \text{with probability } \epsilon \end{cases}$$

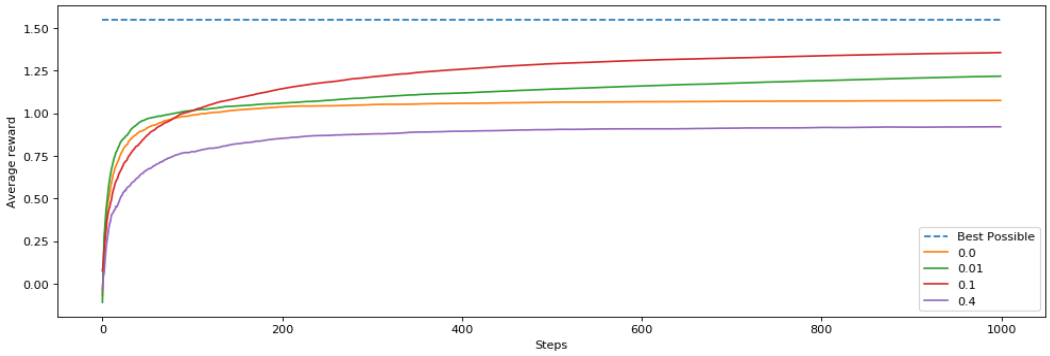
Upper-Confidence Bound (UCB) Action Selection

$$A_t \doteq argmax\left[Q_t(a) + c\sqrt{\frac{ln\ t}{N_t(a)}}\right]$$

Exploit

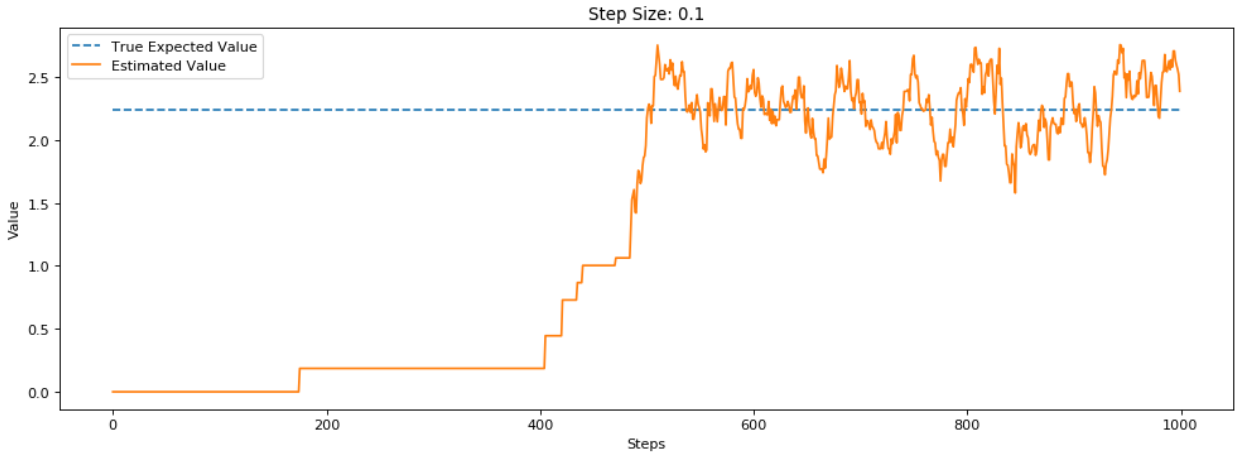
Explore

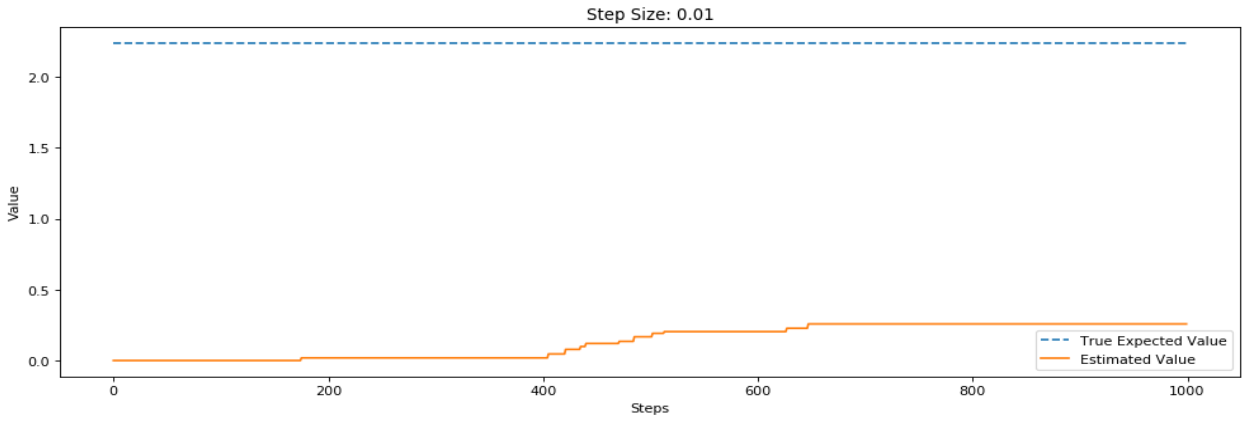
We compare greedy action and different epsilon values in test bed environment for some runs and plotted this results :

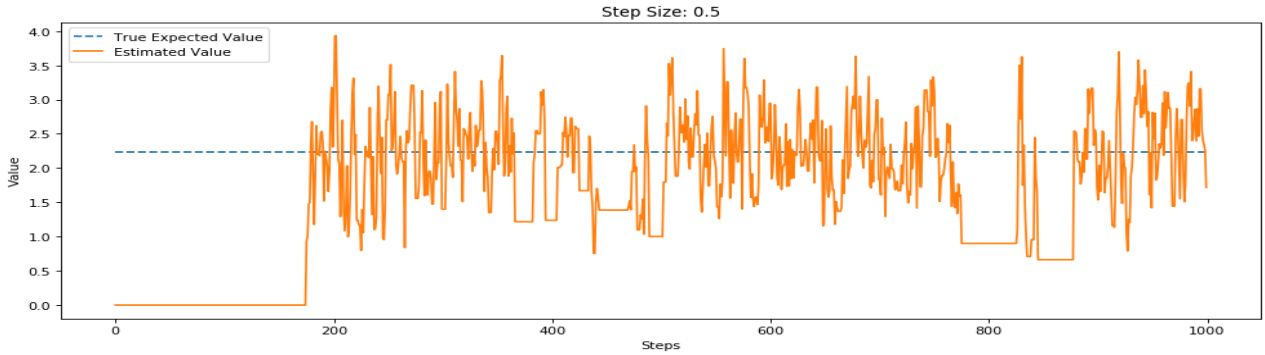


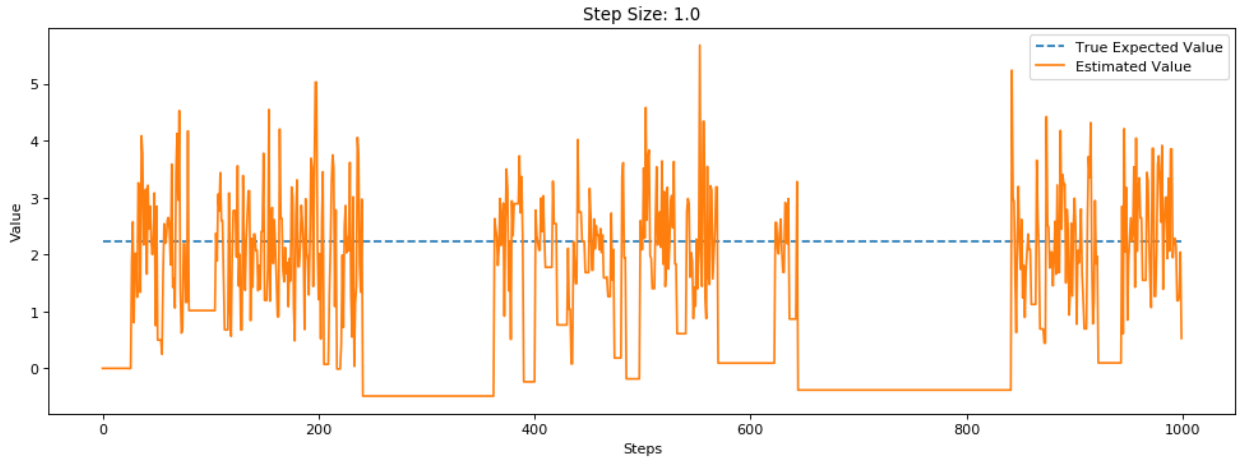
When epsilon values are more it explores most of the time not allowing to perform the action which it has learnt. So it performs worst. Epsilon=0.1 performs better than epsilon=0.01 because the later one does not explore more.so it is always good to choose moderate level of epsilon values. Greedy action performs less than epsilon=0.1 it does not explore at all.

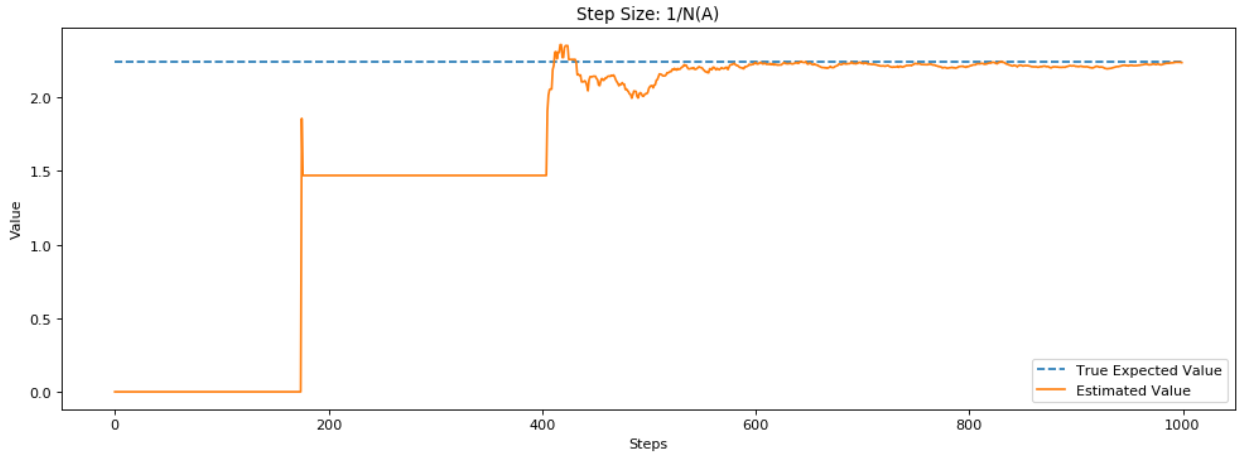
IMPACT ON STEP-SIZE:







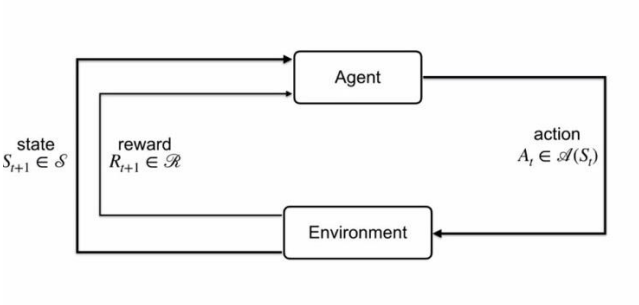




A step size of 0.01 makes such small updates that the agent's value estimate of the best action does not get close to the actual value. Step sizes of 0.5 and 1.0 both get close to the true value quickly but are very susceptible to stochasticity in the rewards. The updates overcorrect too much towards recent rewards, and so oscillate around the true value. This means that on many steps, the action that pulls the best arm may seem worse than it is. A step size of 0.1 updates quickly to the true value and does not oscillate as widely around the true values as 0.5 and 1.0. This is one of the reasons that 0.1 performs quite well. Finally, we see why 1/N(A) performed well. Early on while the step size is still reasonably high it moves quickly to the true expected value, but as it gets pulled more its step size is reduced which makes it less susceptible to the stochasticity of the rewards.

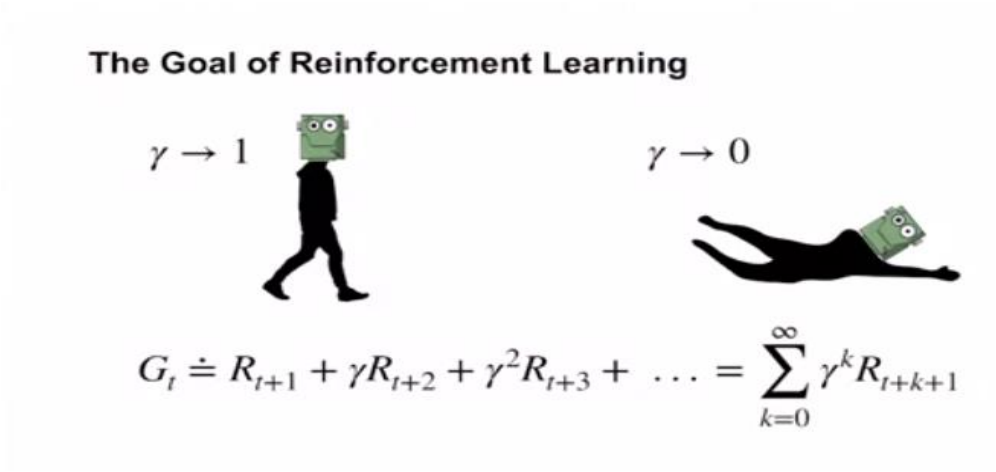
Does this mean that 1/N(A) is always the best? When might it not be? One possible setting where it might not be as effective is in non-stationary problems. You learned about non-stationarity in the lessons. Non-stationarity means that the environment may change over time. This could manifest itself as continual change over time of the environment, or a sudden change in the environment.

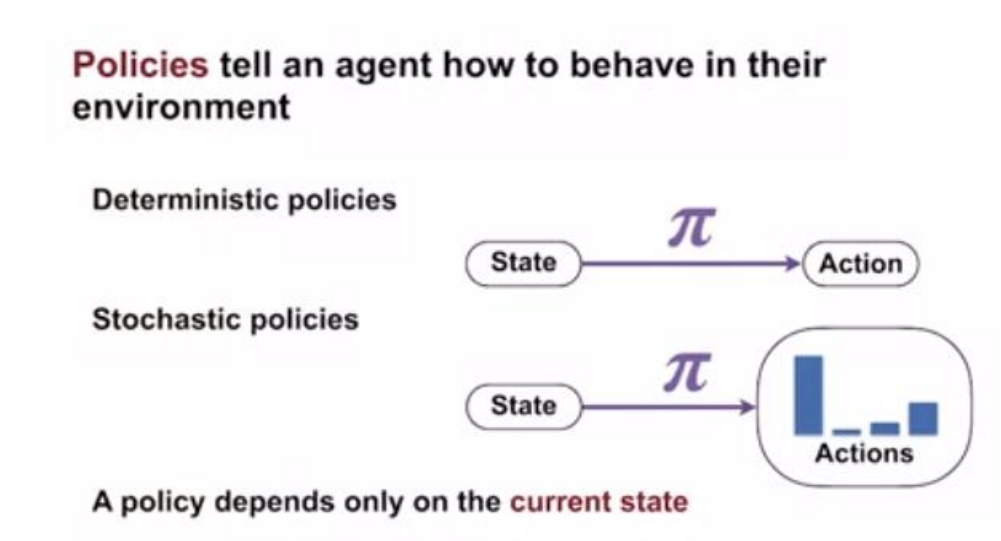
Markov Decision process :

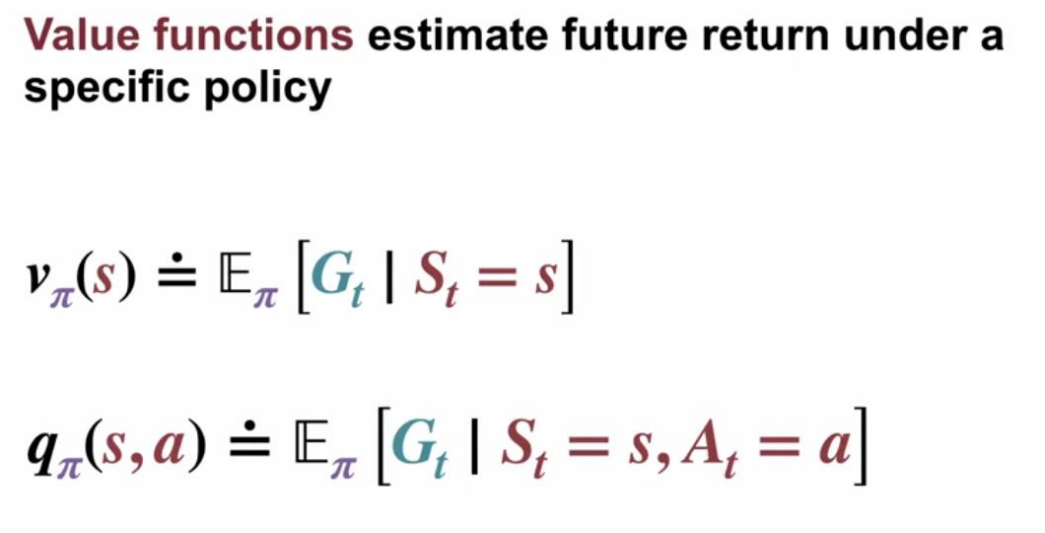


MDPs formalize
the problem of an agent interacting with an environment. The agent and environment interact at discrete time steps. At each time, the agent observes the current state of the environment. Based on this state, the agent selects an action.

After that, the environment transitions to a new state and emits a reward. Remember, the agent's choices have long-term consequences. The action it selects influences future states and rewards. The goal of reinforced learning is to maximize total future reward. This often means balancing immediate reward with the long-term consequences of actions







Value functions are like magic. Value functions capture the future total reward under a particular policy.

We discussed two kinds of value functions: state value functions, and action value functions. The state value function gives the expected return from the current state under a policy. The action value function gives the expected return from state S if the agent first selects actions A and follows Pi after that. Value functions simplify things by aggregating many possible future returns into a single number

Bellman equations define a relationship between the value of a state, or state-action pair, and its possible successors

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

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

The Bellman equation for the state value function gives the value of the current state as a sum over the values of all the successor states, and immediate rewards. The Bellman equation for the action value function gives the value of a particular state-action pair as the sum over the values of all possible next state-action pairs and rewards

optimal policies ,optimal state value function, optimal action value function

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

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

The optimal state value function is equal to the highest possible value in every state. Every optimal policy shares the same optimal state value function. The same is true for optimal action value functions and optimal policies. Like all value functions, the optimal value functions have Bellman equations. These Bellman equations do not reference a specific policy. This amounts to replace in the policy in the Bellman equation with a max over all actions. The optimal policy must always select the best available action. We can extract the optimal policy from the optimal state value function. But to do so, we also need the one-step dynamics of the MDP. We can get the optimal policy with much less work if we have the optimal action value function. We simply select the action with the highest value in each state.

Policy Evaluation and Policy Iteration

Iterative policy evaluation takes the Bellman equation for v pi and turns it into an update rule. It produces a sequence of better and better approximations to v pi. Control refers to the task of improving a policy.The policy improvement theorem tells us how to construct a better policy from a given policy. The new policy pi prime is produced by simply greenifying with respect to the current values. Pi prime is guaranteed to be strictly better than pi, unless pi was already optimal

Policy evaluation is the task of determining the state-value function v_π, for a particular policy π

Iterative Policy Evaluation

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

$$\downarrow$$

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

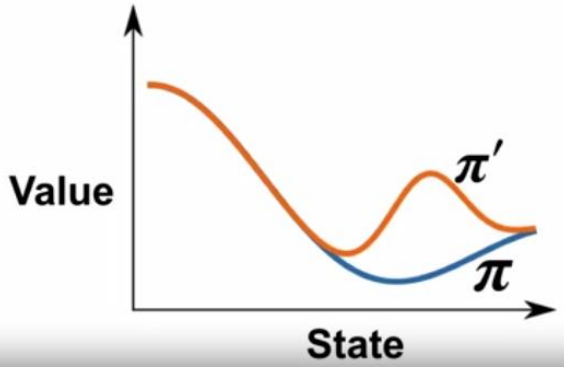
and turns it into an update rule.

Control refers to the task of improving a policy

Policy improvement theorem

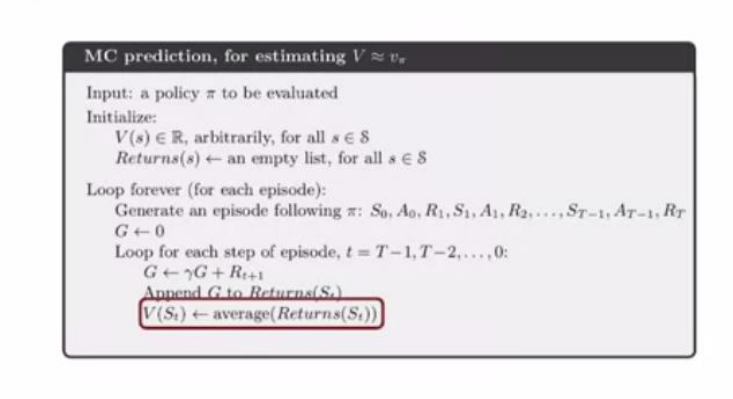
$$\pi'(s) \doteq \operatorname*{argmax}_a \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

$\pi' > \pi$ unless π is optimal



Monte Carlo methods :

Monte Carlo algorithms are sample-based methods. They can be used when the model is unavailable or hard to write down. Monte Carlo algorithms estimate value functions by averaging over multiple observed returns. They wait for the full return before updating their values. Therefore, we use Monte Carlo only for episodic MDPs.

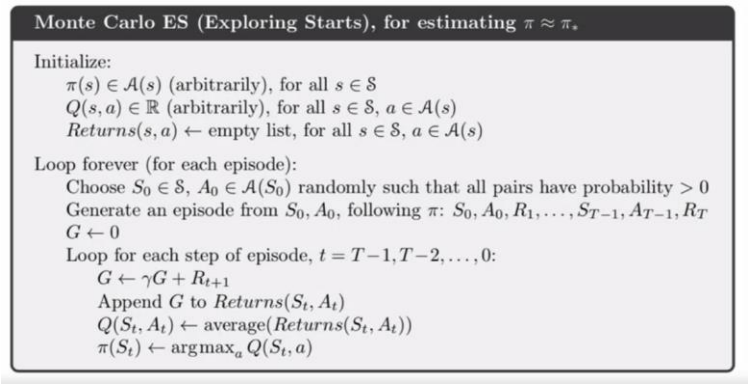


MC prediction, for estimating $V \approx v_\pi$

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, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t)$
$V(S_t) \leftarrow \text{average}(Returns(S_t))$

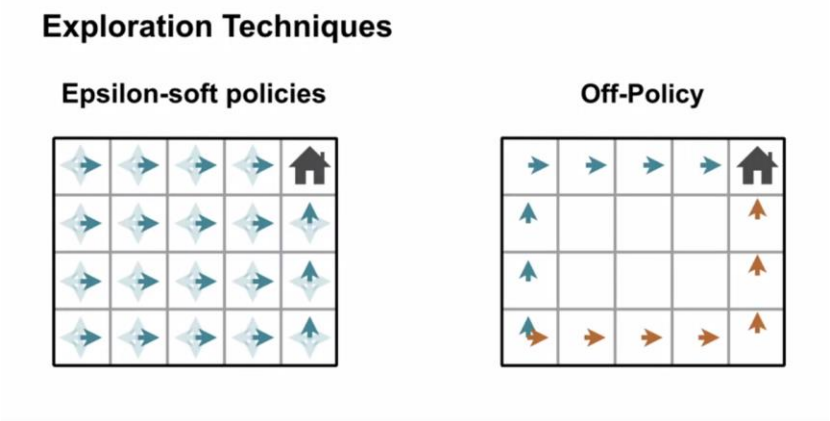
With exploration starts :



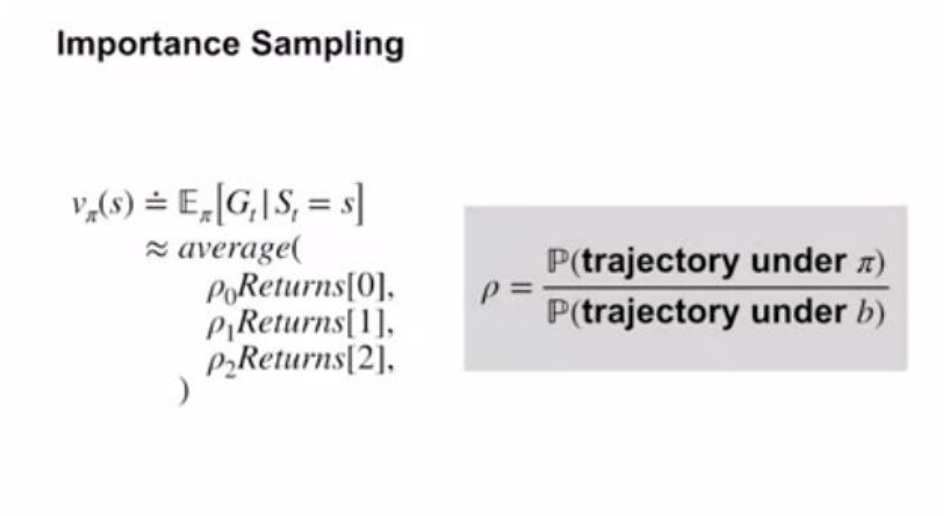
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

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, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$



Learning on policy with Epsilon-soft policies and learning off-policy. For the first strategy, the agent follows and learns about a stochastic policy. It usually takes the greedy action. A small fraction of the time it takes a random action. This way the value estimates for all state action pairs are guaranteed to continue to improve over time. This on policy strategy forced us to learn a near optimal policy instead of an optimal policy.



The ratio re-weights the samples. It increases the importance of returns that were more likely to be seen under Pi and it decreases those that were unlikely. The sample average effectively contains the right proportion of each return so that in in expectation it is as if the returns had been sampled under Pi.

Temporal difference Learning

TD can update its value estimates on each step of the episode. It does not have to wait for the episode to complete. It just has to remember the previous state. How does TD compare to Monte Carlo and dynamic programming? Well, TD often converges faster than Monte Carlo, TD does not require a model unlike dynamic programming, and TD is online and fully incremental unlike either Monte Carlo or dynamic programming.

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'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ ←
    $S \leftarrow S'$
  until S is terminal

In the long run, TD learned more quickly than Monte Carlo and achieved better final error.

The TD control algorithms are based on Bellman equations. We learned about three of them. Sarsa uses a sample-based version of the Bellman equation.

It learns Q-pi.Q-learning uses the Bellman optimality equation. It learns Q-star. Expected sarsa uses the same Bellman equation as Sarsa, but samples it differently. It takes an expectation over the next action values. What's the story with on-policy and off-policy learning? Sarsa is a on-policy algorithm that learns the action values for the policy it's currently following. Q-learning is an off-policy algorithm that learns the optimal action values. And Expected Sarsa is both an on-policy and an off-policy algorithm that can learn the action values for any policy.

TD control and Bellman equations

$$q_\pi(s,a) = \sum_{s',r} p(s',r \mid s,a)\left(r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s',a')\right)$$

$$q_*(s,a) = \sum_{s',r} p(s',r \mid s,a)\left(r + \gamma \max_{a'} q_*(s',a')\right)$$

Sarsa

$$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

On-policy

Expected Sarsa

$$R_{t+1} + \gamma \sum_{a'} \pi(a' \mid S_{t+1}) Q(S_{t+1}, a')$$

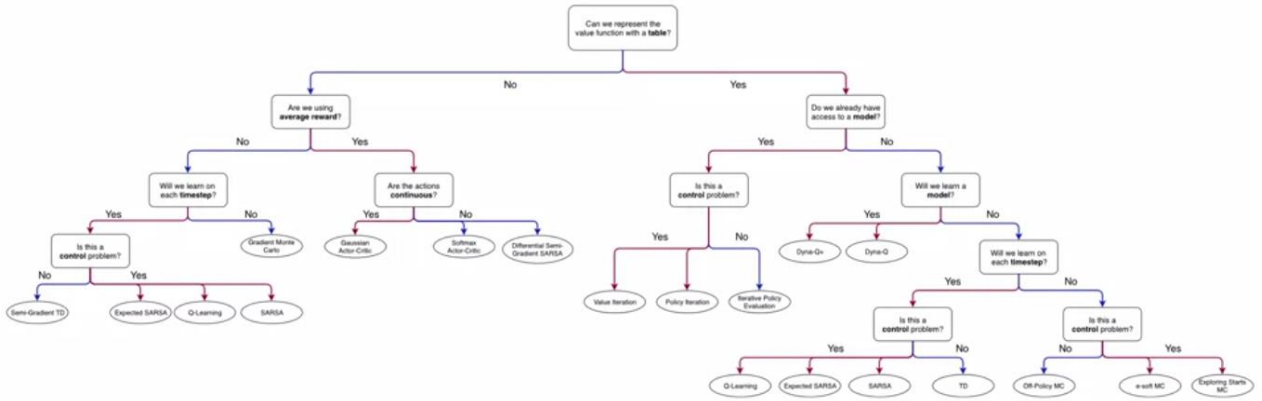
Off-policy

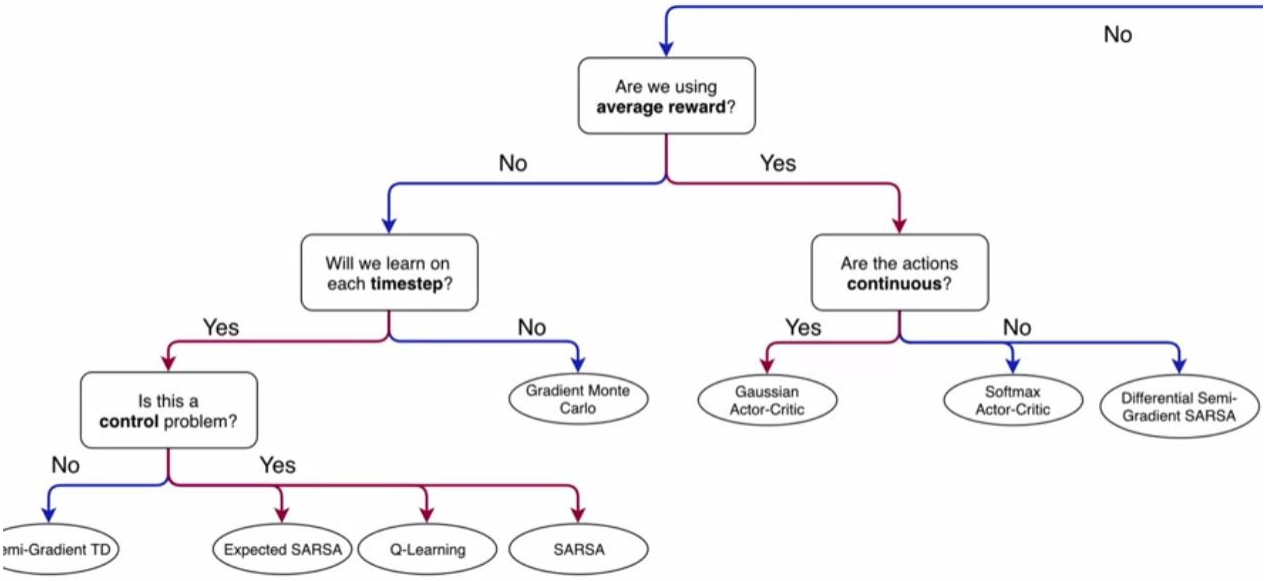
Q-learning

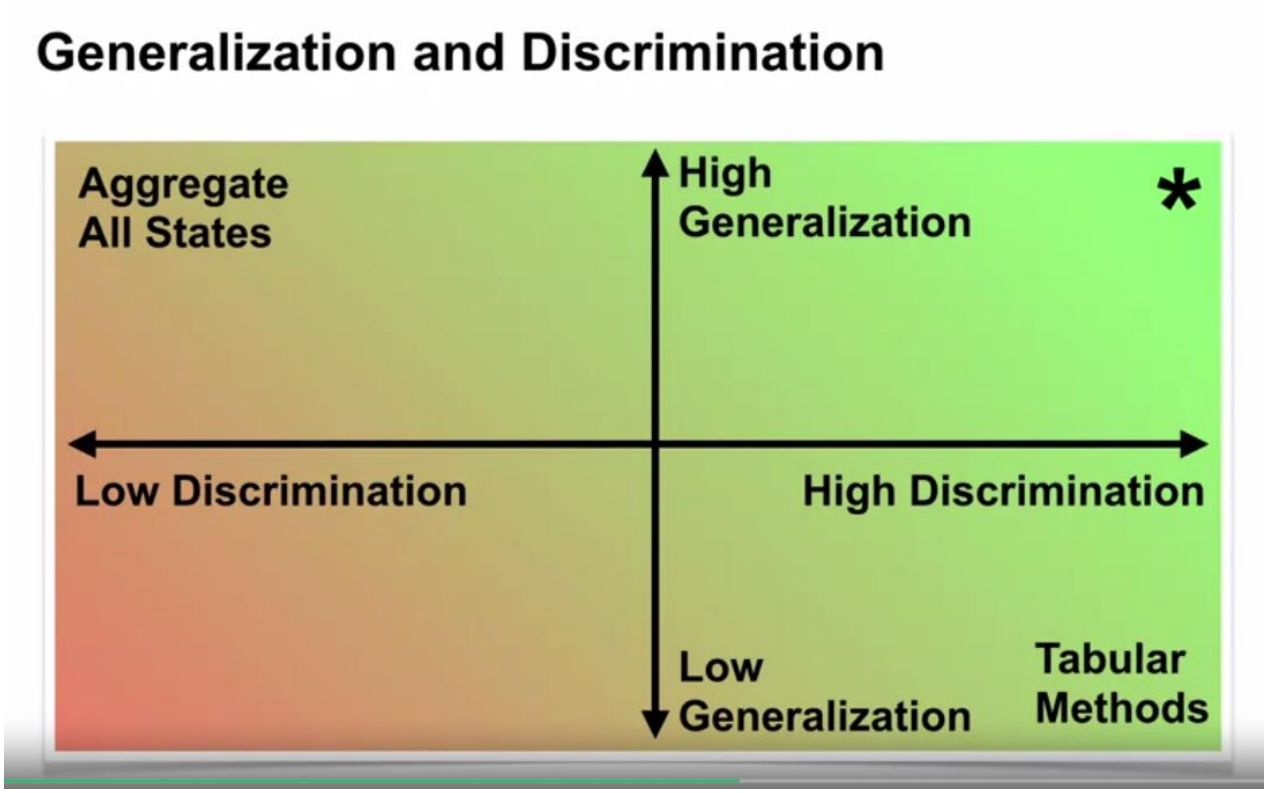
$$R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$$

Off-policy

Course-3 : Prediction and control methods:







We talked about two key properties of function approximation, generalization and discrimination. Updating the value of one state can improve the value estimates of other states. This generalization can make learning faster. We also want our value function to assign different values when states are very different. We call this discrimination.

Mean Squared Value Error Objective

A real-valued quantity which measured the distance between our approximation and the true values

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

The Gradient Monte Carlo algorithm performs stochastic gradient descent using sampled returns to update the weights. This means, it can learn approximate value from the experience generated by an agent interacting with the world. We also introduced semi gradient TD as an approximation to stochastic gradient descent. Semi-Gradient TD takes advantage of bootstrapping, and in practice may converge much faster than Monte Carlo. Semi-Gradient part of the name reminds us that it's not a gradient descent algorithm.

Optimizing the $\overline{VE}$

- **Stochastic Gradient Descent**

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

- **Gradient Monte Carlo**

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha[\ G_t \ - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

State Aggregation

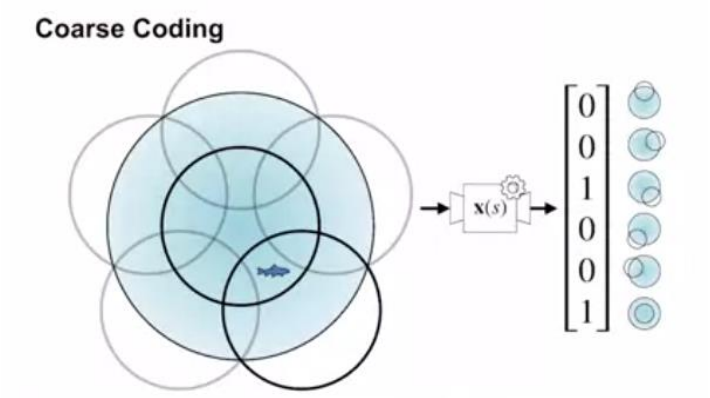
- Which state aggregation resolution do you think is the best after running 2000 episodes? Which state aggregation resolution do you think would be the best if we could train for only 200 episodes? What if we could train for a million episodes?
- Should we use tabular representation (state aggregation of resolution 500) whenever possible? Why might we want to use function approximation?

From the plots, using 100 state aggregation with step-size 0.05 reaches the best performance: the lowest RMSVE after 2000 episodes. If the agent can only be trained for 200 episodes, then 10 state aggregation with step-size 0.05 reaches the lowest error. Increasing the resolution of state aggregation makes the function approximation closer to a tabular representation, which would be able to learn exactly correct state values for all states. But learning will be slower.

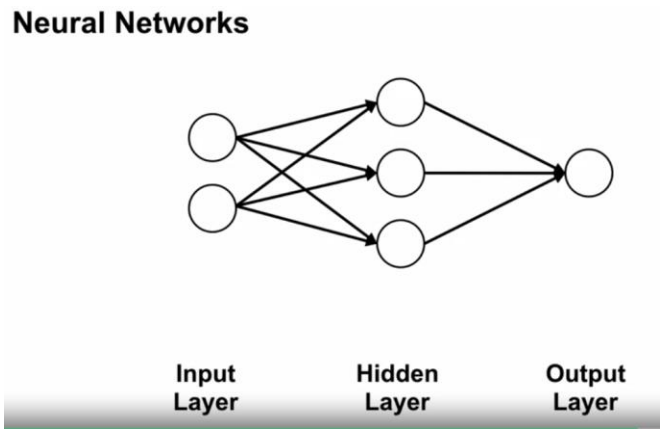
Step-Size

- How did different step-sizes affect learning?

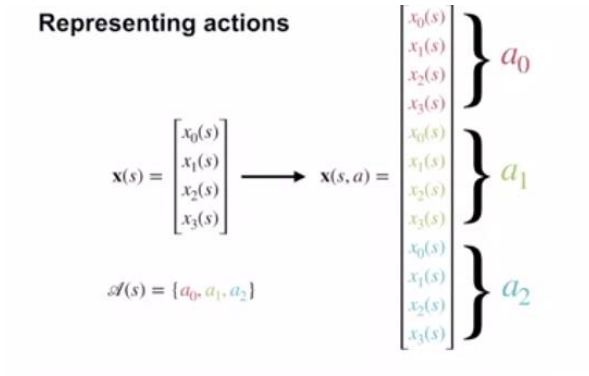
The best step-size is different for different state aggregation resolutions. A larger step-size allows the agent to learn faster, but might not perform as well asymptotically. A smaller step-size causes it to learn more slowly, but may perform well asymptotically.



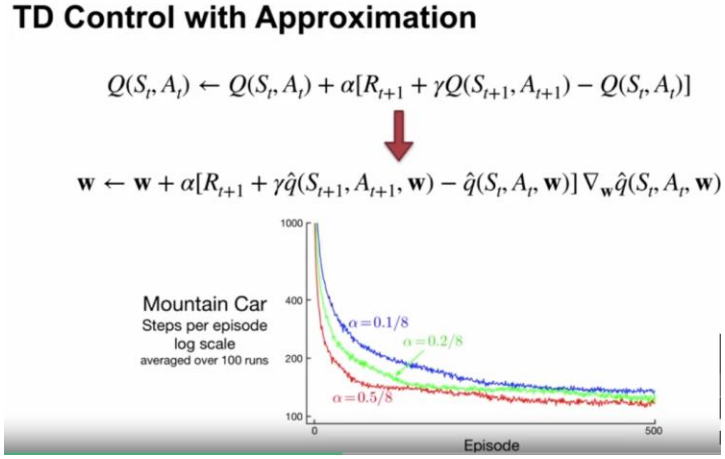
two-dimensional course coding is represented by these overlapping circles.Each circle is a feature that is one when the state is inside the circle, and zero when the state is outside the circle. Next, we discuss a particular type of course coding called tile coding.



With coarse coding techniques, the representation is fixed before learning. A feed-forward neural network uses a series of layers to produce a representation. In each layer, multiple neurons received the same input and produce distinct outputs. These outputs are then fed to the next layer and the process repeats. Each neuron computes its output by taking a weighted sum of the inputs and passing the sum through an activation function.



how to estimate action values with function approximation. If the action space is discrete, it's probably easiest to stack the state features. If the action space is continuous or you want to generalize over actions, the action can be passed as an input like any other state variable.



SARSA, expected SARSA, and Q-Learning. These are all extensions of the tabular control algorithms we covered in Course 2, the only difference between these algorithms and their tabular counterparts are the update equations. The updates are all adapted for function approximation in the same way, using the gradient to update the weights. We also saw how episodic SARSA could be used to solve the mountain car problem. In this case, the larger step size is 0.5 was able to learn more quickly.

Average Reward

$$r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r$$

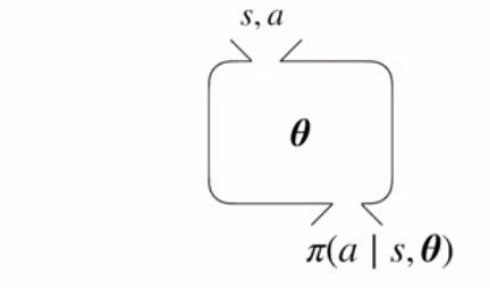
$$G_t = \boxed{R_{t+1} - r(\pi)} + \boxed{R_{t+2} - r(\pi)} + \boxed{R_{t+3} - r(\pi)} + \dots$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)\big(r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a')\big)$$

we talked about exploration, optimistic initialization can be used with some structured feature representations like tele-coding. But in general, it's not clear how to optimistically initialize values with nonlinear function approximators like neural networks, and it might not behave as expected, for example the optimism may fade too quickly. Epsilon-greedy can be used regardless of the function approximator.

regardless of the function approximator. Finally, we talked about a new way to think about the continuing control problem. Instead of maximizing the discounted return from the current state, we can think about maximizing the average reward that a policy receives overtime. We defined differential returns and differential values, these enable the agent to assess the relative value of actions in the average reward setting. Finally, we introduced differential semi-gradient SARSA that approximates differential values to learn policies.

Parameterized Policy



We started this week by making the shift from parameterize action values to parameterize policies. Parameterized policies taken state action pairs and output the associated action probabilities. This function is parameterized by a vector of parameters denoted by Theta.

Softmax Parameterization

$$\pi(a \mid s, \boldsymbol{\theta}) \geq 0 \qquad \text{for all } a \in \mathcal{A} \text{ and } s \in \mathcal{S}$$

$$\sum_{a \in \mathcal{A}} \pi(a \mid s, \boldsymbol{\theta}) = 1 \qquad \text{for all } s \in \mathcal{S}$$

$$\pi(a \mid s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}}$$

The Advantages of Policy Parameterization

- The ability to autonomously converge to a deterministic policy over time
- The ability to learn stochastic policies
- The fact that a good policy might be easier to learn and represent than precise action-values

The Average Reward Objective

$$r(\pi) = \sum_{s} \mu(s) \sum_{a} \pi(a \mid s, \boldsymbol{\theta}) \sum_{s',r} p(s', r \mid s, a) r$$

Our strategy to optimize this objective was to use stochastic gradient descent. For that, we needed an estimate of the gradient of the average reward. This is challenging because the state distribution Mu depends on the policy parameters.

The Policy Gradient Theorem

$$\nabla r(\pi) = \sum_{s} \mu(s) \sum_{a} \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi}(s, a)$$

The policy gradient theorem provides an expression for the gradient of the average reward objective that's convenient to optimize. This form allows us to estimate the gradient by sampling states from Mu by following the policy, Pi. Using the policy gradient theorem, we derive the actor-critic algorithm. Actor-critic simultaneously learns a parameterized policy, the actor, and an estimate of the policies value function, the critic. The Theta air from the critic reflects that the actor did better or worse than the critic expected. The actor is trained to favor actions that exceed the critic's expectations, the critic is trained to improve its value estimates of the actor so that it knows what value it should expect for this actor.

The Actor-Critic Algorithm