Policy Updates and Policy Gradients

Team MehtaKnights
(Ethan Mehta, Sean Lin, Jaiveer Singh)

Roadmap

- **Connection to EECS 16AB**
- **Policy Updates and Policy Gradients**
  - Motivating Example
    - Taxi World
  - Agents, States, Rewards
  - Policy
    - Action - Value Policies
    - Policy Gradients
  - Policy Gradient Methods
  - Actor Critic Methods
    - Asynchronous Advantage Actor-Critic (A3C)
- **Appendix A: Markov Decision Processes (MDPs)**
- **Appendix B: Offline Learning: Techniques for Solving MDPs (Introduction to Policy)**
- **Appendix C: Reinforcement Learning**

Connection to EECS 16AB

- What is Reinforcement Learning?
  - **Reinforcement Learning** is about training a model or **Agent** (decision-maker) that continually observes the **State** of its environment, and then takes the 'best' **Actions** to maximize **Reward**.
- Spiritually Reminiscent of EECS 16AB Control Theory
  - State Space Idea You Have Seen
    - State ⇒ A vector encapsulating the current status of Environment
  - Closed-Loop Control Connection
    - In closed-loop control we were trying to find input u(t) that would get us to a desired state $x^*(t)$.
    - In RL, we have access to **actions** that we can take. Generalization of notion of input.
    - Uncertainty is also modeled in the RL environment's transition function. This is like the random disturbances we had to deal with in closed loop control.
  - Essentially adding more degrees of freedom to control law!

Connection to EECS 16AB

- What is Reinforcement Learning?
  - **Reinforcement Learning** is about training a model or **Agent** (decision-maker) that continually observes the **State** of its environment, and then takes the 'best' **Actions** to maximize **Reward**.
- Solving RL Problems boils down to finding the best **Actions** given the state of the environment! This is called a **policy** and will be the focus of everything that follows!

Motivating Example: Taxi World

- **Taxi World**
  - **Agent:** An Autonomous Taxi Cab
  - **Goal:**
    - Pick-up passenger at the pink highlighted letter (RGBY) location.
    - Take the passenger to the blue highlighted drop-off location in as few moves as possible.
    - As quickly as possible!
  - **Actions:**
    - Up, Down, Left, Right, or Pick-up, Drop-off
- Self-Driving Car? This is a complicated problem!

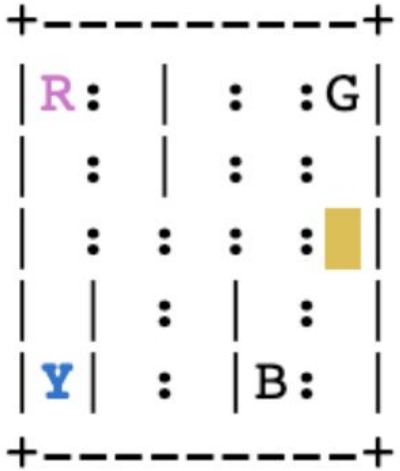


Figure 1: Taxi World

Motivating Example: Taxi World

- **Taxi World**

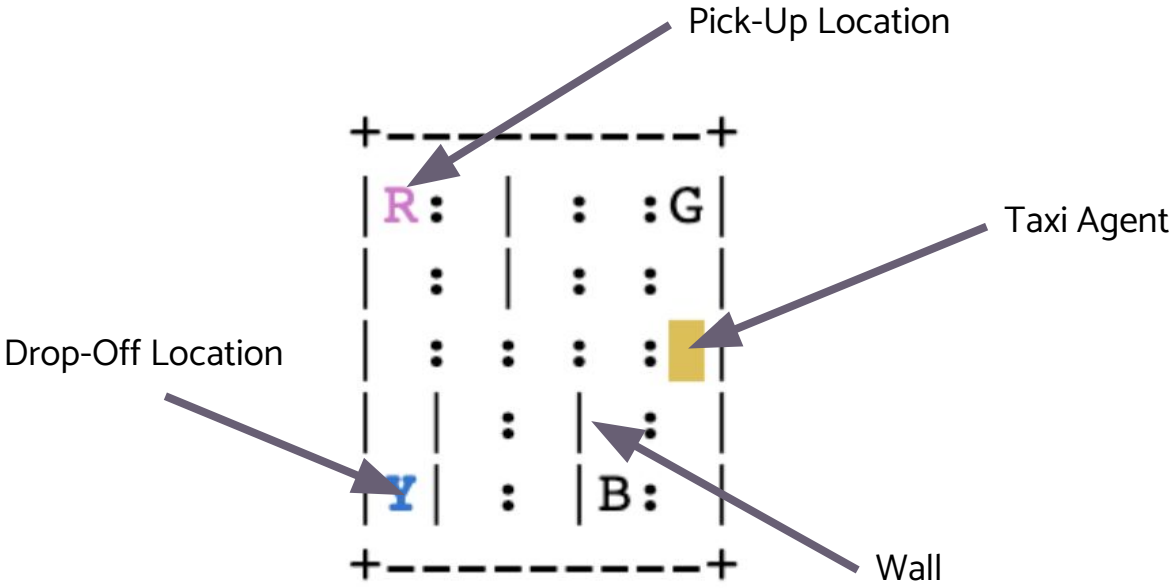
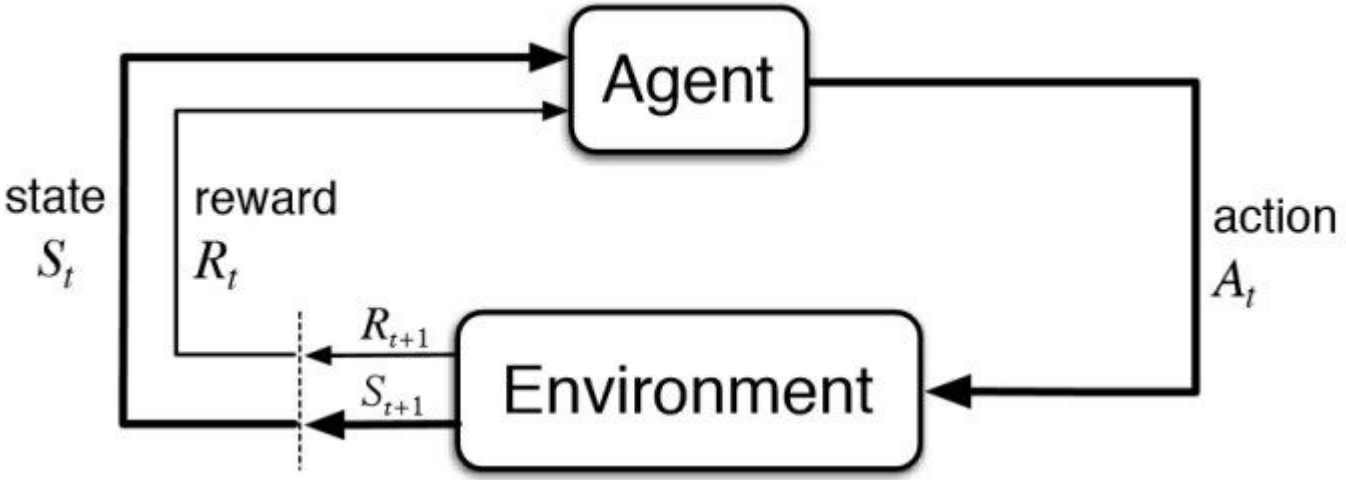


Figure 1: Taxi World

Motivating Example: Taxi World

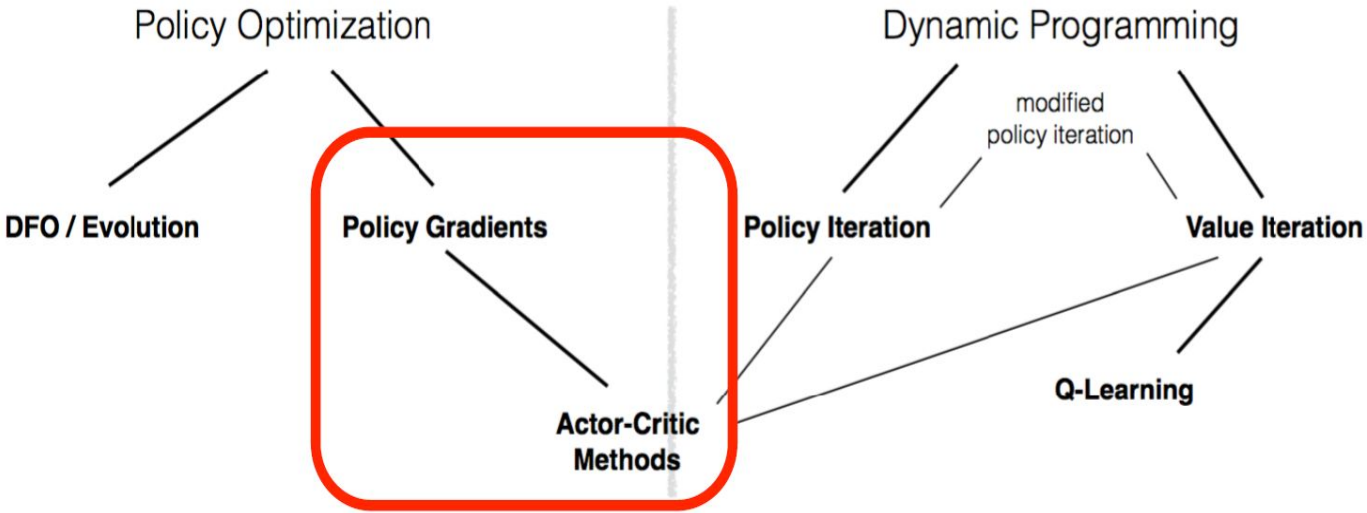
- How will Reinforcement Learning solve this problem?
  - Reinforcement Learning is about training a model (Agent) that continually observes the State of its environment, and then takes the 'best' Actions to maximize Reward.
  - Taxi Agent → Observes World → Takes 'best' Actions given its observation, and learns as it goes along.



Source: www.kdnuggets.com

Deep Reinforcement Learning

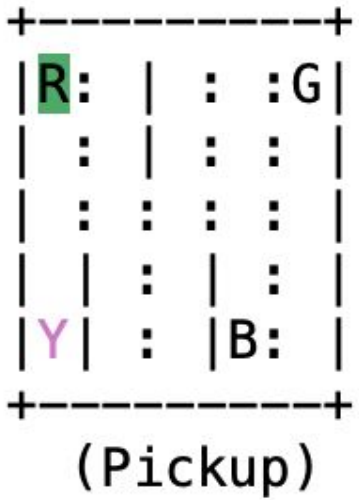
- **Introduction to Deep Reinforcement Learning**
  - Finding optimal policies using policy gradients and gradient ascent
  - Asynchronous Advantage Actor-Critic (A3C)



Defining a Policy

- Conceptually, a policy is a classifier that chooses an action based on the input observations
- Policies can be deterministic (a specific state maps to a specific action) or stochastic (a state maps probabilities to actions)
- Naive classifiers, such as a logistic linear boundary, can perform well in binary action spaces

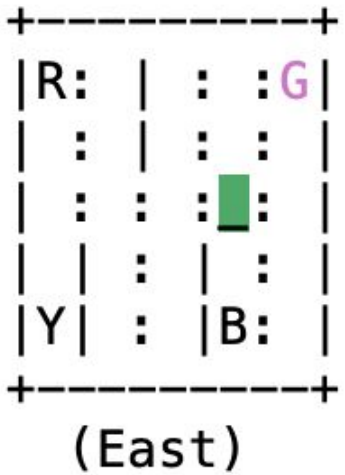
The agent has reached the location of the passenger, **R**. The optimal policy should output a pickup action given this state to actually pick up the passenger



Defining a Policy

- Conceptually, a policy is a classifier that chooses an action based on the input observations
- Policies can be deterministic (a specific state maps to a specific action) or stochastic (a state maps probabilities to actions)
- Naive classifiers, such as a logistic linear boundary, can perform well in binary action spaces

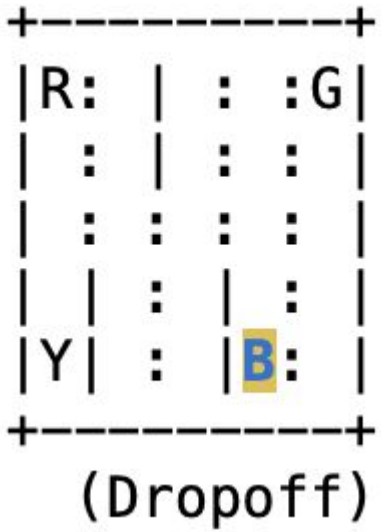
The agent has picked up the passenger, so the optimal policy would output an action to move towards pink drop-off area **G**. The optimal action is to move east or north



Defining a Policy

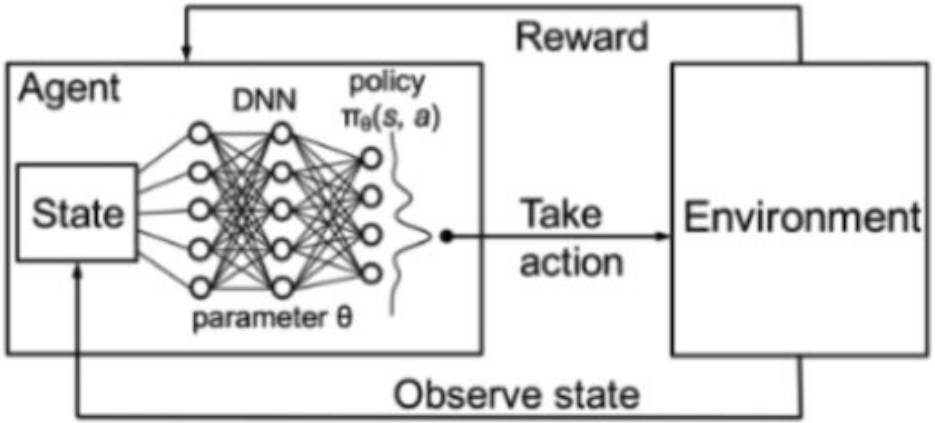
- Conceptually, a policy is a classifier that chooses an action based on the input observations
- Policies can be deterministic (a specific state maps to a specific action) or stochastic (a state maps probabilities to actions)
- Naive classifiers, such as a logistic linear boundary, can perform well in binary action spaces

The agent has reached **B**, the dropoff point, so the policy should classify to a dropoff action given this state of the world



Advanced Policies: Deep RL

- In practice, a policy is generally implemented as a classification neural network
- Given continuous observations as input, a classification neural network is used to output logits (log-odds), which are then processed into probabilities that represent the chance that we choose an Action
- Neural networks make it easy to compute the gradient and update the parameters using backpropogation



Policy Gradients

- **Main Idea:**
  - Compute the gradient of the policy parameters with respect to Expected Reward
  - Use gradient ascent to iteratively improve the policy parameters for the task
- Policy gradients optimize by increasing the probability of selecting actions that provide higher return and decreasing the probability of selecting actions that provide lower return
- See full derivation of the policy gradient ascent equation in the notes

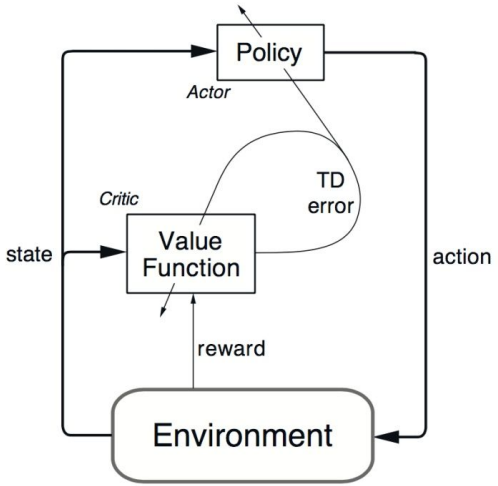
$$\nabla_\theta E_\theta[R\tau] = \nabla_\theta \int_\tau P(\tau|\theta)R(\tau)$$

$$J(\theta) = \nabla_\theta E_\theta[R\tau] = E_\theta[\sum_{t=1} \nabla_\theta \log P(s_{t+1}|s_t, a_t)R(\tau)]$$

$$\theta_{t+1} = \theta_t + \alpha * (E_\theta[\sum_{t=1}^{T} \nabla_\theta \log P(s_{t+1}|s_t, a_t)R(\tau)])$$

Actor Critic Methods: A3C (Slide 1)

- Asynchronous Advantage Actor-Critic
  - More advanced, state-of-the-art DeepRL technique
  - Google's DeepMind in 2016: https://arxiv.org/pdf/1602.01783.pdf



Actor Critic Methods: A3C (Slide 2)

- Asynchronous:
  - Naive:
    - Single Agent training against a single environment to build a single model
  - A3C:
    - Multiple asynchronous worker Agents, each with local model and trains against a local environment
    - Each checks its performance against the single, global model - update if local > global
    - Parallelizable! Encourages broader exploration
- Advantage: $$A(s,a) = Q(s,a) - V^{\pi}(s)$$
  - Defined as Q-Value minus Policy-determined Value at a given state
  - Instead of learning a Value function over States, learns an Advantage function over State-Action pairs
  - Encourages more effort investigating 'promising' areas, where actual rewards > expected rewards

- Actor-Critic:
  - Actor: decides which Action the agent should take
  - Critic: estimate how 'good' Actor's Action was
  - Both the Actor and the Critic train off each other

The following slides are optional supplementary material that delve into more advanced RL topics and terminology

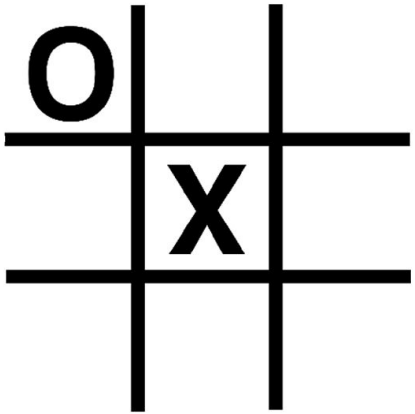
Appendix A: MDPs

Part I: MDPs

- **Markov Decision Processes (MDPs)**
  - **Markov Property**
  - States
  - Actions
  - Q-States
  - Transition Functions
  - Reward Functions
  - Discount Factors
- MDPs must obey the **Markov Property**, also known as the memoryless property. Essentially, all Transitions, Actions, and Rewards must be based on only the current and immediate next States not on past states.

Part I: States

- A **State** captures the the current status of each element in the environment.
- The set of all possible States of the environment is known as an MDP's State Space.
- **Example: Tic Tac Toe**
  - Location of X's
  - Location of O's
  - Boolean Variable (True if Agent's Turn)
- **Example: Chess**
  - Location of Black Pieces
  - Location of White Pieces
  - Boolean Variable (True if Agent's Turn)

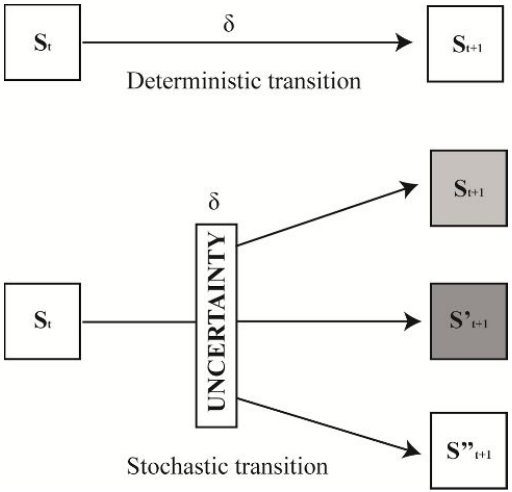


Part I: Actions (Slide 1)

- Given a state, the Agent will have a set of possible **Actions** available to it.
- An **Action** is a ...
  - controlled choice about what future outcome the Agent desires ...
  - based on the Agent's observation of the current State.
- The set of **Actions** is known as the **Action** Space.
- Two types: Discrete & Continuous
- **Example: Chess (Discrete Action Space)**
  - Finite set of actions to choose from.
- **Example: Self-Driving Car (Continuous Action Space)**
  - Infinite set of actions to choose from.
- MDPs → Discrete Action Spaces

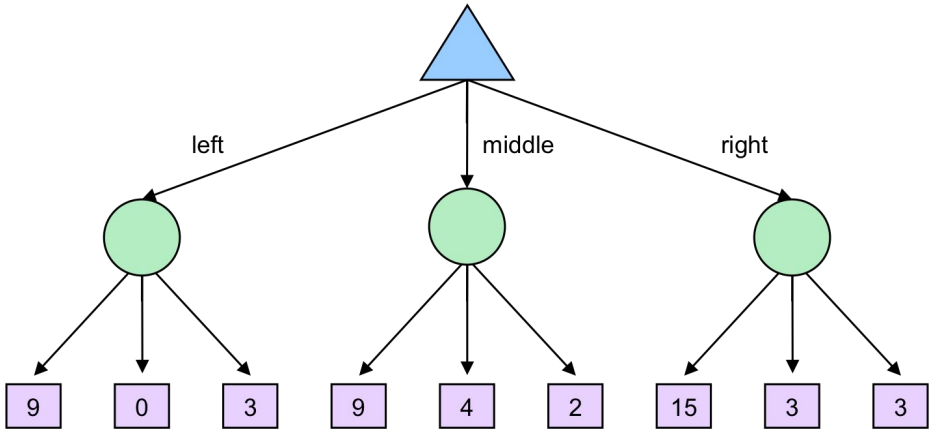
Part I: Actions (Slide 2)

- An MDP allows for modelling environments with **uncertainty**.
  - An Agent's Actions are not the only factor that determine the next State.
  - In stochastic environments, it may be the case that there are stochastic Transitions from State to State.
- **Example: Chess (Deterministic Transitions)**
  - As long as the move is valid, a player may indicate an intention to move their pawn forward (Action) and then know with absolute certainty that the pawn has reached its new square in the subsequent State.
- **Example: Blackjack (Stochastic)**
  - An Agent's decision to take the "Hit" Action and receive a new card from the dealer will send them to one State randomly determined out of a number of States based on which exact card has been dealt. We will address this idea of uncertainty again when we define the Transition function.



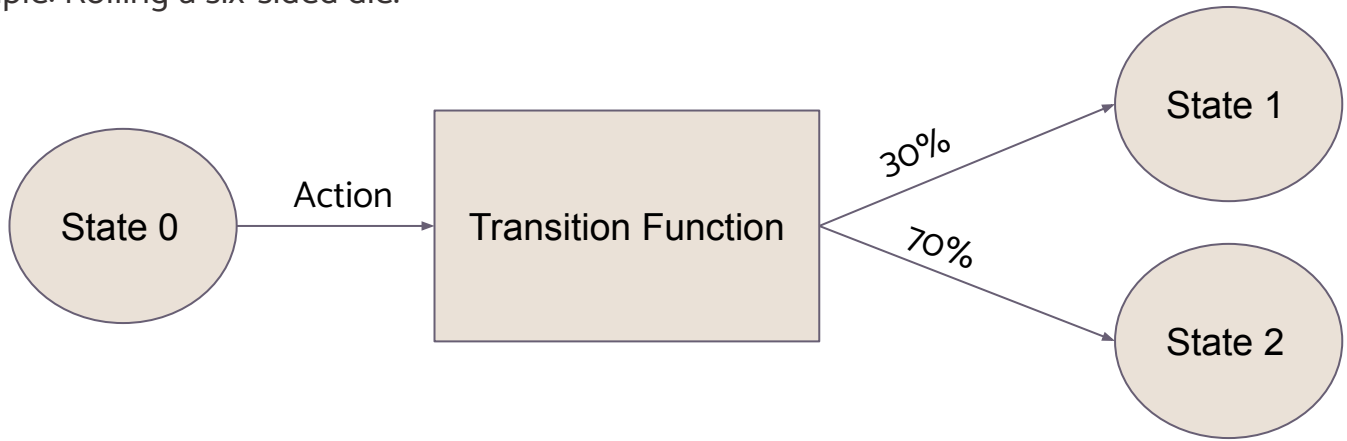
Part I: Q-States

- **Q-States** are a combination of a State and an Action taken from the Action Space available at that State.
  - Q-State → State & Action
- Q-States are intermediate States that are useful in representing the process of the Agent's decision making.
- More explicitly, a Q-State is the State the environment is in after an Agent has decided which Action they will take, but before the Agent has actually executed that Action.



Part I: Transition Function

- Uncertainty and Randomness associated with Actions can be formally expressed using a **Transition function**.
- A Transition function, takes the form T(s, a, s'),
- T(s, a, s') → the probability of transitioning to State s' given that the agent was previously in State s and took Action a.
- For deterministic games each State-Action pair can only result in exactly one next State s'.
  - T(s, a, s') = 1 for exactly one s' and T(s, a, s'') = 0 for all s'' ≠ s'.
- For stochastic games, the Transition function is often a probability distribution over multiple destination States s'. Example: Rolling a six-sided die.



Part I: Reward Function (Slide 1)

- At the same time that a Transition is made, the Agent will receive some **Reward** based on its previous State, Action taken, and next State; the function that determines the value of this Reward is the Reward function.
- **Example: Pacman**
  - A simple Reward Function will give the Agent a +1 Reward each time the Pacman eats a food pellet, and a +50 Reward each time the Pacman eats a vulnerable Ghost.

Part I: Reward Function (Slide 2)

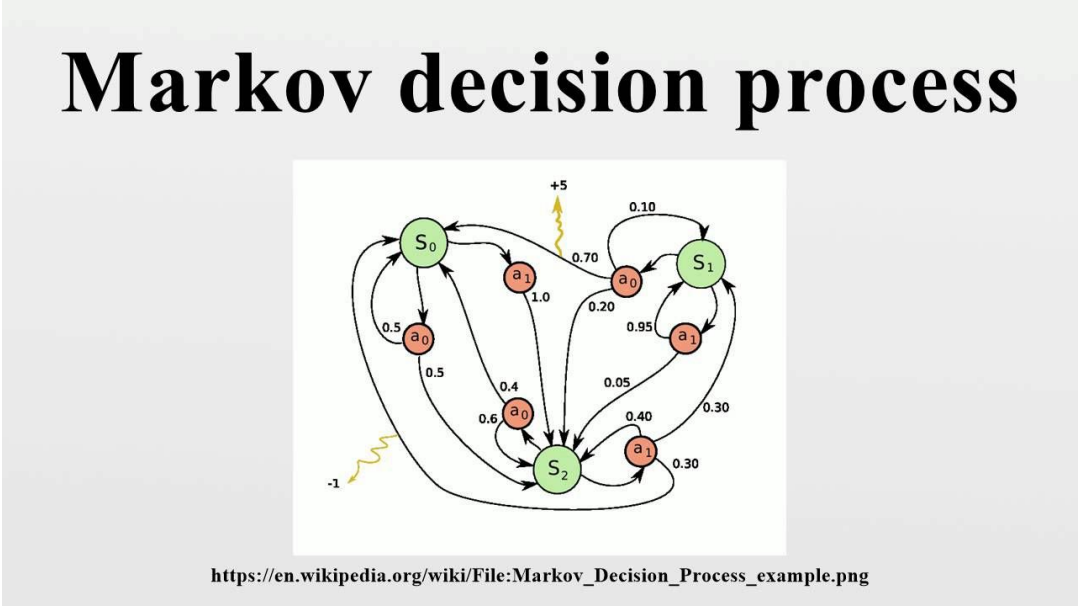
- **More Sophisticated Reward Functions**
  - One way to incentivize an Agent to solve games quicker is to introduce what is (counterintuitively) known as a **Living Reward.**
  - Each timestep that the Agent is alive, it receives a small penalty → motivates the Agent to find shorter solutions. This ends the game sooner, with less accumulated penalty!
  - Reward functions can also be designed to incorporate **risk-averse** or **risk-seeking** behaviors.
    - Artificially shrink down or blow up Rewards at either extreme.
    - A risk-averse Pacman might exhibit a strong fear of Ghosts and miss out on food pellets that are close to enemies
    - A risk-seeking Pacman might attempt a feat of bravery, dashing in and out of a Ghost's path to secure a single pellet
    - Utility functions are a common mechanism to specify risk tolerance

Part I: Discount Factor

- A **discount factor**, typically notated as the Greek letter `gamma', specifies a multiplicative "discount" applied towards future rewards.
- **Example:**
  - Agent is contemplating taking a certain Action with accompanying Reward +100, and gamma =0.1
  - If that Action is taken on the very next timestep, the Agent enjoys the full +100 Reward.
  - If, the Agent decides to wait a timestep before taking the Action on the second timestep, then the `current value' of the Reward is only 0.1 * 100 = 10. And so on!
- **Purpose:**
  - Discounting Indicates a preference for earlier Reward over later Reward.
  - Intuitive from a real-world perspective: 10,000 in hand today is more useful than 10,000 received fifteen years from now.
  - The sooner the Agent incurs a Reward, the less discounted the Reward will be, and so the greater the total Reward the agent will receive.

Part I: MDPs

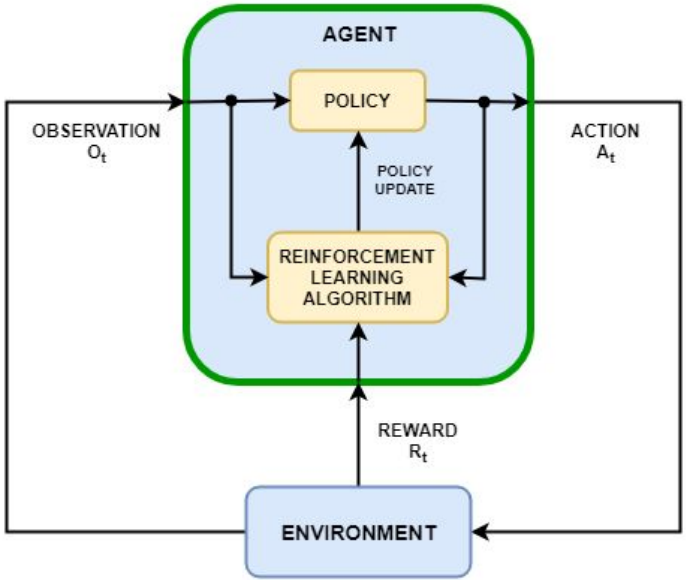
- **Markov Decision Processes (MDPs)**
  - States
  - Actions
  - Q-States
  - Transition Functions
  - Reward Functions
  - Discount Factors
- **Next: Solving MDPs**



Appendix B: Solving MDPs

Part 2: Solving MDPs

- **Offline Learning: Techniques for Solving MDPs (Introduction to Policy)**
  - Policy
  - Values and Q-Values
  - Bellman Equation
  - Value Iteration and Policy Extraction
  - Policy Iteration (Evaluation and Improvement)



Part 2: Policy

- How does the Agent 'behave'? ⇒ **Policy**.
- A **Policy** is a mapping from States to Actions, which can be used by the Agent to select which Action to take from its current State.
- In environments with discrete State and Action Spaces, a Policy can essentially function as a `lookup table', often implemented as a dictionary.
- A complete Policy maps each state to a single action, and can be represented as **π(s)**. The optimal Policy function, $\pi^*(s)$, is what we would like to learn.
- Our goal is to now build up a formal framework that will allow the agents we write to learn that optimal Policy $\pi^*(s)$.

Part 2: Values and Q-Values

Our goal is to maximize the expected total Reward over the lifetime of the agent. But **how can our Agent in the present deal with future Rewards that are uncertain and unrealized?**

Start by defining two quantities: **V*(s)** and **Q*(s, a).**

V*(s) is the expected value of the Reward that an Agent starting from state s and behaving optimally afterwards will receive over its entire lifetime.

Q*(s, a) is the expected value of the Reward that an Agent starting from Q-State (s, a) and behaving optimally afterwards will receive over its entire lifetime. That is, the Agent begins in State s, takes an Action a, and then proceeds optimally after reaching the next State s'.

The astute reader will recognize that there is an inherent recursion at play between the Values and Q-Values. The Value for a State 0 should conceivably be impacted by the Values of the States 1, ... ,n that follow State 0.

More on the next slides!

Part 2: Bellman Equations (Slide 1)

- The **Bellman Equation** formulates this recursion mathematically, representing the optimal Values and Q-Values of each State as a function of other States' optimal Values and Q-Values.
- We will present the equation here, and then derive the Bellman Equation through logical reasoning.
-

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Part 2: Bellman Equations (Slide 2)

- **Consider the following thought experiment:** You are the Agent in a world modeled by an MDP, and you know the optimal value V*(s) for all States of the MDP. Currently, you are in a State o. **How would you choose which Action to take?**
- Intuitively, you would want to choose the Action that moves yourself to the State with the highest value of V*(s).
- This notion of maximizing over Actions suggests the following structure so far:
-

$$V^*(s) = \max_{a \in actions} (...)$$

Part 2: Bellman Equations (Slide 3)

- However, in an uncertain world, there are multiple resultant States s' that you may find yourself in after taking Action a.
- You need to consider each of these States s', as well as the probabilities of reaching those States after having taken the Action a.
- Looping over all s' suggests this:
-

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')(...)$$

Part 2: Bellman Equations (Slide 4)

- Recall that each Transition may be accompanied by a Reward. Since your goal is to find the Action that maximizes expected Reward, it is obvious that you must include the Reward function in your equation:
-

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + ...]$$

Part 2: Bellman Equations (Slide 5)

- Now, recall also that the Reward function only encapsulates the Reward earned while transitioning from s to s' via a. If the game ends immediately after that Transition, then that is all you would need to consider. However, if the game continues, you need some way of representing the expected future Reward from the new State s' you have just arrived in.
- This is exactly the definition of V*(s')! This insight yields the following:
- $$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + (...) \times V^*(s')]$$

Part 2: Bellman Equations (Slide 6)

- Finally, recall that future Rewards are typically less beneficial than sooner Rewards of equal magnitude. As a result, you must incorporate the discount factor gamma. The result is the complete **Bellman Equation:**
- $$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

Part 2: Bellman Equations (Slide 7)

- After going through this derivation, the equation for Q*(s, a) should be clear. Instead of maximizing over all Actions as in the previous equation, we are explicitly given the first Action a. The rest of the equation follows cleanly:
- $$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \times V^*(s')]$$

Part 2: Value Iteration (Slide 1)

- Now we will introduce **Value Iteration**, a technique to actually solve for these optimal values.

- The idea of Value Iteration is to iteratively improve our estimates of the Values of each State until we have learned the true optimal Values and satisfied the Bellman Equation. The key subproblem that we will use to solve for V*(s) is V_k(s). V_k(s) is the expected value of the Reward that an optimally behaving Agent starting in State s will receive over the next k timesteps (as opposed to over the Agent's entire lifetime).

- When the numerical values for V_k(s) = V_{k+1}(s) for all s, we know that the Values have converged. Though we will not show it here, it is possible to prove that the converged Values must be the optimal ones: V_k(s) = V*(s). Thus, once Value Iteration does not change any Values from one iteration to the next, we have achieved our final Values.

Part 2: Value Iteration (Slide 2)

The Algorithm:

1. For all states, initialize $V_0(s) = 0$.
2. Repeat the following while $V_k(s) \neq V_{k+1}(s)$ for any s

$$V_{k+1}(s) \leftarrow \max_{a \in actions} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma * V_k(s')]$$

Part 2: Policy Extraction

- Now that we have the optimal Values of each State, we want to extract a policy or do **Policy Extraction**
- Bellman Equation written in terms of Q-Values:
  - $V^*(s) = \max_a Q^*(s, a)$

- Now, we only care about what that optimal **Action** is.
- Replace the max function with an argmax and substitute for Q*(s, a):
- $$\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

- Now, we have an equation for the **Policy**!

Part 2: Policy Iteration (Slide 1)

- Value Iteration with Policy Extraction works, but it can be very slow!
- Notice that while the values of V*(s) converge only gradually, the optimal Policy does not care about the numerical values.
- The optimal Policy is only determined by the State-to-State comparisons.
- **Policies Iteration** will converge faster than all the Values converge.
- We should perform **Policy Iteration** along with Value Iteration.
- Algorithm on next slide!

Part 2: Policy Iteration (Slide 2)

Here are the basic steps of the Policy Iteration algorithm:

1. Initialize $\pi_0(s)$ to an arbitrary Policy. Recall that a Policy is simply a mapping from states and actions.
2. Repeat the following while $\pi_k(s) \neq \pi_{k+1}(s)$ for any s:

   Use Value Iteration to find the Value of each State while following Policy π_k (defined as $V^{\pi_k}(s)$, as opposed to the optimal Policy $V^*(s)$), by repeating while $V_j^{\pi_k}(s) \neq V_{j+1}^{\pi_k}(s)$ for any s:

$$V_{j+1}^{\pi_k}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma \times V_j^{\pi_k}(s')]$$

   Update the Policy using Policy Extraction:

$$\pi_{k+1}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \times V^{\pi_k}(s')]$$

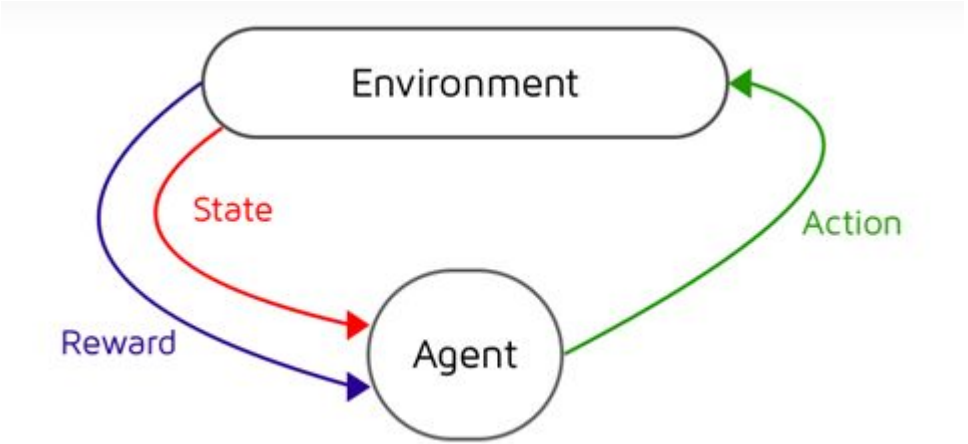
Part 2: Solving MDPs

- **Offline Learning: Techniques for Solving MDPs (Introduction to Policy)**
  - Policy
  - Values and Q-Values
  - Bellman Equation
  - Value Iteration and Policy Extraction
  - Policy Iteration (Evaluation and Improvement)
- **Next: Reinforcement Learning**

Appendix C: RL Fundamentals Other Than Policy Gradients and Updates

Part 3: Reinforcement Learning

- **Reinforcement Learning**
  - Offline Learning vs Online Learning
  - Online Learning: Model-Based Learning
  - Online Learning: Model-Free Learning
  - Model-Free Learning
    - Passive RL
      - Direct Evaluation
      - Temporal Difference Learning
    - Active RL
      - Q-Learning



Part 3: Offline & Online Learning

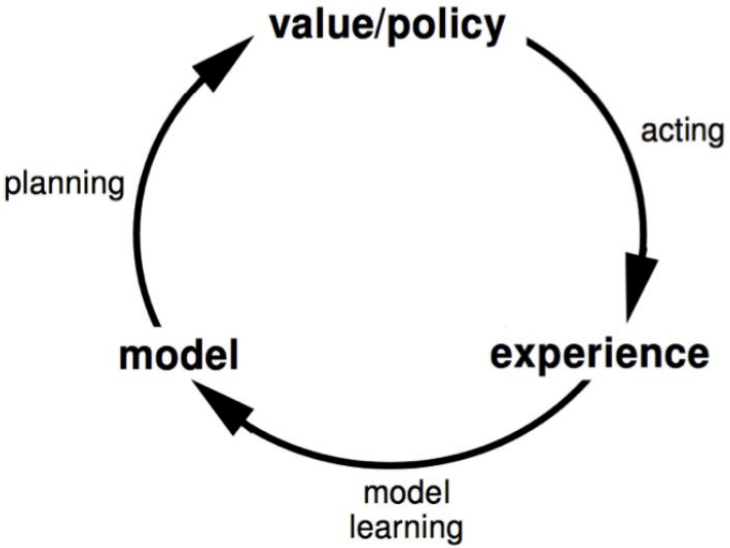
- One question we have not addressed yet involves challenging the fundamental assumption made throughout the first part of this note. Namely, what if we do not know the underlying MDP of the environment?

- Everything we have done so far assumes that the the Agent has complete knowledge of both the Transition function and the Reward function. In such a situation, we can perform Offline Planning, because the Agent can learn an optimal Policy without ever having to explore the real world.

- However, this big assumption about ground-truth MDP knowledge is not reasonable in the real world and in many environments. In situations where the Agent does not know the Transition or Reward functions, the Agent must explore the environment and learn the environment's intrinsics. This is called Online Planning.

Part 3: Online Learning: Model-Based

- **Exploration** as a way to approximate the underlying MDP
  - Keep track of all of the Transitions and Rewards during exploration
  - Calculate probabilities and build an approximation of the actual Transition and Reward functions
  - Building a model, hence Model-Based



Part 3: Online Learning: Model-Free

- Agent **does not** try to learn the Transition and Reward functions
- Instead, the Agent learns the optimal Policy directly,
  - Usually learns the Values or Q-Values of each State
  - Then, performs Policy Extraction
- **Model-Free** methods are more complex - the rest of this section will be Model Free
  - Evaluate specific policies:
    - Direct Evaluation
    - Temporal Difference Learning
  - Learn the optimal Policy:
    - Q-Learning.

Part 3: Model-Free Algorithm #1: Direct Evaluation

- Simplest Model-Free method:
  - Following the given policy for many episodes
  - Record total Reward received after leaving each state, along with counts of number of times visited
  - Approximate value as total Reward divided by times visited
- Number of flaws:
  - Discards all information about the trajectory of States and Actions - only keeps the end results
  - Simple average, so vulnerable to poor evaluation caused by a single outlier
  - Observation quality is stagnant throughout run

$$V^{\pi}(s) = \frac{\text{total reward achieved from state } s}{\text{times state } s \text{ visited}}$$

Part 3: Model-Free Algorithm #2: TD Learning

- Core concept: the Agent can learn after **each Transition**
  - Maintain a running estimate of each State's optimal Value, collect samples, update stored values
- Algorithm explicitly:
  - Start with an initial Value of 0 for all states
  - Repeat for the number of exploration episodes:
    - While the current state is not terminal:
      - Using the specific policy and from the current state, determine the action
      - Transition from current state to new state using policy-specified action, receiving Reward
      - Construct sample and perform weighted average update
- Update rule
  - Uses a learning rate to guide how much weight to give the new sample
  - Favors newer samples, because the values converge closer to the true values over time
    - Since samples depend on values, newer samples are more accurate and need higher weight

$$\text{sample} = R(s, a, s') + \gamma \times V^{\pi}(s')$$

$$V^{\pi}(s) \leftarrow (1 - \alpha) \times V^{\pi}(s) + \alpha \times \text{sample}$$

Part 3: Model-Free Algorithm #3: Q-Learning

- Q-Learning: allows us to actually learn the optimal policy
  - Learns the optimal Policy by learning the optimal Q-Values at each state
  - Takes samples, incorporates into the running estimates
- Epsilon-Greedy
  - If we were to follow our current Policy at all times, we never explore new States!
  - Solution: With some small probability epsilon, take a random Action. Otherwise, use the best Action as determined by the current Policy.
- Algorithm:
  - Initialize all Q-Values as 0
  - Repeat for the number of exploration episodes:
    - While the current state is not terminal:
      - Determine action using Epsilon-Greedy and transition to new state
      - Construct a sample using the incurred Reward
      - Update Q-Value using this new sample and learning rate alpha

$$\text{sample} = R(s, a, s') + \gamma \times \max_a Q(s', a) \qquad Q(s, a) \leftarrow (1 - \alpha) * Q(s, a) + \alpha * \text{sample}$$

Part 3: Reinforcement Learning

- **Reinforcement Learning**
  - Offline Learning vs Online Learning
  - Online Learning: Model-Based Learning
  - Online Learning: Model-Free Learning
  - Model-Free Learning
    - Passive RL
      - Direct Evaluation
      - Temporal Difference Learning
    - Active RL
      - Q-Learning