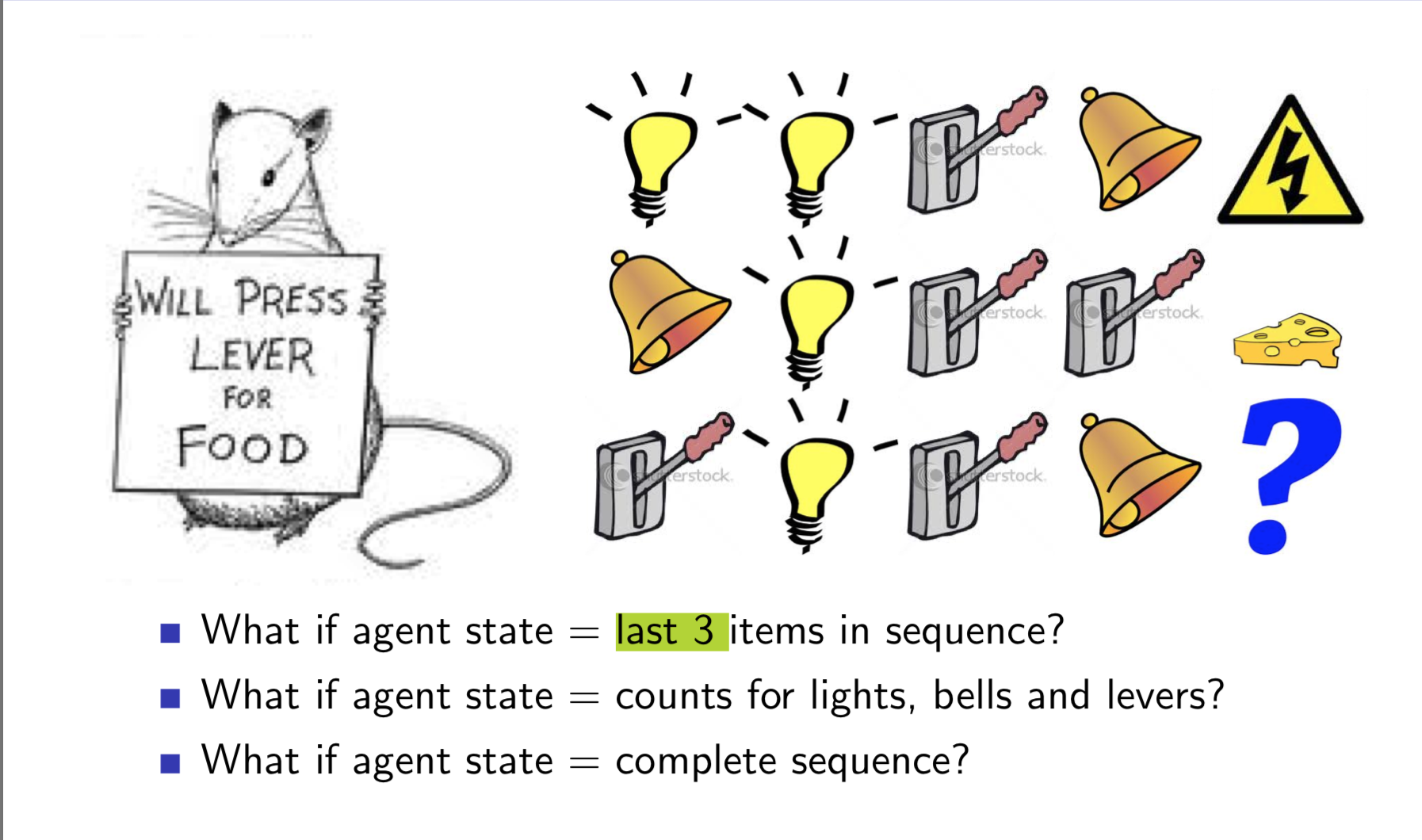
# RL Lecture Takeaways & Important Notes

Lecture 1:



Given the two sequences of historical data, what do we think is going to happen next?

* Well it depends on how we define the agent state. If we define the agent state using the last 3 items in the sequence, we expect to get electrocuted, however, if we use the total number of each observation, we will expect to get cheese as there are two levers, one bell and one light.
* This illustrates how important the state representation is in building an effective RL algorithm.
* Agent must construct its own state representation. Can use the complete state history or take a Bayesian approach and construct *beliefs* about the environment state with a probability distribution over all possible states.
* You can also use an RNN for state representation. Just takes a linear transformation of old state to a new state which takes into account some observations and squashes it through a non-linear activation function.

Fundamental distinction in RL is model-free vs model-based RL agents:

**Model Free:**

* Do not try to explicitly understand (i.e. model) the environment. We just use a policy and/or value function to map states to actions guided by experience with the environment

**Model Based:**

* Uses a model of how the environment works in conjunction with a policy and/or value function

**Prediction Problem vs Control problem:**

The notion of the prediction problem is simple: given our current estimate of the value function and a policy, how well will the policy do? (Policy evaluation)

The control problem is essentially just the problem of finding the optimal policy.(policy improvement)

**Temporal Difference Algorithms** suggest there is a time delay between the reward r which results from taking an action a. More on these in lecture 4.

## Q-Learning Notes

Source [here](http://outlace.com/rlpart3.html)

* It is infeasible to store state-action space in a tabular format for the vast majority of interesting problems. For instance, in computer vision problems, there would be a state for the permutation of every possible value of a pixel for all pixels of the image, i.e. if one pixel changes, it necessitates us to represent that as a new state.
  + This doesn’t make a lot of sense though and is wasteful.
  + We can use neural nets to train the program to learn the value for certain *kinds* of states.
  + This is one reason why deep reinforcement learning is allowing us to tackle more interesting problems.

**On-Policy Methods:**

* The value of a state depends on the current policy.
* If there’s a symbol in the update part of the equation, it’s on-policy.
* On policy methods involve updating and improving one policy – the one used to make decisions from state to state throughout the duration of an episode.
* **The policy is generally soft, meaning that all action probabilities from a given state are greater than 0. They gradually shift toward a deterministic optimal policy.**

**Off-Policy Methods:**

* Uses two policies: the target policy, which is the policy learned about and becomes the optimal policy, and the behavioral policy, which is exploratory and used to generate behavior.
* Do not consider the policy when updating the value function.
* The benefit of an off-policy method is that we can follow one policy while learning about another one.
* E.g. **Q-learning**
* Off-policy methods involve learning about and updating one policy until it is an optimal policy and using another policy for decision making and exploring the environment in hope of finding a better policy. The policy used for decision making is called the *behavioral policy* and the one being learned about is referred to as the *target policy.* Since we are learning from data given by the behavioral policy, we say these methods are learning “off the target policy” or just *off-policy.*
* Nearly all off-policy methods utilize **importance sampling, a technique to estimate the expected values under a distribution using samples from a different distribution.**
* If we want to use episodes generated following the behavioral policy to estimate values for the target policy, every action taken under the target policy must also be taken, at least occasionally, under the behavioral policy too. This is the concept of *coverage.*

Neural net takes a state and outputs a probability distribution over all available actions. Using softmax function, we can train this network to learn to optimize what actions should be taken for a given state.

**Catastrophic Forgetting: a problem that arises with gradient descent based learning methods**

We can overcome catastrophic forgetting with a method called **experience replay.**

**Temporal Credit Assignment Problem:** is the problem where we need to decide which actions from which states are responsible for a delayed reward. The influence of a reward becomes diluted as more timesteps intervene between the action and receipt of the reward.

Challenging problem: When the agent receives a good reward (or a bad punishment), it's probably not due to the latest action it took alone but because of the series of actions that lead it to this rewarding state. A challenge here arises on how to assign this reward (or punishment) backwards in time across the path the agent took. The credit assignment problem is the problem of telling the agent which sequence of states and actions are responsible for the outcome they have received.

**Dynamic Programming for Planning** can only be utilized when we have full knowledge of the MDP at hand. This is a severe disadvantage as the vast majority of the time the RL problem will not meet this condition.

## Lecture 3 – Dynamic Programming for Planning

Take notes on 1:24:00 (lecture 3)

**What is the difference between synchronous and asynchronous dynamic programming?**

Synchronous DPinvolves backing up all the states at each iteration when updating the value function.

Asynchronous DP **does not necessitate that we sweep through the entire state space on each policy evaluation.** We back up states individually in any order. Since we are not backing up every state with each iteration, there are notable performance improvements. ***We can update states that are most relevant to the problem at hand predicated on the notion that some states may be more important and visited more frequently than others. E.g. we can update only the states the agent interacts with through experience.***

3 ideas of asynchronous DP:

1. In-Place DP:
   1. Only store one copy of the value function for every state in memory
2. Prioritized Sweeping:
   1. In order to understand this, we need to understand the notion of Bellman error. Bellman error is simply the difference between the maximum value we can get from state s and the computed value of being in state, s.
   2. The idea here is to backup the state which had the largest bellman error.
3. Real-Time DP:

Value and policy iteration converge by the contraction mapping theorem.

**What’s the difference between policy iteration and value iteration?**

**Summary:**

*For policy iteration, we commence with a blank policy. We consider each state and greedily select the action that yields the highest expected utility. These greedily selected actions are used to formulate a new policy. If, at any state, the greedily selected action is different than the stored policy, we say the policy is unstable. If the policy is unstable, we update our value function by evaluating the policy. Policy stability indicates policy iteration should be terminated. That is, we will repeat the process of computing a greedy policy using the current value function until it matches the previously computed policy.*

*For value iteration, we update the value function sporadically. That is, unlike policy iteration where we update the value function for all states simultaneously, we update the value function one state at a time when performing value iteration. Just like policy iteration, we compute the expected value of the available actions for a given state. However, we immediately update the value of the state to be the maximum of these computed expected values and update our policy to greedily selection the action that generates that maximum expected value. We continue this process until the maximum change in the value of a state is below a prespecified threshold.*

In order to optimize a policy, we must develop a way to evaluate policies so we can dictate whether or not one policy is better than another. We evaluate policies by looking at their value functions for all states:

Computationally, this means traversing through every state and computing the expected value achieved by following that policy for each state. We continue to iterate over every state and update the value function until the value function converges. In theory, the value function converges with infinite iterations, but in practice we just keep traversing through the state space until the change in the updated value function is sufficiently small.

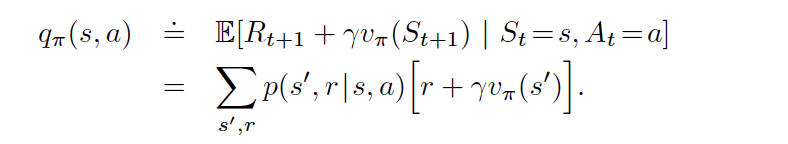
At each state, s, the agent has an array of actions they can take. Each action may transfer the agent to one of many possible successor states. The successor state the agent is transferred to is governed by the transition probability dynamics of the MDP. Thus, the value of a state, s, is expected value of taking each action:

V(s) = sum(action\_prob \* sum( transition\_prob \* (reward + discount\_factor \* expected\_val\_for\_ss)))

Note: we sum over all feasible actions and all successor states for each action

* action\_prob is the probability we take action a under policy pi, i.e. =
* Transition prob is the probability that we end up in a particular successor state as a result of taking action a from state s.

Now that we have a way of determining if one policy is better than another, we can improve any policy by acting greedily (i.e. choosing at each state, s, s.t. the action maximizes



This equation makes sense: it’s saying the state-action value for state, s, is the expected value of the immediate reward from taking that action from that state plus the discounted value of the successor state. Since this is an expected value, it fundamentally boils down to the bottom line, which says:

* By taking action a from state s, we don’t know what our successor state is with absolute certainty
* So, we weight the immediate reward of taking action a plus the discounted value of the successor state by the probability that we will end up in that successor state by taking action a from state s for every successor state that we can end up in

**Policy Evaluation Implementation Notes:**

* By implementing policy evaluation (DP) I have learned that we must consider all of the actions and their corresponding probabilities dictated by the policy when updating the value of each state. I had originally thought we would only consider the transition probabilities of the MDP for the action with the highest probability given the policy and state.

***Policy* *iteration*:**

We can now employ these tactics to iteratively improve an initial policy until it converges to an optimal one. We start off with an initial policy and value function. We begin by considering the action with the highest probability from that state given the current policy. We can refer to this as our old action. Then we compute the expected value for each action from the state we’re in. This is essentially a one-step lookahead in our state “tree”. We then consider the action with the highest expected value from this lookahead and do this for all states in the MDP. Using this action, we update our current policy in a deterministic fashion. If, for any state, the new action is not equal to our old action, we must do this process again for all states and we say the policy is unstable. If unstable, we update our value function and repeat the process.

It’s called policy iteration because we evaluate a policy, then improve it by greedily taking the action with the highest expected utility for every state in the MDP. We continue to evaluate and improve policies until taking the greedy action at each state corresponds directly to the current policy. At this point, the policy cannot be improved any further and is optimal.

***Value iteration:***

Value iteration considers no policy explicitly whilst updating the value function. As a result, the value function at certain iterations will not correspond to an existing policy. Nevertheless, employing this method will still allow us to converge to the optimal value function for our MDP. In practice, this method is faster than policy iteration.

Value iteration is very simple. We compute the expected value of each action and update the value for the state as the maximum of the action expected values. We then update our policy to deterministically take the action that yields the maximum expected value. We do this for all states and keep track of the largest change between the newly value and the previously compute value for the state. If the difference is sufficiently small for every state, then we return the policy and value function as they have been optimized.

**What’s the difference between the Bellman Expectation Equation and Bellman Optimality Equation?**

**Bellman Optimality Equations:**

Bellman optimality equation says that the optimal state-value function is the value function which obeys the policy that extracts the most value out of all states. The optimal state-action value function is the action-value function which results from a policy that selects actions from each state which maximize the total return. Solving for q\* is the problem we are trying to solve.

We stop computation once these equations are satisfied.

There’s a max over our actions.

**Bellman Expectation Equations:**

There’s an expectation over our actions:

**Principle of Optimality:**

Based on the notion that an optimal policy can be divided into two parts:

1. An optimal first action A\*
2. An optimal policy from the successor state S’

This principle states: a policy results in the optimal value from s iff for all reachable states from s, the policy achieves the optimal value from the successor state.

How do we know that value iteration converges to v∗? Or that iterative policy evaluation converges to vπ? And therefore that policy iteration converges to v∗?  
Is the solution unique? How fast do these algorithms converge?  
These questions are resolved by contraction mapping theorem.

**Contraction Mapping Theorem:**

**GPI (Generalized Policy Iteration)** is the notion of solving the optimal value function for an MDP by iteratively evaluating and improving a policy. This term exists to unify several variations in DP algorithms and other methods. For instance, policy iteration necessitates that we update the value function as we iterate over each state, whereas

iteration Synchronous DP involves

## Lecture 4 – Model Free RL

### Monte-Carlo Policy Evaluation:

The idea is simple: You run episodes, look at the returns you’ve attained at the end of each episode and use them to update your estimate of the mean value for each state that you visit.

**What is the difference between first-visit MC and every-visit MC?**

Both MC methods involve estimating state values or state-action values from experience, a sequence of states, actions and rewards.

First-visit MC updates the value of a state based on the average expected return the agent will realize following policy pi from the first time they visit that state.

In contrast, every-visit MC updates the value of a state using the average expected return for the policy each time the state is visited.

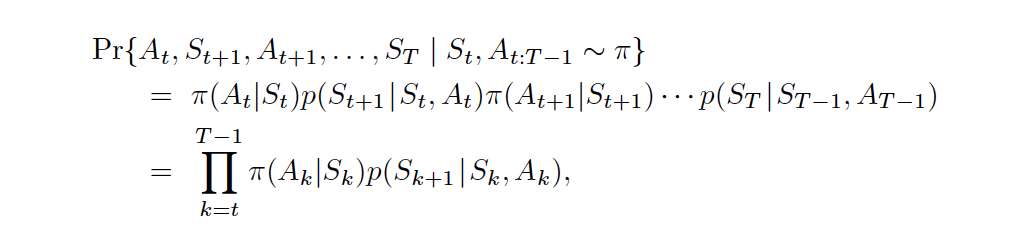
Note: quantifies the value of following policy from state s.

**What is the importance sampling ratio?**

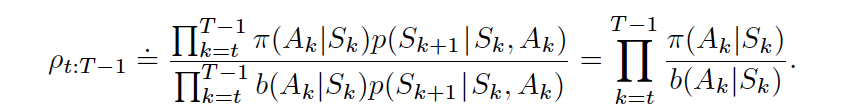
In order to understand the importance sampling ratio, we must first comprehend importance sampling. *Importance sampling* is a method in which we use samples from one distribution, D, to estimate the expected value of another distribution, D’.

In the context of RL, we are taking sample episodes (which are fundamentally sequences of states, actions and rewards) governed by the behavioral policy, b, and using those samples to estimate the expected value of the target policy, .

Each sample episode has a return. It doesn’t make sense to treat all of the returns equally. That is, we cannot simply average the sample returns generated from the behavioral policy to compute the expected return under the target policy. However, it does make sense to compute the expected return by weighting each sample return by the probability of that episode trajectory occurring under the behavioral and target policies. The probability of these trajectories occurring under the target policy over the behavioral policy is the *importance sampling ratio –* it dictates how important each sample is to computing the expected return. Starting at a state, St, the probability of the subsequent state-action trajectory occurring under any policy is:



The relative probability of a sample trajectory under the behavioral and target policies (the importance sampling ratio is):



For example, an importance sampling ratio of 5 indicates the episode is five times as likely to occur under the target policy than the behavioral policy.

Importance Sampling: [Chapter on Importance Sampling](http://statweb.stanford.edu/~owen/mc/Ch-var-is.pdf)

**Why have two different policies for off-policy methods?**

### Temporal-Difference Learning:

Unlike MCPE, this method breaks up the episode and uses incomplete returns.

TD exploits the Markov Property by implicitly building and solving the MDP structure of the observed sequence of actions, rewards and state transitions.

Bootstrapping concept (fundamental idea underpinning TD Learning):

Online updates means we immediately update our value function after leaving a state.

Offline updates means we wait until the end of an episode to update the value function.

TD( gives you a combination of the monte-carlo and TD(0) methods for propagating information between states so as to update the value function. The value of lambda dictates where on the spectrum it will fall, for instance lambda = 0 is the TD(0) algorithm and lambda = 1 is the monte-carlo algorithm.

There are two implementations for TD(

1. Forward View
   1. Way of taking a weighted average over all n-step returns
2. Backward View
   1. Accumulate eligibility traces which can be very efficiently updated and are indicative of how much credit should be assigned to each state

What is the difference between sampling complete returns (Monte Carlo) and partial returns?

DO THIS AFTER LECTURE 5: <https://github.com/dennybritz/reinforcement-learning/tree/master/MC/>