David Silver Reinforcement Learning Course

These are my personal notes for David Silver's RL Course. These aren't really refined/edited in any way (not meant to be a complete tutorial and not sure how coherent the thoughts are), they were just my way of keeping up with the course. It might be best to follow along with each of the videos while reading these notes. Hopefully they can be helpful for you!

Lecture 1 - Introduction to RL

- **Reinforcement learning** is trying to understand the optimal way of making decisions.
- We learn by interacting with our environment.
- **Machine learning** as a whole is made up of supervised learning, unsupervised learning, and reinforcement learning.
    - In RL, there is no supervisor or label, just a **reward signal**. Nobody is telling us the correct action to take, it's just a trial and error paradigm. The reward signal just tells you "that was good" or "that was bad", but it doesn't say what the best action is.
    - In RL, the feedback is delayed and not instantaneous, whereas if you think of CNNs, you can immediately compute some sort of loss by comparing output of network to groundtruth. In RL, you might not know whether a decision was good or bad until a set of time steps later.
    - Unique thing about RL is that the agent's actions affect the subsequent data that it receives.
- Rewards are fundamental in RL
    - A **reward R** is a scalar feedback signal that indicates how well the agent is doing at time step t. The agent wants to maximize this cumulative reward.
    - **Reward hypothesis**: All goals can be described by the maximization of expected cumulative reward.
    - Goal of RL is to select actions that maximize total future reward.
- Actions may have long term consequences and rewards might be delayed. At every time step, the agent gets an input in the form of its observation of the world and also gets an input of a reward signal telling the agent how well it is currently doing. It will then take some action A, which is the "output"
- **History** is the stream/sequence of observations, actions, and rewards.
    - Basically, all of the observable variables up till time t
- Our job in RL is to build an algorithm that maps the history to the next action that we decide to take.
- The environment, on the other hand, looks at what actions the agent takes and also looks at the history, and selects different observations and rewards.

- However, since history can be so large, state is the more common measure of the information used to determine what happens next. It seeks to encapsulate/summarize all the information that history has, and use this info to determine what happens next.
- **State** is a function of history. Aka S = f(H)
    - A valid definition of state would be to only look at the last observation in the history.
    - **Environment state** is the information used by the environment to determine what it is going to do next. This state determines what observations and rewards the environment outputs on the next time step.
        - Important note is that the environment state isn't something that is visible to the agent.
        - Another way of putting it is that the agent doesn't see the *state* of the environment, but it does see the *outputs* of the environment, which are the observations and rewards.
    - **Agent state** is the set of numbers that lives within our algorithm. It summarizes what has happened to the agent so far, and uses those numbers to pick the next action.
        - Another way to put it is that the agent state is the information the agent uses to pick the next action.
    - **Information state (Markov state)** contains all useful information from the history.
        - A state is Markov IF the probability of the next state (conditioned on the state that you are in) = probability of the next state, if you showed all of the previous states to the system.
        - If a state is Markov, that means that the future is independent of the past given the present.
            - If you have a state S and its Markov, you can pretty much throw away the rest of the history H because it doesn't give you any *more* information about what might possibly happen in the future.
            - Basically, the state is now a sufficient statistic of the future.
            - In the helicopter example, a Markov state would be the current wind speed, current wing speed, etc and since this is all that really matters when determining what future action to take, we can throw away history H because it doesn't matter what the wind speed was 10 minutes ago. It won't affect how we take our next decision.
            - However a non Markov state would be if you only had current position, but not another relevant piece of info like velocity. Therefore, you may need to look back at the history to extrapolate.
- **Fully observable environments** are where the agent gets to completely see the environment state.
    - Therefore, agent state = environment state = information state
    - This is a Markov Decision Process (MDP)
- **Partial observability** is where the agent indirectly observes the environment.

    - Robot with only a camera doesn't have info about its absolute location.
    - In this case, agent state doesn't equal environment state.
    - This is a partially observable Markov decision process (POMDP)
    - Agent must construct its own state representation.
        - Naive approach is to just remember everything and have S = H.
        - Or you can keep beliefs about the environment (Bayesian approach) where the agent keeps a probability distribution over where it thinks it is in the environment.
        - Use a recurrent neural network where you use the agent state at the previous time step, along with the observations of the current time step, to construct a new agent state.
- So, we've described the problem, now let's look at how to solve it. An RL agent includes a policy, a value function and a model.
    - **Policy** is the agent's behaviour function.
        - The way that it goes from state to the action decision.
        - Defines agent's way of behaving at a certain time.
        - Deterministic policy: a = f(s). We want to learn this from experience and have it trained so that the action we take gets the biggest reward.
        - Stochastic policy where we have probabilities of actions conditioned on being in some state.
    - **Value function** is measure of how good each state/action is
        - How much reward we expect to get if we take a particular action.
        - Prediction of future reward.
        - The actual reward signal indicates what's good in the immediate sense while the value function is more indicative of how good it is to be in this position in the long run (prediction of future reward).
            - It does this by taking into account the states that are likely to follow and the corresponding values functions that they have.
        - Most important component of an RL system is efficiently and accurately estimating values.
        - value(policy) = expectation *(reward at different time steps, each multiplied by some discount to say that rewards in certain time steps mean more/less than others)
    - **Model** is the agent's representation of the environment.
        - How the agent thinks the environment works. It predicts what the environment will do next.
        - This is helpful because it allows the agent to think about how the environment will behave given certain courses of action.
        - There is a transition component P which predicts the next state of the agent, given information about its view of the environment. (actual physical dynamics)
        - There is a rewards component that predicts the next (immediate) reward.
- Example of RL: Maze

    - Rewards: -1 per time step
    - Actions: Move N, E, S, W
    - States: Action's location
    - The arrows inside each grid space indicate the policy (aka what the agent would do if it is in that space)
    - Value function (for a maze example) shows number of -(# steps it will take to reach goal) for every location.
    - Model function is basically the agent trying to figure out a map of what the environment/maze looks like. Will put a -1 on every grid that it has seen because that characterizes the immediate reward from each state.
- Types of RL Agents
    - Value Based: No policy (implicit), but there is a value function
    - Policy Based: Policy, but no value function.
    - Actor Critic: Policy and value function
    - Model Free: Basically the agent doesn't try to understand the environment (aka we don't build the dynamics of how the helicopter moves)
    - Model Based: Has everything (Policy and/or Value and Model)
- Two Problems in sequential decision making
    - Reinforcement Learning: The environment is initially unknown.
        - Agent doesn't know how the wind blows, what obstacles there are, etc.
        - The agent will interact with the environment and slowly improve its policy.
    - Planning: In this case, a model of the environment is known and given
        - We'll give the differential equations describing wind for example.
        - The agent uses this given environment model, performs computations, and thus improves its policy.
- **Exploration**: Act of finding more information about the environment
- **Exploitation**: Act of exploiting known information to maximize the reward

.

Lecture 2 - Markov Decision Processes

- Markov decision processes formally describe an environment for RL
- The environment is fully observable in this process
    - Even partially observable problems cans be converted into MDPs
- A **transition matrix P** defines the transition probabilities from all states s to all successor states s'.
- A **Markov Process** is a random process, or a sequence of random states S1, S2, …. All with the Markov property.
    - It can be represented as a tuple (S,P) where S is the finite set of states and P is the state transition probability matrix.
- The part where the "random" term comes in is where you pick a sample of sets where there's going to be different chain of events that happen because there's a probability distribution of going from one state to any one of a couple possible next states.

- A **Markov Reward Process** is a tuple (S,P,R,y) where S and P are the same as before, R is the reward function just based on the immediate reward from the current state, and y is the discount factor.
    - Basically calculates the reward achieved after going through a certain sequence of states.
    - The cumulative reward is the **return G**. Goal of RL is to maximize G. G is the total discounted reward *starting from* time step t until the process terminates.
        - G = sum of reward at each time step * discount factor. Discount factor's purpose is to weigh each reward at a time step differently depending on that time step.
- The value function v(s) is the long term value of the state s.
    - If an agent gets dropped into S1, the value function will calculate the expected return from going from that S1 to when it terminates.
    - To find the value of a certain state, you have to write down all the possible "paths" the state can take from its initialization to its termination. Calculate the expected return for each process and then find the average.
- **Bellman Equation for MRPs**: The value equation that we just saw can be decomposed into two parts, one that analyzes the immediate reward for R at time step (t+1) and another part that focuses on the discounted value of the successor state.
    - $v(s) = E[R_{t+1} + yv(S_{t+1})]$
    - $v = R + yPv$
    - Basically, it takes into account the reward of the next time step and also takes into account the value of being in that next state.
    - These Bellman equations give us an opportunity to be able to solve for v.
- A **Markov Decision Process** is a Markov reward process with decisions.
    - It is a tuple (S,A,P,R,y) where everything is the same except A is a finite set of actions.
    - One change is that the state transition probability matrix is dependent on A.
        - If you move left, you'll have different a different state transition probability than if you moved right.
    - Now instead of probabilities of moving from a state S to a successor state S', there are actual choices.
- Again, a policy is a distribution over actions given states. It basically defines the behaviour of the agent.
- In an MDP, policies depend on just the current state, not the history.
    - Remember, this is a requirement of the characteristic of states being Markov.
- Given some fixed policy in an MDP, the state and reward sequence that you get as a result is a Markov Reward Process.
- Now, because we have a policy in an MDP, there is a way for agents to choose how they behave. Therefore, there isn't a fixed expectation anymore. The value function for a given state depends on the policy.
- The **state-value function** characterizes the expected return from being in state s given that we are following some policy π.

- The **action-value function** $q_\pi(s,a)$characterizes the expected return from being in state s, taking a particular action a, and following some policy π.
- Bellman expectation equations are still true in an MDP.
    - State value functions just make sure to follow a policy π for the value function of the successive state.
    - Action-value functions: Same expect just replace v with q of the successive state you end up in.
- Important note: How you calculate the value function in an MDP and an MRP are different. In and MDP, the reward depends on the action you take, not just the state you're in.
- Now, what we want to do is find the behaviour in an MDP.
- The **optimal state-value function** is the max state-value function over all policies.
- The **optimal action-value function** is the max action-value function over all policies. This is quantity that we really want. If we know that the action that will give us max reward over any policy. The MDP is basically solved when you have q*.
- We also need to figure out the best way to behave, or the **optimal policy**.
- We define an ordering of the policies: π >= π' if v(s) with policy π >= v(s) with policy π'
- **Optimal Policy Theorem**: In any MDP, there is an optimal policy that is better than or equal to any other policy. All optimal policies will achieve the optimal state-value functions and the optimal action-value function.
- The optimal policy is found by maximizing over q*
    - You solve for q*, and then pick the action a that results in the highest value for q*
- **Bellman Optimality Equation for V*** says that $v_*(s) = \max [q_*(s,a)]$
- **Bellman Optimality Equation for Q*** says that $q_*(s,a)$ = R + summation (transition matrix P * optimal value function $v_*(s')$ for each successive state)
- To solve for the Ballman Optimality Equation, there is no closed form solution os you have to do value iteration, policy iteration, Q-learning, or Sarsa

Lecture 3 - Planning by Dynamic Programming

- Dynamic programming allows us to solve complex problems by breaking the problem into subproblems, solving the subproblems, and then combining the solutions.
- Use for problems that have these character.
    - Optimal substructure as in they can be broken into parts and then when you find the solutions to those parts, you've found the solution to the whole problem.
    - Overlapping subproblems. Basically this means that the process of breaking the task into parts is *helpful* and more efficient for us to solve the original problem.
        - Basically these subproblems occur many times and can be cached and reused.
- MDPs can satisfy both of the prior characteristics because the Bellman equation gave us this recursive decomposition and the value functions we create are the part that store and reuse solutions.
- Planning in an MDP can be for control or for prediction

    - Prediction
        - Let's say somebody gives us an input MDP (S, A, P, R, y) and policy π
        - Our output would be a value function v_s for any given state s.
    - Control
        - Let's say somebody gives us an input MDP (S, A, P, R, y)
        - Output: Optimal value function v* and optimal policy π*
            - Basically, among all given policies, what's the best reward that can be achieved in this MDP or what's the best mapping from states to actions
- Looking at planning an MDP for prediction, let's say you're given the MDP and a given policy. The approach is that we use an iterative application of the Bellman expectation backup where where start with some v_1 and then compute v_2 and so on until we get vπ.
    - Using synchronous backups means that we consider all states (in our value function) before computing the successive one.
- Look at 20:27 - 29:30 in Lecture 3 for a really good example of how to come up with a value function for each location on a gridworld, and then how to create a greedy policy that acts based on that value function.
    - Main Takeaway: Value function helps us create better policies.
    - To create a value function for a random policy, you iterate the Bellman equation by doing one step lookaheads and figure out the new values at every state.
- Now that we know how to create our value function for an initial random policy, we know want to figure out how to get the best policy. In order to do this…
    - Let's say we're given a policy π
    - We're first going to evaluate this policy by figuring out the value function vπ for that given policy. This is the **policy evaluation** step.
    - Improve the policy by acting greedily with respect to the value function (The relation to the gridworld example is that the policy would change from random to a policy where we would take a step toward the location with the greater value function).This is the **policy improvement** step.
        - π' = greedy(vπ)
    - In the gridworld example, the improved policy of just acting greedily wrt to the values of the value function was all we needed to get an optimal policy.
    - In general, we need more iterations and the process of policy iteration **always** converges to the optimal policy π*.
- Let's look at that policy improvement step a bit more closely.
    - Consider a deterministic policy a = π(s)
    - We are able to obtain a better policy π' by seeing what action maximizes the action value function and then taking that action a.
        - π'(s) = argmax(qπ(s,a))
        - Remember that q is the immediate reward + the state value function of the location that you end up at.

    - It's guaranteed that you will get more value (a higher q function value) **at least** for the most immediate step (when compared to the q function if you just followed the original policy π).
        - $q\pi(s,\pi'(s)) >= v\pi(s)$
    - If the process stops (where the both terms in the above equation are equal), then we've satisfied the Bellman optimality equation (where our current policy is equal to the max of q) and then we've founded our optimal policy.
- Finding an optimal policy = We've solved the MDP!
- Let's say we don't necessarily want to get the *most* optimal policy?
    - We can introduce a stopping condition where if the change between iterations is less than some epsilon, then we stop iterating.
    - We can stop after k iterations.
- Any optimal policy can be separated into 2 components.
    - An optimal first action A
    - An optimal policy from the successor state S'.
- Theorem: A policy achieves the optimal value from state s ($v\pi(s) = v_*(s)$) if and only if for any state s' reachable from s, that the policy π is optimal from state s' onwards.
- IF we know $v_*(s')$, then we can also find $v_*(s)$ by using one-step lookaheads. This is the idea of **value iteration** where we're starting with final rewards and working backwards.
- In value iteration, we're trying to find the optimal policy and we're going to do this with an iterative application of the Bellman optimality backup.
- We're going from v_1 to v_* through iterations.
- The difference between value iteration and policy iteration is that we're not building and then testing an explicit policy at every step. In that previous method, we would evaluate a value function based on a policy, and then create a whole new policy, and then evaluate, and so on. That was **policy iteration**. In value iteration, we're just going from value function to value function until we reach an optimal solution.
    - One caveat is that if you stop the iterations at let's say v_{13}, then there might not actually be a policy that will yield that intermediate value function.
- 2 Different Planning Problems (Given an MDP, we just want to solve the MDP)
    - Prediction: We're trying to find how much reward you get for a given policy. In order to do this, we use the Bellman Expectation Equation and we do an Iterative Policy Evaluation.
    - Control: We're trying to find the optimal policy, and here we can use either value iteration or policy iteration. Value iteration uses the Bellman Optimality Equation while policy iteration uses the Bellman Expectation Equation as well as a Greedy Policy Improvement method.

Lecture 4 - Model Free Prediction

- MDP essentially tells you how the environment works, which realistically isn't going to be given most times.

- Model free methods go from the experience/interactions with the environment to the value functions and policies.
    - How to find the value function for an unknown MDP when someone gives us the policy.
- **Monte Carlo Learning**
    - Learn some value function directly from episodes of experience.
    - Must be some sort of game over time steps where there is always some termination.
    - If you get a return of 5 from the first episode, and a return of 7 from the 2nd episode, and then it terminates, the estimation of the value function of the start state is 6.
    - The goal is to learn a value function from experiences that result from some policy π. The value function is the expected return from time t onwards at a the current state. Instead of this expected return, Monte Carlo methods uses an empirical mean return.
- First Visit Monte-Carlo Policy Evaluation
    - To evaluate a state, consider the first time you visit the state, measure the total return you get from that point onward until termination (over all episodes), and then divide that by the number of times you visit that state over all episodes. This will give you the **mean return** which is the value.
- Every Visit Monte-Carlo Policy Evaluation
    - To evaluate a state, consider every time t you visit the state, measure the total return you get from that point onward until termination (over all episodes and including all the returns from all the time steps where you visited that state), and then divide that by the number of times you visit that state over all episodes. This will give you the **mean return** which is the value.
- We're going to make incremental updates to the value function/mean return by having a loop for every state S with return G in an episode,
    - We're going to have V(S) = V(S) + (1/N(S)) * (G-V(S))
    - G is the actual return we observe from a given state while V(S) was the expected return we thought we were going to get. This is somewhat of a loss function and weight update, where we are adjusting the value function so that it moves in the direction of making V(S) as close to G as possible.
    - This is done at the *end* of every episode because you have to look at what the actual return was for every state that you were in.
- You can also have an incremental mean that "forgets" about episodes a long time ago using some sort of decay factor.
- **Summary of Monte Carlo Learning**: You have some set of episodes, you look at the complete returns that you see, and then you update the estimate of the mean value to the sample return of each state you visit.
- **Temporal Difference Learning** is also a model free method of learn directly from past experiences, but unlike Monte Carlo, TD can learn from incomplete episodes by bootstrapping.

    - Bootstrapping is the idea of replacing the remainder of the trajectory with the estimate of what will happen from that position.
- TD has the same goal: To learn a value function under a certain policy.
- This time, our incremental update is $V(S) = V(S) + \text{alpha} * (R_{t+1} + V(S_{t+1}) - V(S))$
    - We basically substitute the real return G with an estimated return $R_{t+1} + V(S_{t+1})$
    - That estimated return is called a **TD Target**
    - The difference between estimated return and the predicted value function is the **TD Error**
- The return G (in MC method) is an *unbiased* estimated of the value function because G is like the actual return. However, the TD target $R_{t+1} + V(S_{t+1})$ is a *biased* estimate of the value function because the V is basically a guess of the value function, we don't know if it's actually accurate.
- TD target, however, has a much lower variance than actual return G because of the noisiness in returns from future time steps.
- MC converges to the solution with minimum mean-squared error between the actual returns G and the value functions over all time steps and episodes. TD converges to the solution of max likelihood MDP
- Difference between MC and TD
    - MC
        - Updates the value function based on the difference between the actual return and the predicted value function.
        - High variance, but no bias, which = good convergence properties.
        - More effective in non-Markov environments.
        - Doesn't use bootstrapping (estimate of trajectory)
    - TD
        - Can learn *before* knowing the final outcome, while MC has to wait until the end of the episode to know the actual return. Therefore, TD can learn in non-terminating, or **continuing** environments and from incomplete sequences.
        - Low variance, some bias, but is a lot more efficient than MC.
        - This is because TD makes use of the Markov property in that you don't need to just blindly look at complete trajectories. You can understand the environment in terms of states in that the states have to summarize everything that came before (pretty much definition of Markov property)
        - Uses bootstrapping
- Dynamic Programming is similar to TD in that it bootstraps (in one step lookaheads), but it doesn't sample in that DP uses MDP dynamics to compute a *full* expectation instead of sampling over the different routes.
- **TD (Lamda)** is a way of combining TD and MC in that you take n steps into the future and then compute the estimate return/value function.
    - $V(S) = V(S) + \text{alpha} * (Gn - V(S))$
- You can also average n step returns over different n
    - (1/2) * G2 + (1/2) * G4

- Averaging over all lamba is the TD(lamda) method
    - Glamda = (1 - lamda) * summation from n to infinity of lamda(n-1) * Gn
    - V(S) = V(S) + alpha * (Glamda -V(S))
    - Lamda is a hyperparameter that tells us how much to decay the weighting for each successive n.

Lecture 5 - Model Free Control

- In the last lecture, we learned how to estimate the value function for an unknown MDP using MC and TD and TD(lamda) methods.
- Now we want to optimize that value function and find the best policy.
- A lot of real world problems can be modelled as MDPs, but the problem is that sometimes the MDP is unknown and sometimes the MDP is known but is so complicated that it is easier to just sample over the environment and use a model free method.
- **On-policy learning**: “Learn on the job”. Learn about policy π from experience sampled from π. The actions that you take determine the policy you want to evaluate.
- **On-policy learning**: “Look at somebody else”. Learn about policy π from experience sampled from some other distribution.
- So, to find the optimal policy or to maximize the value function, we are going to use the same steps of policy evaluation and then greedy policy improvement.
    - The problem with just plugging in an MC method is that if you act greedily all the time, you don’t guarantee that the trajectories that you’re following will explore the whole state space.
    - The problem also with being model free is that we don’t necessarily know what’s the best way to improve the policy without having a model of the MDP (specifically without having a solid function V).
- The solution is to do a greedy policy improvement over Q instead of V.
- The follow is on-policy learning.
- In a model free environment (with no MDP), one problem is that you could get stuck in a local minima of a having policy that *seems* to have the most possible reward (but since we don’t know the dynamics of the system), we’re not sure if that really is the best choice since we haven’t explored the whole state space.
    - The solution is the **Epsilon-Greedy Exploration** where when we have to decide what action to take, there is probability epsilon that we choose an action at random and probability (1 - epsilon) that we choose the greedy action.
    - This makes sense if epsilon is a small number (and it probably will be). You want to make the most greedy action most of the time, but every now and again, you want to explore other actions.
- Policy evaluation = Monte-Carlo policy evaluation using action value function
- Policy Improvement = epsilon greedy policy improvement.
- Because we’re dealing with Monte Carlo, we have to do the evaluation/improvement process after every episode.
- **GLIE Monte Carlo**: Method for getting to an optimal Q function

    - Sample a bunch of episodes. Use Q(S,A) = Q(S,A) + alpha * (G -Q(S,A)) for the incremental updates to the Q function. This is the policy evaluation.
    - Then, improve policy using epsilon greedy policy improvement.
- Now, let's look on how TD fits in
- We're going to update our Q function using **SARSA**.
    - Q(S,A) = Q(S,A) + alpha * (R + gamma*(Q(S',A')) -Q(S,A))
- Policy evaluation = SARSA policy evaluation using action value function
- Policy Improvement = epsilon greedy policy improvement.
- There is also an n-step SARSA which is basically a combination of MC and TD (as described in the previous lecture)
- There is also a SARSA(lamda) which is similar to previous lecture
    - Qlamda = (1 - lamda) * summation from n to infinity of lamda(n-1) * Qn
    - Q(S,A) = Q(S,A) + alpha * (Qlamda -Q(S,A))
- Now, let's look at off-policy learning, which is the "looking over your shoulder" one.
- Evaluate some target policy π while following a different behaviour policy.
    - Goal is to compute v or q (for following policy π)
- ***Couldn't really understand off-policy MC and TD***
- Q Learning
    - Consider off-policy learning of Q function
    - Next action A_{t+1} is chosen using some behaviour policy mu
    - Consider an *alternative* action A' using some target policy π.
    - Update Q toward the value of alternative action.
    - Q(S,A) = Q(S,A) + alpha * (R + gamma*(Q(S_{t+1},A')) -Q(S,A))
- Q Learning algorithm seeks to learn about greedy behaviour while following some exploratory behaviour.
    - The target policy π is going to be greedy wrt to Q.
    - The behaviour policy mu is epsilon greedy wrt to Q.
    - The unique thing is that both the behaviour and target policies can improve.
    - The algorithm is SARSAMAX
        - Q(S,A) = Q(S,A) + alpha * (R + gamma*max(Q(S',a')) -Q(S,A))

Lecture 6 - Value Function Approximation

- In real world RL problems, there are millions of millions of states in the respective spaces and therefore we can't really "write everything in a table" now.
- Especially in continuous state spaces, it's important that our value functions understand generalization in that we don't have to store a value function for a certain location and then store a completely new function for the location 1 millimeter to the right.
- Up until now, every state s has had a value function V(s) and for every state-action pair s,a there is a Q(s,a) for the action value function.
    - Basically we have a lookup table for V and Q.
- The problem with large MDPs is that there are too many states and actions to store in memory and it is too slow to learn the value of each state individually.

- Solution is to compute the value function for unknown states using a function approximator.
    - **v**(s,w) = $v_\pi(s)$ (Equal sign supposed to be approximately equal to)
    - **v** is the approximation while $v_\pi(s)$ is the actual value function for that state.
    - We want to create some function that takes in the state s and a parameter vector w that outputs something close to the true value function
    - Update w using MC or TD learning
- J(w) is a function of parameter vector w and the gradient is a vector of the partial derivatives of J wrt to each of the elements in the w vector.
- Gradient descent means finding the local min of J(w) and then adjusting w in the direction of the negative gradient.
- IF this was like supervised learning, we could just use a MSE loss function where y is our true value function and yhat is our value function approximator and we could just minimize over that.
    - However, we don't have that true value function, so it's a bit harder
- First, we represent our state using a feature vector x(S). Each element in this feature vector could be distance from agent to some landmark, # of pieces on gameboard, etc.
- We're going to estimate our value function by taking the dot product of x(S) and the parameter vector w, and if you have that true value function, then you can run gradient descent.
- Since we don't have the true value function, we substitute a target that can be calculated using MC or TD.
    - MC: Substitute true value function with return G
        - We can create samples of "training data".
        - $<S_1, G_1>$, $<S_2, G_2>$....
        - Remember the unique thing about MC is that it is unbiased while TD is biased
    - TD: Substitute true value function with $R_{t+1}+ V(S_{t+1}, w)$
        - We can create samples of "training data".
        - $<S_1, R_2+ V(S_2, w)>$, $<S_2, R_3+ V(S_3, w)>$....
    - TD(lambda): Substitute true value function with return G lambda
        - Same as MC except with lambda returns
- In all these cases, the thing you're substituting is becoming your "label" and then it becomes a supervised learning problem as you run normal gradient descent to update your parameter vector w.
- We then do our policy evaluation and improvement steps using the function approximators
- Just like we did in last lecture, we're going to replace all our V's with Q's as we want to approximate the action-value function.
- **Q**(s,a,w) = $q_\pi(s,a)$ (Equal sign supposed to be approximately equal to)
- Now, since we have states *and* actions, we need a feature vector that is a function of both. x(S,A)

- We're going to estimate our action value function by taking the dot product of x(S,A) and the parameter vector w.
- Since we don't have the true action value function, we substitute a target that can be calculated using MC or TD. It will be the same as above (except replace the V with Q).
- The main reason for using Q instead of V is that we don't need the model and we can just choose the max over our actions.
- All of the methods described previously are not sample efficient as in we see an experience once, take an update, and then we throw that experience away and move onto the next one.
- Batch methods work to find the best value function that represents all the experiences we've see in that batch.
- We have some experience D where D = set of training examples with <state, value> pairs
- Goal is to find the parameters that will give you the best fitting value approximator.
- You can do this using least squares method which minimizes the sum of squared errors between **v**(s,w) and $v_\pi(s)$ for all training examples.
- We can solve this least squares problem using **experience replay**.
    - We basically make our experience D a cache for training examples.
    - At every time step, we randomly sample a state and value from our experience and then make one SGD (stochastic gradient descent) update toward that target.
    - This is pretty much supervised learning in that we're taking a random sample from our training data (experience D) and making a weight update
    - You can also solve the least squares problem through a linear algebra technique where you take the inverse (not totally clear to me)
- David then talks about DQN, which I summarized in the blog post :)

Lecture 7 - Policy Gradient Methods

- In last lecture, we talked about how to approximate our value function and our state value function using a weight vector w.
- In this lecture, we'll look at how we can directly parametrize the policy using a model free method.
- Might be better to work with policy methods as opposed to policy based methods because there are situations where it's easier/better to represent the policy instead of the value in that it is more compact than a value function.
    - It also has better convergence properties and are effective in high dimensional or continuous action spaces.
    - Disadvantage is that it typically converges to a local rather than global optimum.
    - Evaluating a policy is also inefficient and high variance.
    - Value based methods are extremely aggressive in that you get very close to the optimum policy whereas policy gradient methods take little steps in that direction.

- There are cases when it is desirable to have a stochastic policy as opposed to a deterministic one because finding a way to maximize reward can be a deterministic process.
    - Ex: Rock paper scissors because a deterministic policy is easily exploited.
- Policy based RL uses a parameterized policy where $\pi = g(x(s,a), w)$
- We can represent the quality of a policy through $J(w) = V(\text{start state})$
- Our goal now is to find some w that maximizes J(w). This is a sort of gradient ascent where we're trying to maximize this policy objective function J.
- We can compute the gradients numerically by using pretty much the limit definition. This is the method of **finite differences** where we basically perturb w by a little bit, and calculate the corresponding change in the J function.
    - Useful in low dimensions where we have light computational costs
- Now let's look at if we want to compute the policy gradient analytically.
- The gradient of $\pi(s,a) = \pi(s,a) *$ **gradient of log $\pi(s,a)$.** The bold component is our **score function**.
    - That is the term that tells you how to adjust your policy in the direction of how to get more of a particular action.
- The most simple policy is that of a softmax policy where we weight each action using a linear combination of features and weights. The probability of an action is proportional to the exponentiated weight.
- (Couldn't understand all of the details of Policy Gradient)