Summary of David Silver's Lecture on Reinforcement Learning

Jérémy Scheurer

September 24, 2018

Abstract

This is a summary on the Lectures of David Silver on Reinforcement Learning

1 Introduction to Reinforcement Learning

- Many branches of science have a crossover with Reinforcement Learning. This is because RL tries to solve the problem of decision making, which reappears in a lot of different sciences such as Computer science, Neuroscience, Psychology, Economy, Engineeringe etc.
- The difference between reinforcement learning and supervised or unsupervised learning is, that there is no supervisor which tells us what is the correct action. We only get a reward signal, and even that feedback may come much later. Time matters in RL, the learning is sequential, so we do not have i.i.d data. So what happens at one timestep correlates a lot to what happens in the next timestep. Also the agent can influence the environment and thus what it sees and learns.
- A reward R_t is a scalar feedback signal which indicates how well an agent is doing at step t. We cannot have a vector as a reward, because in the end we should be able to lay out every action based on one reward over timesteps. The agent's job is to maximize the cumulative reward. Reinforcement learning is based on the reward hypothesis:

  Theorem 1 *All goals can be described by the maximization of expected cumulative reward.*

- The goal is to select actions to maximize total future reward. It's important that one needs to plan(one cannot be greedy), because actions might have long term consequences. One might have to sacrifice immediate reward to gain more reward in the long term.
- We have an agent which observes observations O_t, it takes actions A_t and receives a reward R_t. On the other hand we have an environment which provides the observations and reward based on the action it gets.

- The history is what the reward has seen so far, i.e. all the observable variables up to time t.

  $$H_t = A_1, O_1, R_1, ...A_t, O_t, R_t$$

  The agent selects actions based on this history. So this history determines what it is going to happen next. But we do not want to go back through the whole history every time to predict the next action, thus we save everything in a state.

  $$s_t = f(H_t)$$

- The environment state S_t^e is the environment's private representation. That means it is whatever data the environment uses to pick the next observation and reward. So it's not visible to the agent. Even if S_t^e is visible, it may contain irrelevant information.

- The agent state S_t^a is the agent's internal representation. That means whatever information the agent sues to pick the next action.

- An iformation state(a.k.a. Markov state) contains a ll useful information from the history. So we make a Markov assumption, which means that the future is independent of the past given the rpesent.

  Theorem 2 $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$

- Fully observable environments are environments where the agent directly observes the environment state, i.e. $O_t = S_t^a = S_t^e$. This is a Markov decision process. The other case is where we have a partial observability of th environment. The agent indirectly observes the environment, ex. A robot with camera vision isn't told its absolute location. This is now a partially observable Markov decision process. Now the agent must construct its own state representation, ex:

  - Complete History $S_t^a = H_t$
  - Beliefs of environment states $S_t^a = (P[S_t^e = s^1], ..., P[S_t^e = s^n])$
  - Recurrent neural network: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$

- an RL agent my include one or more of the components:

  - Policy: Agent's behavior function, how it picks its action.
  - Value Function: How good is each state and/or action.
  - Model: Agent's representation of the environment.

- A policy is the agent's behavior. it is a map from state to action, i.e. deterministic policy $a = \pi(s)$ or stochastic policy $\pi(a|s) = P[A = a|S = s]$

- The value function is a prediction of expected future reward. So it is used to evaluate the goodness/badness of states and therefore to select between actions. The value function is dependent on you behavior(so the policy)

  $$v_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s]$$

  where gamma are the discounting factors.

- A model predicts what the environment will do next. So its not exactly the environment, but what the agent thinks, the environment might do in the future. We could try to learn transitions which predict the next state. Or we could learn a reward model which predicts the next immediate reward.

$$\rho_{ss}^{a} = P[S' = s'|S = s, A = a]$$

$$R_{s}^{a} = E[R|S = s, A = a]$$

- We can now taxonomize RL based on these criterion's.
  - It's a Value Based Algorithm if it contains a value function. It could potentially only have a value function without a policy and then read out the best states.
  - It's policy based if it only has a Policy and updates those according to the reward.
  - Actor critic RL agent has policies and value functions.
  - Now it can also be a model-free or model-based RL agent.
- There are several problems in reinforcement learning, are two of the biggest problem settings:
  - Reinforcement Learning Problem: The environment is initially unknown.The agent interacts with the environment and improves its policy.(Atari game)
  - Planning: A model of the environment is known. The agent performs computations with its model(without any external interaction), so it only kind of thinks. The agent then improves its policy. So this allows us to build a search tree and look ahead without actually acting.(If we had access to an emulator of the Atari game)
  - The exploration and exploitation Trade off. Reinforcement Learning is like trial and error learning. The agent should discover a good policy but it should not give up opportunity of exploiting what is has discovered.
  - The final problem is prediction and control. Prediction means to evaluate the future with a given policy. Control means to optimize the future i.e. find the best policy. In RL usually we have to solve the prediction problem to solve the Control problem.

2 Markov Decision Processes

- Markov decision process formally describes an environment for reinforcement learning where the environment is fully observable, i.e. the current state completely characterizes the process. Almost all RL problems can be formalized as MDP's. Even partially observable problems can be converted into MDP's.
- The central property of an MDP is the Markov property(see Markov Theorem). For a Markov state s and successor state s', the state transition probability is defined by

$$P_{ss} = P[S_{t+1} = s'|S_t = s]$$

Theorem 3 *A Markov process is a memoryless random process, i.e. a sequence of random states S1, S2, ... with the Markov property. So a Markov Process is a Tuple $< S, P >$, where S are a set of finite states and is a state transition probability matrix.*

- **Theorem 4** *A Markov reward process is a Markov chain with values. It is a Tuple $< S, P, R, \gamma >$, where S is a finite set of states, P is a state transition probability matrix, R is a reward function $R_s = E[R_{t+1}|S_t = s]$ and $\gamma \epsilon [0, 1]$ is a discount facto.*

- The goal of RL is to maximize the return G_t which is the total discounted reward from time step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Why should we use a discount factor? Because we cannot trust our decisions fully, because we only have a model of the environment and we cannot be perfectly certain about the future. Also its mathematically convenient and avoids infinite returns in cyclic Markov processes. Also if the reward is financial, immediate rewards may earn more interest than delayed rewards. Cognitively animals and humans prefer immediate rewards. If all possible return sequences are finite we can omit discounting.

- **Theorem 5** *Value function v(s) gives the long term value of state s. So the state value funciton of an MRP is th eexpected return starting from state s:*

$$v(s) = E[G_t|S_t = s]$$

- **Theorem 6** *The Bellman Equation for MRP's. The value funciton can be decomposed into two parts: immediate reward R_{t+1} and discounted value of successor state:*

$$\begin{aligned} v(s) &= E[G_t|S_t = s] \\ &= E[R_{t+1} + \gamma R_{t+2} + y^2 R_{t+3} + ...|S_t = s] \\ &= E[R_{t+1} + \gamma R_{t+2} + y^2 R_{t+3} + ...|S_t = s] \\ &= E[R_{t+1} + \gamma (R_{t+2} + y R_{t+3} + ...)|S_t = s] \\ &= E[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= E[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \end{aligned}$$

  In other words:

$$v(s) = R_s + \gamma \sum_{s' \epsilon S} P_{ss'} v(s')$$

- The Bellman equation can be formulated in a vector/matrix formulation: $v = R + \gamma v$ where $v \epsilon R^n, R \epsilon R^n, P \epsilon R^{n,n}$ The Bellman equation is a linear equation and can directly be solved. This is not going to be true for Markov Decision Processes. It only works for evaluating rewards, if we want to maximize rewards, its not possible anymore.

$$v = (I - \gamma P)^{-1} R$$

Note that the computational complexity is $O(n^3)$ for n states. Direct solutions only possible for small Markov reward Processes. There are many iterative methods for lagre MRP's such as Dynamic programming, Monte-Carlo evaluation and Temporal-Difference learning.

- Now we look at what we actually use in RL which are Markov decision Processes.

  Theorem 7 *A Markov decision process is a Markov reward process with decisions. It is an environment in which all states are Markov. It is a tuple* $< S, A, P, R, \gamma >$ *where* S *is a finite set of states,* A *is a finite set of actions,* P *is a state transition probability matrix,* γ *is a discount factor,* R *is a reward function,* $R_s^a = E[R_{t+1}|S_t = s, A_t = a]$ *and* $P_{ss'}^a = P[S_{t+1} = s'|S_t = s, A_t = a]$.

- So now we want to find a path(select actions) through our MDP which maximizes the rewards. To do this we need a Policy which fully defines the behaviour of an agent.

  Theorem 8 *A policy* π *is a distribution over actions given states:*

$$\pi(a|s) = P[A_t = a|S_t = s]$$

- Policies are stationary, i.e. time-independent.

- Note the connection between an MDP and a MRP is the following. If we fix a policy and sample a state sequence, this sequence is a Markov process. The state and reward sequence we get is a Markov reward Process. In other words one takes the average of the dynamics as an MRP, i.e.

$$P_{s,s'}^{\pi} = \sum_{a \epsilon A} \pi(a|s) P_{s,s'}^a$$

$$R^{\pi} = \sum_{a \epsilon A} \pi(a|s) R_s^a$$

- How good is it to be in a certain state.

  Theorem 9 *The state-value function* v_π *of an MDP is the expected return starting from state* s*, and then following policy* π

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

- How good is it to take a specific action in a certain state.

  Theorem 10 *The action value function* $q_\pi(s, a)$ *is the expected return starting from state* s*, taking action* a*, and then following policy* π

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$$

- The state-value function can again be decomposed into immediate reward plus discounted value of successor state.

Theorem 11 *Bellmann-Equation:*

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

The action value function can similarly be decomposed:

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

- How do V and Q relate to each other?

$$v_\pi(s) = \sum_{a \epsilon A} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \epsilon S} P_{ss}^a v_\pi(s')$$

You can now insert the second equation into the first equation which gives the Bellmann equation. You could also insert the first equation into the second equation. The difference is in the first method you go from a state and you say okay what different actions could I take. And then you ask after taking these actions where might the wind blow you. For the second method you consider a specific action and wonder where the wind might blow you and from that state you consider all possible actions you could take.

- But what we now want is not to have any policy but the best policy.

Theorem 12 *The optimal state-value function v_* is the maximum value funciton over all policies:*

$$v_*(s) = max_\pi v_\pi(s)$$

The optimal action-value function

$$q_*(s,a) = max_\pi q_\pi(s,a)$$

The optimal value function specifies the best possible performance in the MDP. And MDP is "solved" when we know the optimal value function.
If you think of it, you are trying to find the policy which maximizes the rewards you get in this system and thus maximizes the expected return starting from state s and following policy π(state value function). Or in other words, under every possible policy, we will get a specific state value function which is maximal.
Also if you have the optimal action-value function you are basically done, because you can insert any action possible and then choose the one with the larges action value function.

- We are interested in the optimal policy which is just a function from states to actions. So to find a best we need to introduce a partial ordering of policies. $\pi >= \pi'$ if $v_\pi(s) >= v_{\pi'}(s)$, fo rall s.

Theorem 13 *For any Markov Decision Process:*

- *There exists and optimal policy π_*, that is better than or equal to all other policies.*
- *All optimal policies achieve the optimal value function, $v_{\pi_*} = v_*(s)$*
- *All optimal policies achieve the optimal action -value function, $q_{\pi_*}(s,a) = q_*(s,a)$*

- An optimal policy can be found by maximizing over $q_*(s,a)$, i.e.

$$\pi_*(a|s) = 1 if a = argmax_{a \epsilon A} q_*(s,a)$$

and 0 otherwise

- How can we figure out q_*?

Theorem 14 *The optimal value functions are recursively related by the Bellman optimality equations:*

$$v_*(s) = max_a q_*(s,a)$$

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \epsilon S} P_{ss'}^a v * (s')$$

We can now again insert the second term in the first term or the first into the second.

- Bellman Optimality Equation is non-linear, how can we solve it? There are many iterative solution methods such as value iteration, policy iteration, Q-learning and Sarsa.

3 Planning by Dynamic Programming

- There are different ways of planning, here a few of them we will look at. Policy evaluation, which means given a certain policy, how good is this policy. Then there is policy iteration which takes the idea of policy evaluation to find the best policy. Value iteration optimizes the Bellmann Equation.

- What is dynamic programming? It is optimizing a mathematical policy which has a sequential or temporal component to the problem. You can divide your problem into substructures and solve these. A DP needs two properties. First of all the principle of optimality, i.e. solving the sub-problems also solves the initial problem. The second is overlapping sub-problems which means that one sub-problem is a sub-problem for different initial problems.
  Luckily a MDP satisfies both properties. The Bellman equation gives recursive decomposition. The value functions stores and reuses solutions.

- The difference between RL and Planning is that in planning, we know the transition and reward structures and try to find the optimal policy.

- DP assumes full knowledge of the MDP and is used for planning.

- For prediction we have the input $(< S, A, P, R, \gamma)$ and a policy π, the output is v_π. So this is something like policy evaluation.
- For control(what we really want to find out): The input is $(< S, A, P, R, \gamma)$ and the output is the optimal value function v_* and π_*, so the most reward we can get and thus also what is the best policy.

3.1 Policy Evaluation

- Here we use the Bellman expectation equation and for the Control problem we use the Bellman optimality equation. The problem we have here is to evaluate a given policy π. So we will start with an initial value function and iterate the Bellman expectation backup.
- Synchronous Backups: At each iteration $k + 1$ for all states s, update $v_{k+1}(s)$ from $v_k(s')$, where s' is a successor state of s. That means we iteratively calculate the following:

$$v_{k+1}(s) = \sum_{a \epsilon A} \pi(a|s)(R_s^a + \gamma \sum_{s' \epsilon S} P_{ss'}^a v_k(s'))$$

- If we look at a grid example we can iteratively apply the expectation Bellman equation to find the true value function for the states. One can then learn new policies by acting greedely and always move around according to the best states. But note that here we improved a policy, but the state value improvement with the Bellman equation always assumed a random policy. This now leads us to the next topic.

3.2 Policy Iteration

- The goal is to find an optimal policy given a policy π. How can we improve the policy:
  Evaluate the policy π:

  $$v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + ...|S_t = s]$$

  Then improve the policy by acting greedily with respect to v_π

  $$\pi' = greedy(v_\pi)$$

  In our grid world example we only had to go through the algorithm once to get the optimal policy. But usually we need to do this several times to get smoothing optimal.
- Why do we find the optimal policy with this algorithm? Consider a deterministic policy, $a = \pi(s)$. We can improve the policy by acting greedily: $\pi'(s) = argmax_{a \epsilon A} q_\pi(s, a)$. This imprpoves the value from any state s over one step,

  $$q_\pi(s, \pi'(s)) = max_{a \epsilon A} q_\pi(s, a) >= q_\pi(s, \pi(s)) = v_\pi(s)$$

  With a telescoping prove one can then show that it therefore improves the value function, i.e. $v_{\pi'}(s) >= v_\pi(s)$ Proof:

  $$v_\pi(s) <= q_\pi(s, \pi'(s)) = E_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

$$<= E_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(St+1)))|S_t = s]$$
$$<= E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2}))|S_t = s]$$
$$E_{\pi'}[R_{t+1} + \gamma R_{t+2} + ...|S_t = s] = v_{\pi'}(s)$$

At some point this iterative method is going to stop because it is equally good as the last step. How can we now prove, that this is the optimal policy? If improvements stop,

$$q_\pi(s, \pi'(s)) = max_{a\epsilon A} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

In which case the Bellman optimality equation has been satisfied because :

$$v_\pi(s) = max_{a\epsilon A} q_\pi(s, a)$$

- We saw in the grid world that improving the state value function only a little bit already got us the best optimal policy. So how could we use this and not calculate our algorithm till the end. We could introduce a stopping condition epsilon which looks at convergence of the value function. But even that would waste some effort. So what we could do is to evaluate a policy k-times(improve the value function) and after k steps improve the policy. This also converges to the optimal policy. The extreme case of this is to update the policy every iteration, i.e. stop after k = 1. This is equivalent to value iteration.

3.3 Value iteration

- Any optimal policy can be subdivided into two components, an optimal first action A*, followed by an optimal policy from successor state S'.

  Theorem 15 *principle of Optimality for policies: A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$ if and only if for any state s' reachable from s, πachieves the optimal value from state s', i.e. $v_\pi(s') = v_*(s')$*

- If we know the solution to subproblems $v_*(s')$. Then solution $v_*(s)$ can be found by one-step lookahead:

$$v_*(s) < -max_{a\epsilon A} R_s^a + \gamma \sum_{s'\epsilon S} P_{ss'}^a v_*(s')$$

  You have to note that the idea of having a final state and working backwards is only for the intuition. The algorithm also works if there is no final state and also if we don't know the final state.
  The idea is that each state gets to be the root state. We then initiate all states with some value function(ex. 0). Then for all states(where each individual state is considered to be the root state) we take the max action, which maximizes a one step look ahead of what the environment could do based on this action(where the wind might bring us).

- So in value iteration we try to find the optimal policy in the planning problem(not RL problem). The solution is to iteratively apply the Bellman optimality backup. Also note that before(policy iteration) we applied the Bellman expectation equation which found us the value function for a given policy. No we iterate the Bellman optimality backup which finds us the optimal value function.

- So what is the difference between policy iteration and value iteration? In value iteration, we do not explicitly build a policy at each iteration. Also if you just stop at any value iteration, this value function might not be meaningful for a specific policy, but in the end it will. Also remember that this is equivalent to policy iteration with k = 1. So you evaluate the expected Bellman equation at each step for a new greedy policy which is the argmax and which then yields a max.

- So to summarize what we have done until now, this were all synchronous dynamic programming algorithms.

  - Prediction: Use Bellman Expectation Equation (iterative policy evaluation).
  - Control: Use Bellman Expectation Equation + Greedy Policy Improvement (Policy iteration)
  - Bellman Optimality Equation (Value Iteration).

  All algorithms are based on state value function $v_\pi(s)$ or $v_*(s)$. The complexity is $O(mn^2)$ per iteration for m actions and n states. One could also apply it to action-value functions $q_\pi(s,a)$ or $q_*(s,a)$ which would then have a complexity of $O(m^2n^2)$ per iteration.

3.4 Extensions to Dynamic Programming

- Do you have to look at every single state in your iteration(i.e. all states are backed up in parallel)? No. You can do it asynchronously. So for each selected state, apply the appropriate backup. It will also converge as long as you still select all states frequently, but the order does not matter.

- In-place dynamic programming. You always have to store the old and the new value of your value function for synchronous value iteration. In-place value iteration just uses one memory location for the new and old value functions. That is you just replace the old one by the new one. But you still have a value function for each state.

- Prioritized sweeping tries to come up with an order on how to update the states. The intuition is that the states whose value changes the most, is an important state which one should update the first, because it might influence all tohe other states.

- Real time dynamic programming whose idea is to select the states which the agent actually visits(and the ones near it).

- DP uses full-width backups. That means in each step for each state we look at all possible actions and every successor state. But this is not efficient for

millions of states. The way we solve this problem is by sampling certain trajectories, i.e. actions and where we end up.

4 Model-Free Prediction

- These are situations in which we do not know the dynamics of our environment. We will look at two topics, Monte-Carlo Learning and Temporal Difference Learning. The first one takes samples all the way to the end of the trajectory and then calculates the returns. The second one just looks one step ahead and estimate the further steps. With TD(lambda) methods we can combine both methods.

- Last time we looked at planning by dynamic programming to solve a known MDP. Today we will estimate the value function of an unknown MDP(Model-free prediction). Next time we will optimize the value function of an unknown MDP(Model-free control).

4.1 Monte-Carlo Learning

- MC methods learn directly from episodes of experience(ex. games). MC is model-free: no knowledge of MDP transitions/rewards needed. MC learns from complete episodes, i.e. no bootstrapping. MC uses the simplest possible idea: value = mean return. For example, if we look at two episodes and in one we get a rewards of 7 and in the other a reward of 5, we already know that the start state has a value of 6. The caveat of this is that one can only apply MC to episodic MDP's and all episodes must terminate.

- The goal is to learn v_π from episodes of experience under policy π. Recall that the return is the total discounted reward G_t and the value function is the expected return under our policy starting from a specific state.

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

  Monte-Carlo policy evaluation uses the empirical mean return instead of the expected return. There are two ways to do this.

- First Visit Monte-Carlo Policy. Assume we have some loop in our MDP, but to evaluate a state we say that we are only going to use the first time we visited that state. Then you take the mean of the rewards of the following trajectory from that state on. Note that in each episode if you come back to that state for the first time you do this.
  Goal is to evaluate state s. The first time-step t that the state s is visited in an episode, we increment a counter $N(s) < -N(s) + 1$ and also increment the total return $S(s) < -S(s) + G_t$. The value of the state is estimated by the mean return $V(s) = S(s)/N(s)$. By the law of large numbers, $V(s) - > v_\pi(s) as N(s) - > \infty$

- Now here is a slightly different approach. We do everything exactly the same way, except that every time-step t that state s is visited in an episode we increment the counter and the total return.

- Note that we are able to calculate the mean incrementally.

$$\mu_{k-1} + 1/k(x_k - u_{k-1})$$

Oftentimes we will see exactly this form, where the mean u_{k-1} is an estimate for what we will get as a reward and G_t is the actual empirical total reward for following trajectory in this current episode. If our estimate is higher than the empirical mean, we decrease our estimate and if we are lower we increase it.

- Incremental Monte-Carlo Updates work as follows: Update $V(s)$ incrementally after an episode. For each state S_t with return G_t we do

$$N(S_t) < -N(S_t) + 1$$

and

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

Remember that all these updates will be done at the end of our episode, because G_t is actually the empirical rewards we receive for a specific trajectory.

- In the non-stationary problems, it can be useful to track a running mean, i.e. forget old episodes $V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$

4.2 Temporal-Difference Learning

- TD methods learn directly from episodes of experience. TD is mode-free and it learns from incomplete episodes, i.e. by bootstrapping. So we do not have to go all the way to the end, one can go a certain number of steps and then use an estimate of how much reward one think will get from there till the end. So TD updates a guess towards a guess.

- The goal is to learn v_π online from experience under policy π. Incremental every-visit Monte-Carlo updates a value $v(S_t)$ towards an actual return G_t i.e.

$$V(S_t)?V(S_t) + \alpha(G_t - V(S_t))$$

. Where again G_t is calculated at the end through the actual rewards one got.

- The simplest temporal-difference learning algorithm(TD(0)):
Update value $V(S_t)$ toward estimated return $R_{t+1} + \gamma V(S_{t+1}$ i.e.

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Note that $R_{t+1} + \gamma V(S_{t+1}$ is called the TD target. And also we define $\delta_t = R_{t+1} + \gamma V(S_{t+1} - V(S_t)$ is called the TD error.

- Why is this a good idea? Assume you are driving in a car and you see another one which comes towards you and you think you are going to crash, but in the end you are not going to crash. In Monte Carlo learning you will not be able to update your state value with a negative reward, because in the end you arrive home. In TD-Learning you will put a new value to the state of almost dying which will be very low.

- Advantages and Disadvantages of MC vs. TD:
  - TD can learn before knowing the final outcome, i.e. TD can learn online after every step. MC must wait until the end of an episode before the return is known.
  - TD can learn without the final outcome, i.e. TD can learn from incomplete sequences. MC can only learn from complete sequences.
  - TD works in continuing(non-terminating) environments, whereas MC only works for episodic(terminating) environments.
  - Note that TD can learn the true value function. As long as you run it out, and you have a bad guess, it will always come closer to a true value function.
  - The return G_t of Monte-Carlo is an unbiased estimate of the true value function. Also assuming we knew the true value function under a certain policy, the estimate is also unbiased. But what we actually do is that we do not know the true value function and we only have an estimate for it, and this in turn is biased. But on the other hand the variance of TD is much lower than for MC. But why? Because there is always noise in making a decision for a next state and thus for MC as we add up all rewards for a whole trajectory we add up much noise. In TD we only have noise for one reward, the rest is an estimate of a value function and has no noise.
  - MC has high variance and zero bias. So good convergence properties(even for function approximation), not very sensitive to initial value and simple to understand and use.
    Note: What do we mean by function approximation? We will see in a couple of lectures, that everything we have done up to now is not practical for RL. Because we estimate the value function for all states separately and usually we have more states that we can count. What we will then do is use function approximators.
  - Has low variance and some bias. It is usually more efficient than MC and it also converges to the true value function. But it is more sensitive to initial values.
  - The MC algorithm converges to a solution with minimum mean-squared error which best fits to the observed returns. The TD algorithms converges to a solution of a maximum likelihood Markov Model which best fits the data.
  - TD exploits Markov property(that's why it is more efficient) and MC does not exploti Markov property(might be better if we are in an environment with non Markov property).
- So to summarize, in MC we sample a whole trajectory with certain actions and certain environment dynamics which take us to specific places given one action(where wind might blow us). In TD we do a one step look-ahead, where this one step is just sample. In Dynamic Programming we also did a one step look-ahead, but we had to know the MDP and Environment because we basically computed an expectation of all possible actions and where the wind might blow us for one step. The last possibility we have is

to do an exhaustive treesearch of the whole tree which has an exponential blow up.

- Bootstrapping(you us an estimate value function, rather than real rewards): MC does not bootstrap, TD bootstraps, DP bootstraps.
  Sampling: MC samples, TD samples, DP does not sample.

4.3 TD(λ)

- There is not just MC or TD, but there is also something between it where we do a mix of bootstrapping and not bootstrapping.
- The idea is to take n-steps of reality and after that take the value function(estimate) of the rest. Define n-step return:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}$$

  and n-step temporal difference learning is:

$$V(S_t) = V(S_t) + \alpha(G_t^n - V(S_t))$$

- How can we have an algorithm which uses the best? We can average n-step returns over different n, i.e. average the 2-step and 4-step returns. Combines information from two different time-steps. The big question is now, how can we efficiently combine all n-steps?
- TD-λ Algorithm which is a geometric average: The λ-return G_t^λ combines all n-step returns $G_t^{(n)}$. Using weight $(1-\lambda)\lambda^{n-1}$ we have:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^n$$

  and then the forward-view TD(λ) is:

$$V(S_t) = V(S_t) + \alpha(G_t^\lambda - V(S_t))$$

  Note that if you have an infinite MDP you just assume that you are staying in the final state. That means that we summarize all infinite states with the last sequence(where we stay in the last state). Thus the last sequence might have more weight than previous ones, although before the weight was always decaying.
  Why do we use geometric weighting? It is memory-less and thus we can do it in the same time as TD(0), so its only about computation time.
- So far we had a forward-view TD(λ), which is similar to Monte-Carlo meaning that we need to wait till the end of an episode. We can only compute n-steps returns if we actually saw n-steps. But there is an equivalent mechanistic view that achieves the same result but without having to wait till the result appears.

- Eligibility traces: Credit assignment problem: Did the bell or light cause shock? In the frequency heuristic we assign credit to most frequent states and in the Recency heuristic we assign credit to the most recent states. Eligibility traces combine both heuristics:

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s)$$

  In words, when we see a specific state, we accumulate the eligibility trace. When we don't see the state, it exponentially decreases. One can then update the value function in proportion to the eligibility trace.

- Backward view TD(λ): Keep an eligibility trace for every state. Update value V(s) for every state s in proportion to TD-error δ_t and eligibility trace $E_t(s)$:

$$\delta_t = R_{t+1} + \gamma v(S_{t+1} - V(S_t)$$

$$v(s) = V(s) + \alpha\delta_t E_t(s)$$

  This leads to the state we think are responsible for a certain state, to be updated the most.

- How does this relate to the algorithms we have seen so far? One can see that if $\lambda = 0$ only the current state is updated. So we will only update the value function if we currently are in state s which is exactly what TD(0) does.

$$E_t(s) = 1(S_t = s)$$

$$V(s) = V(s) + \alpha\delta_t E_t(s)$$

  which then leads to a TD(0) update:

$$v(S_t) = V(S_t)\alpha\delta_t$$

  .
  In the other extreme we have $\lambda = 1$, in which case credit is deferred until the end of an episode. So in an episodic environment, if we accumulate all of the updates we do with the backup view, we get the same as for the forward view.

  Theorem 16 *The sum of offline updates is identical for forward-view and backward-view $TD(\lambda)$:*

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \sum_{t=1}^{T} \alpha(G_t^\lambda - V(S_t))1(S_t = s)$$

5 Model-Free Control

- Everything we have done up to now, was leading to this chapter. Today we will look at how your agent can learn something if it is dropped into an environment in which it knows nothing. In other lectures we will look at

how we can scale all of that up. We will look at on-policy Monte Carlo and on-policy TD Learning, and also at off-policy learning. Last lecture we looked at model-free prediction, while we will use this today to do model-free control. So last time we estimated the value function of an unknown MDP and today we will optimize the value function of an unknown MDP.

- Often times for problems we either don't know the MDP model, but experience can be samples or the MDP is known, but it is too big to use, except by samples. Model-free control can solve both of these problems.
- On-policy learning is learning on the job. That means we actually apply a specific policy and learn about it while doing it. On the other hand, off-policy learning means to "look over someone's shoulder". That means we learn about a specific policy from experience sampled from somebody else.
- The main idea we are going to use is the generalized policy iteration. Where we first evaluate a policy, i.e. we estimate v_π e.g. by iterative policy evaluation. This is then followed by policy improvement, i.e. we improve the policy we have e.g. by a greedy policy improvement. Today we are going to vary what slots into the two parts.
- Generalized policy iteration with Monte-Carlo Evaluation: Let's start with Monte-Carlo evaluation. That means for the policy evaluation($v = v_\pi$) we use Monte-Carlo policy evaluation instead of dynamic programming. Remember that Monte-Carlo is also a method to evaluate a policy, but instead of looking at all possible actions we could take under the policy and where we could end up etc. we just sample specific trajectories, take the average and evaluate our state from that.
- What are problems with that approach? For the first step the problem is, that we need to sample a lot of episodes to get a good policy evaluation(we will improve on that using TD). But the real problem is that we try to be model-free, but with a policy improvement over V(s) we require the model of the MDP:

  $$\pi'(s) = argmax_{a\epsilon A} R_s^a + P_{ss'}^a V(s')$$

  So if you take an action we do not know where you might end up(where the wind might push you to). The alternative which is model free is to use action value functions:

  $$\pi'(s) = argmax_{a\epsilon A} Q(s, a)$$

  There is also an exploration issue with the second step, if you act greedily all the time, you don't guarantee that the trajectory you follow will explore the entire state-space.
- How do we make sure to explore states? We do ϵ-Greedy Exploration instead of greedy-exploration. All m actions are tried with non-zero probability. With probability ϵ you choose the greedy action and with probability ϵ you choose an action at random.

- **Theorem 17** *For any epsilon-greedy policy π, the ϵ-greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) >= v_\pi(s)$*

- Remember that it is not absolutely necessary to fully evaluate our whole policy all the time. We can only look at some specific states only. So we could sample one episode and then update only the value of the states that were actually sampled and then improve our policy right away.

- How can we really guarantee that we find the best possible policy? We have to balance exploring and finally getting to a policy where we are not exploring anymore, because it is optimal.

  Theorem 18 *Greedy in the Limit Infinite Exploration(GLIE). All state-action pairs are explored infinitely many times, $lim_{k\to\infty}N_k(s,a) = \infty$. The policy converges on a greedy policy, $lim_{k\to\infty}\pi_k(a|s) = 1(a = argmax_{a'\epsilon A}Q_k(s,a'))$ For example epsilon-greedy has this property, that if we sometimes behave randomly, every state will be visited infinitely may times. Because in the end we need to satisfy the Bellman equation, we do not want to explore anymore. ONe simple idea is to choose an epsilon greedy policy and to slowly decay epsilon to zero.*

- GLIE Monte-Carlo Control Algorithm: Sample k-th episode using π : $S_1, A_1, R_2, ..., S_T$ π. For each state S_t and action A_t in the episode,

$$N(S_t, A_t) = N(S_t, A_t) + 1$$

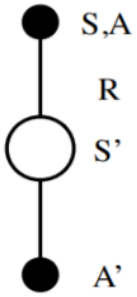
$$Q(S_t, A_t) = q(S_t, A_t) + \frac{1}{N(S_t, A_t}(G_t - Q(S_t, A_t))$$

  . Then we improve our policy based on new aciton-value function $\epsilon = 1/k$ and $\pi = \epsilon - greedy(Q)$
  Note that the further we improve our policy, the better our reward will be.

- The initialization of Q in theory does not matter. But in practice we want to have a good estimate, such that the algorithm terminates faster. But in this algorithm we just saw, it does not matter, because we replace Q after the first step anyway.

- We will now look at TD learning which has several advantages over Monte-Carlo, such as a lower variance, one can run it online and one can use incomplete sequences. The idea is to use TD instead of MC in our control loop(but still have iterative policy improvement), i.e. apply TD to Q(S,A) and use ϵ-greedy policy improvement and update every times-step.

5.1 SARSA



$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right)$$

- That means now I do a policy evaluation with Sarsa and after each evaluation I do a policy update(ex. with epsilon greedy). Here the algorithm:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

- Sarsa is an on-policy algorithm.
- We can also do n-step Sarsa, where we look at several rewards and then look at the q-value. If we look at all rewards from the whole episode, we have MC.

$$Q(S_t,A_t) = Q(S_t,A_t) + \alpha(q_t^{(n)} - Q(S_t,A_t))$$

with the n-step Q-return

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

- There is also a forward View Sarsa(λ). The q^λ return combines all n-step Q-returns $q_t^{(n)}$, using weight $(1-\lambda)\lambda^{n-1}$:

$$q_t^\lambda = (1-)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

.
Forward-view Sarsa(λ):

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(q_t^\lambda - Q(S_t, A_t))$$

- Remember there is also a backward view, because the problem of the forward view is that we do not have an online algorithm. We need to wait until the end of an episode to update a current state. Just like in TD(lambda) we use eligibility traces in an online algorithm. But Sarsa(lambda) has one eligibility trace for each state-action pair:

$$E_0(s, a) = 0$$

$$E_t(s, a) = \gamma\lambda E_{t-1}(s, a) + 1(S_t = s, A_t = a)$$

So When we visit a state action pair, we increment it. And all the states that we do not visit we decay a little bit, until we visit them again. Q(s,a) is updated for every state s and action a in proportion to TD-error δ_t and eligibility trace $E_t(s, a)$:

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

$$Q(s, a) = Q(s, a) + \alpha\delta_t E_t(s, a)$$

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s, a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta E(s, a)$
      $E(s, a) \leftarrow \gamma\lambda E(s, a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

- What is the difference between Sarsa(0) and Sarsa(lambda)? When you are in a grid world and you have a random walk which eventually leads

you to some gold, with Sarsa(0) you only update the value of the second last state(just before the gold). Because you thought you would get a reward of 0 but actually got a reward of 1. All other states remain at value 0. If you now would take the exact same path again, the state of 1 is propagated back a little bit more. For Sarsa(lambda) with an Eligibility trace you increase all the states of the path according to the weight of the eligibility trace, i.e. the last steps get more update than the first ones. Note if lambda is equal to 1 we have the same as MC, because all the old states are not decayed.

- If we look at it with the forward view, you look at a state and you have to wait until the end of the episode. And for each state you look at the one-step return, two-step return, which all would yield nothing. Except the n-step return which would give you something, but which would be weighted such that the actual update would be exactly the same as with the eligibility trace backward view(there we also have a weighting with the eligibility trace). But the backward view is much more efficient.

5.2 Off- Policy Learning/Q-learning

- evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$, while following behaviour policy $\mu(a|s)$. Why would we want to do this? Because we want to learn from observing humans or other agents. We also mgiht want to re-use experience generated from old policies. Or we might want to learn about optimal policy while following an exploratory policy. Or finally we might want to learn about multiple policies while following one policy.

- Importance sampling: Estimate the expectation of a different distribution.

$$E_{X\ P}[f(X)] = \sum P(X)f(x) = \sum Q(x)\frac{P(x)}{Q(x)}f(x) = E_{X\ Q}[\frac{P(X)}{Q(x)}f(X)]$$

- We can apply importance sampling to Monte-Carlo learning by using returns generated from μ to evaluate π. Weight return G_t according to similarity between policies. Multiply importance sampling corrections along whole episode

$$G_t^{\pi/mu} = \frac{\pi(A_t|S_t)\pi(A_{t+1}|S_{t+1})}{\mu(A_t|S_t)\mu(A_{t+1}|S_{t+1}}...\frac{\pi(A_T|S_T}{\mu(A_T|S_T)}G_t$$

  Update value towards corrected return:

$$V(S_t) = V(S_t) + \alpha(G_t^{\pi/\mu} - V(S_t))$$

  In practice this is useless as it has a very high variance.

- So we need to use TD learning and Bootstrap. So now we only do importance sampling over one step. WE weigh TD target $R + \gamma v(S')$ by importance sampling and thus only need one sampling correction:

$$V(S_t) = V(S_t) + \alpha(\frac{\pi(A_T|S_t)}{\mu(A_t|S_t}(r_{t+1} + \gamma V(S_{t+1})) - V(S_t)$$

- The policy which works best for off-policy learning is Q-Learning. Here no importance sampling is required. The next action is chosen using behaviour policy A_{t+1} $\mu(.|S_t)$. But we consider alternative successor action A' $\pi(.|S_t)$ and update $Q(S_t, A_t)$ towards value of alternative action.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma q(S_{t+1}, a') - Q(S_t, A_t)$$

- When we talk about Q-learning(SARSAMAX), one usually means the following special case. We no allow both behaviour and target policies to improve. The target policy π is greedy w.r.t Q(s,a)

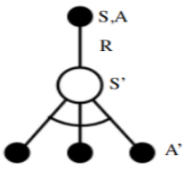
$$\pi(S_{t+1} = argmax_{a'} Q(S_{t+1}, a')$$

The behaviour policy μ is e.g. epsilon-greedy w.r.t Q(S,a).
The Q-learning target then simplifies:

$$R_t + \gamma Q(S_{t+1}, A')$$

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

Q-Learning Control Algorithm



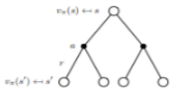
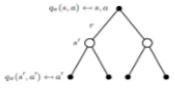
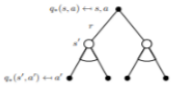
$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

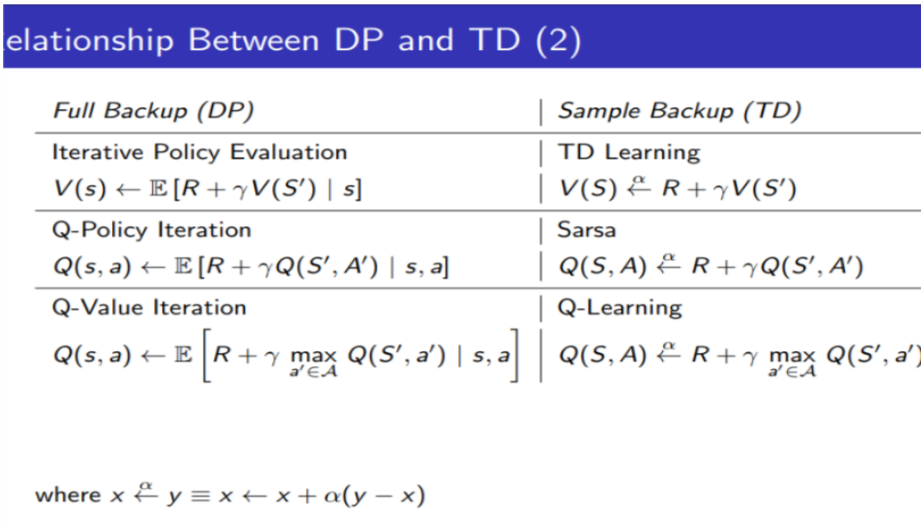
Theorem

Q-learning control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Relationship Between DP and TD

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 Q-Value Iteration	 Q-Learning



6 Value Function Approximation

- Today we will look at incremental methods and batch methods. With incremental methods you take a function approximator(like a NN) and incrementally every single step you see a new piece of data that comes in and immediately online you take some step to update your value function online. Batch methods look at everything you have seen so far and update the value function accordingly.

6.1 Incremental Methods

- The problem with real problems is that they might get arbitrarily large and we cannot just build a table anymore. Also we want to be able to generalize to different settings. So today we are going to look at how to compute value functions.
- So again we want to scale up prediction and control.
- So far we have represented value functions by a lookup table, i.e. every state or every state-action pair has an entry V(s) or Q(s,a). For control we need to be able to pick our actions without having a model(so we used Q(s,a). The problem with large MDP's is that there are too many states and/or actions to store in memory but also it is to slow to learn the value of each state individually.
- The solution to that is function approximation:

$$v'(s,w) \approx v_\pi(s)$$

or

$$q'(s,a,w) \approx q_\pi(s,a)$$

Note that our approximation works for any state s, even if we have not seen it yet. W are weights. This allows us to generalize to unseen states. We can update the parameters w using MC or TD learning.

- With value functions we can use different types of function approximators. You feed in your state s and can get $v'(s, w)$. You can feed in s and a and get $q'(s, a, w)$. We can also feed in a state s and get $q'(s, a_1, w)...q'(s, a_m, w)$
- There are a lot of different function approximators such as linear combinations of features, Neural Net's, Decision trees, Nearest Neighbours, Fourier/Wavelet based etc. We will focus on differential function approximators such as Linear combinations of features and Neural networks. Here its easy to update the weights with the gradient we can calculate. Furthermore, we require a training method that is suitable for non-stationary, non-iid data. Because the value function we try to approximate always changes depending on our changing policy and the rewards. It is non-iid because where I am now is very highly correlated to where I am going to be next.
- Lets have a look at SGD:

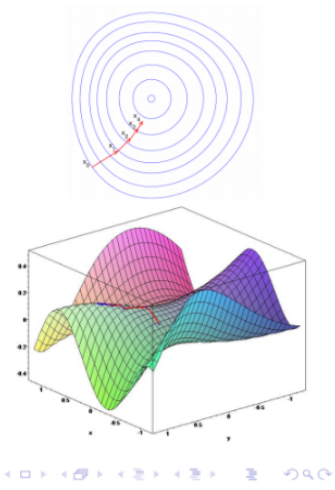
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the *gradient* of $J(\mathbf{w})$ to be

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

- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient

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

where α is a step-size parameter



Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

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

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update

Remember that the expectation is just averaging.

- So the idea is that we sample a state and then we compare our estimate for our value function with a true(oracle) value function. How can we do it without having the oracle knowledge?
- we often represent a state by a feature vector, for example the distance of a robot to landmarks or trends in the stock market etc.
- How does Linear Value Function approximation works? The idea is to use some weights and to use some state-feature instead of just a state. The nice thing is also that you only multiply by the state-features that are actually active. So for example if you don't have a queen, you won't multiply by the "queen-feature" of the state, because it is 0.

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S) \mathbf{w}_j$$

- Objective function is quadratic in parameters $\mathbf{w}$

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

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\begin{aligned} \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) &= \mathbf{x}(S) \\ \Delta \mathbf{w} &= \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S) \end{aligned}$$

Update = *step-size* $\times$ *prediction error* $\times$ *feature value*

- The connection to last week is that table lookup is a special case of linear value function approximation using table lookup features:

$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

- Parameter vector $\mathbf{w}$ gives value of each individual state

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

- Obviously we cheated, because we do not have an oracle true value function. But we can estimate the value function v_π.

$$\Delta\mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$

$$\Delta\mathbf{w} = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$$

- For TD(λ), the target is the λ-return G_t^λ

$$\Delta\mathbf{w} = \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$$

- Monte-Carlo with Value Function approximation: Return G_t is an unbiased, noisy sample of the true value $v_\pi(S_t)$. We can therefore apply supervised learning to the training data that we generate:

$$< S_1, G_1 >, < S_2, G_2 >, ... < S_T, G_T >$$

- The TD-target $R_{t+1} + \gamma v'(S_{t+1}, w)$ is a biased sample of the true value v_π. Here one can also build a data set and apply supervised learning on:

$$< S_1, R_2 + \gamma v'(S_2, w) >, < S_2, R_3 + \gamma v'(S_3, w) >, ...$$

- Remember that for the $TD(lambda)$ we can use a forward or backward view. The dataset looks like:

$$< S_1, G_1^\lambda >, < S_2, G_2^\lambda > ...$$

Forward view linear TD(lambda):

$$\Delta w = \alpha(G_t^\lambda - v'(S_t, w))\nabla_w v'(S_t, w)$$

$$= \alpha(G_t^\lambda - v'(S_t, w))x(S_t)$$

Backward view:

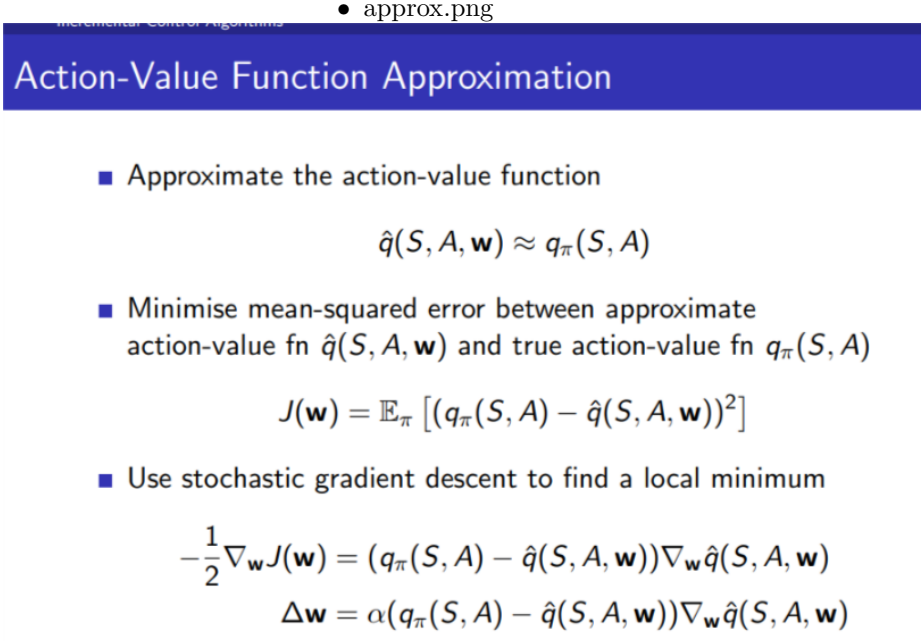
$$\delta_t = R_{t+1} + \gamma v'(S_{t+1}, w) - v'(S_t, w)$$

$$E_t = \gamma\lambda E_{t-1} + x(S_t)$$

$$\Delta w = \alpha\delta_t E_t$$

- How can we do Control with Value Function approximation? We are going to start with some weights and we start with some epsilon greedy policy. So we do:
  Policy evaluation: Approximate policy evaluation $q'(.,.,w) \approx q_\pi$
  Policy improvement: epsilon-greedy policy improvement.

- approx.png



Note that from there on we can do the same steps as before, use linear state-action feature vectors, TD(0), MC, TD(lambda) etc.

- Sometimes we should not use TD, because it could diverge.

Gradient Temporal-Difference Learning

- TD does not follow the gradient of *any* objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- Gradient TD follows true gradient of projected Bellman error

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

- Convergence of Control Algorithms:

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Chatter means that we take steps towards the goal, and then again we move away from the goal. And like that we kind of not get closer to the real goal.

6.2 Batch methods

- Gradient descent is simple and appealing but it is not very sample efficient. Batch methods seek to find the best fitting value function given the agent's (whole) experience(training data).
- Given value function approximation $v'(s, w) \approx v_\pi(s)$ and experience D consisting of ¡state, value¿ pairs(assuming we again have an oracle):

$$D0< s_1, v_1^\pi >, < s_2, v_2^\pi, ...$$

Which parameters w give the best fitting value function v'(s,w)? Least squares algorithm finds parameter vector w minimizing sum-squared error between our estimate of $v'(s_t, w)$ and the target values v_t^π.

$$LS(w) = \sum_{t=1}^{T} (V_t^\pi j - v'(s_t, w))^2$$

$$E_D[(v^\pi - v'(s, w))^2]$$

Note, before we always looked at one specific state which we were at in a specific timestep. Now we look at the whole data. There is an easy way to find the solution, its called experience replay.
1. Sample state, value from experience $< s, v^\pi$ from D 2. Do stochastic gradient descent update:

$$\Delta w = \alpha(v^\pi - v'(s, w))\nabla_w v'(s, w)$$

So note that here everything is decorrelated as we sample from it in a random fashion. The idea is also that you keep going over your data and squeze out everything you can from it, instead of throwing it away after one step.

- Deep Q-Networks: It uses experience replay and fixed Q-targets. Take action a_t according to epsilon reedy policy. We then store a transition $(s_t, a_t, r_{t+1}, s_{t+1}$ in replay memory D. We then sample a random mini-batch of transitions (s,a,r,s') form D and compute Q-learning targets w.r.t. old, fixed parametersw^-. We the optimize MSE between Q-networks and Q-learning targets:

$$L_i(W_i) = E_{s,a,r,s' \ D_i}[(r + \gamma max_{a'} Q(s', a', w_i^-) - Q(s, a, w_i))^2]$$

We use a variant of SGD.
Note that we have a giant NN which approximates Q. So the idea is that instead of having q-value from an oracle, you have an estimate of those from Q-learning(instead of SARSA as before) and we then approximate these.

- IMPORTANT NOTE: There is a key thing I missed until now. Remember, we estimate the value of a state for example with SARSA and then try to approximate this estimate with a Neural Net. Well this state value is computed as $R_{t+1} + \gamma v'(S_t, w)$. So we actually see one reward. But the estimate of the value of the next state is not calculated by trying every action or something similar, for the estimate of the next state we actually use our Neural Network that we are currently training with the old weights it had.

- Note that this algorithm is stable. Why? Because we do experience replay, i.e. we decorrelate our data. The second idea is that we keep two different Neural Net's with different parameters. We have our estimate of the state action value which is calculated with Q-learning. But if you look above, this gives us a reward plus the state-action value of the next state. This state-action value of the next state is computed with our Neural Network

which tries to approximate the estimate of our state-action value. And now while we try to calculate the gradient of the weights, we use the old Networks with old weights to approximate the state-action value of the next state. These old weights are fixed and only updated ex. every 1000 steps.

- For the Atari game, They did end-to-end learning of values Q(s,a) from pixels s. The Input state s was a stack of raw pixels from the last 4 frames. The output Q(s,a) for 18 joystick/button positions. The rewards is change in score for that step.
- Experience replay finds least squares solution but it may take many iterations. Using linear value function approximation we can solve the least squares solution directly.

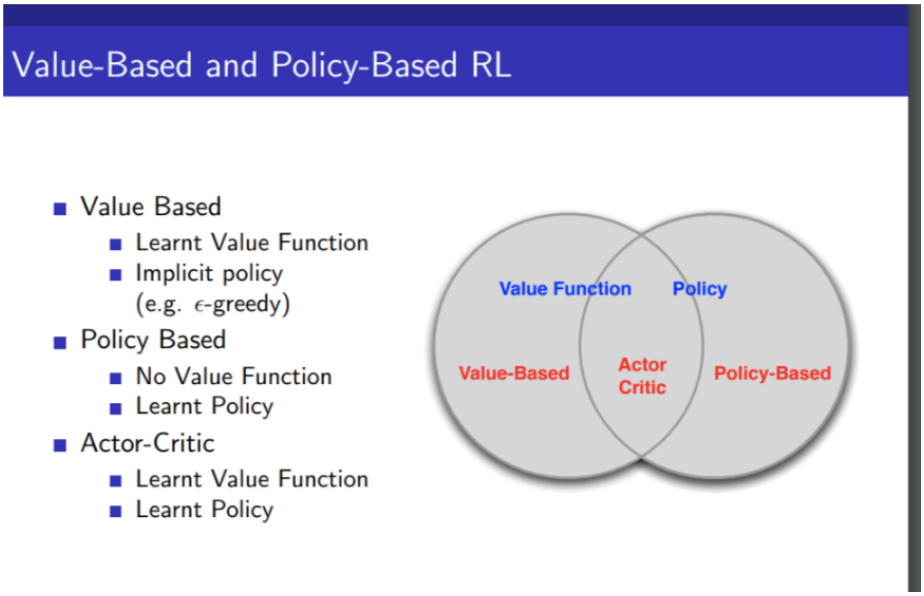
7 Policy Gradient Methods

- Today we will look at methods which optimize the policy directly. So today we will look at finite difference Policy Gradient, Monte-Carlo Policy Gradient and Actor Critic policy gradient. Last week we tried to estimate the action-value or value function. The policy was then generated directly from the value function using epsilon greedy. In this lecture we will directly parametrize the policy

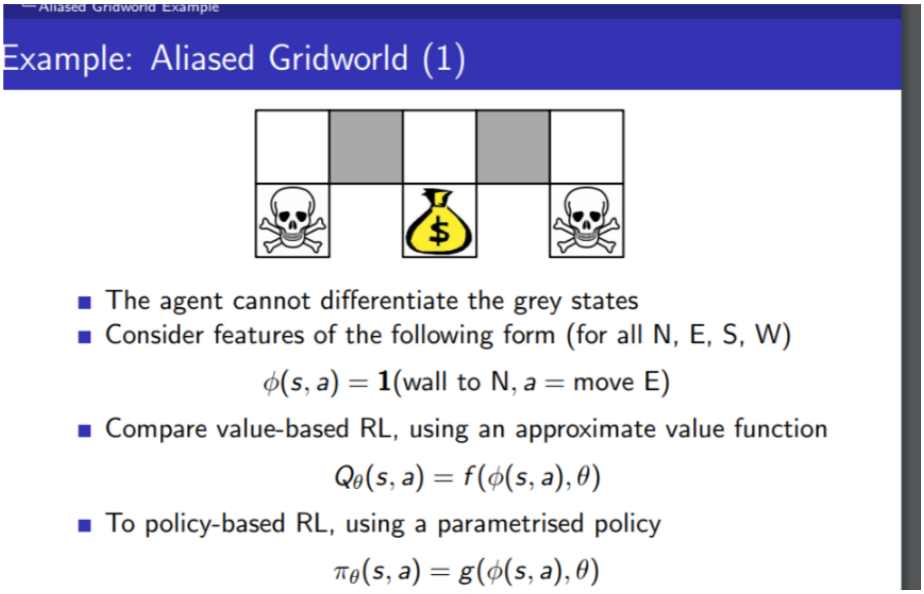
$$\pi_0(s, a) = P(a|s, a\theta)$$

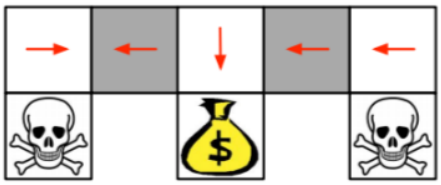
We will again focus on model-free reinforcement learning.

- The reason we do this function approximation is because we want to be able to scale to large, complicated MDP's. Where sometimes it is not possible for each distinct state to say I need to take this action or this action. That's why we need to do function approximation.

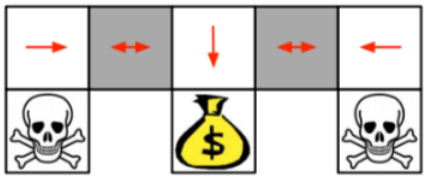


- What are some advantages of Policy Based RL? In some situations it is more efficient to store the policy instead of the value function. Better convergence properties and it is effective in high-dimensional or continuous action spaces(We do not have to compute a max). Also it can learn stochastic policies.
  Disadvantages: Typically it converges to a local rather than a global optimum. Evaluating a policy is typically inefficient and high variance.
- Why would we want a stochastic policy? Consider a two-player game of rock-paper-scissors. A deterministic policy can easily exploited.
- Also if we are in a partially observed MDP(i.e. we do not know all the transitions etc.), a deterministic policy might be bad. See this example:





- Under aliasing, an optimal deterministic policy will either
  - move W in both grey states (shown by red arrows)
  - move E in both grey states
- Either way, it can get stuck and *never* reach the money
- Value-based RL learns a near-deterministic policy
  - e.g. greedy or ϵ-greedy
- So it will traverse the corridor for a long time



- An optimal stochastic policy will randomly move E or W in grey states

$$\pi_\theta(\text{wall to N and S, move E}) = 0.5$$
$$\pi_\theta(\text{wall to N and S, move W}) = 0.5$$

- It will reach the goal state in a few steps with high probability
- Policy-based RL can learn the optimal stochastic policy

- There are different objective Functions.

Policy Objective Functions

- Goal: given policy $\pi_\theta(s,a)$ with parameters θ, find best θ
- But how do we measure the quality of a policy π_θ?
- In episodic environments we can use the start value

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

- In continuing environments we can use the average value

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- Or the average reward per time-step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) \mathcal{R}_s^a$$

- where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ

Fortunately for us the same methods apply to all three objectives. The only thing that differs is th distribution that I am in some state s.

- Now we want to optimize the policies(i.e. the theta). So find θ that maximizes $J(\theta)$. Some approaches do not use gradient such as Hill climbing, Simplex or Genetic algorithms. Greater efficiency is often possible using gradients such as gradient descent, conjugate gradient or quasi-newton. We focus on gradient descent.

7.1 Finite Difference Policy Gradient

- We have an objective function and we try to search for a maximization of this function. Thus we want to do gradient ascent.
- A very naive approach to evaluate policy gradient of $\pi_0(s,a)$ is to just perturb our parameters θ by a small amount epsilon in the k-th dimension(to estimate k-th partial derivative). We do this then for each dimension(which is ineffective). But this works for arbitrary policies, even if the policy is not differentiable.

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k - J(\theta)}{\epsilon}$$

- We now compute the policy gradient analytically. Assume the policy is differentiable whenever it is non-zero and we know the gradient $\nabla \pi_0(s,a)$.Likelihood ratios exploit the following identity of the log:

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}$$

$$= \pi_\theta(s,a) \nabla_\theta log \pi_\theta(s,a)$$

Now the score function is $\nabla_\theta log \pi_\theta(s,a)$

- We will us a a softmax policy as a running example. It is basically something where we want to have some smoothly parametrized policy that tells us how frequently we should choose an action for each discrete set of actions. So its an alternative to epsilon-greedy etc. Weight actions using linear combination of features $\phi(s,a)^T\theta$. The probability of an action is proportional to exponentiated weight:

  $$\pi_\theta(s,a) \propto e^{\phi(s,a)^T\theta}$$

  The score function is then

  $$\nabla_\theta log \pi_\theta(s,a) = \phi(s,a) - E_{\pi_\theta}[\phi(s,.)]$$

  For example in atari, we would have some features for going left and some for going right and whichever scores higher would get higher probability when we pick these actions. We would like to calculate the gradient of this thing. Note that the gradient is just the feature minus the average feature. So if a feature occurs more than usual and gets a good reward, we want to change the policy. Note that the expectation is the normalization constant.

- in continuous action spaces, a Gaussian policy is natural. The mean is a linear combination of state features $\mu(s) = \phi(s)^T\theta$. The variance may be fixed or can also be parametrized. The policy is a Gaussian, $a \sim N(\mu(s), \sigma^2$. The score function is then

  $$\nabla_\theta log \pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}$$

7.2 Monte Carlo Policy Gradient/REINFORCE

- Our goal is to maximize our reward. So to maximize our expected reward, we need to move in the direction of our score-function times our reward. If you have positive or negative reward you move into opposite directions.

One-Step MDPs

- Consider a simple class of one-step MDPs
  - Starting in state $s \sim d(s)$
  - Terminating after one time-step with reward $r = \mathcal{R}_{s,a}$
- Use likelihood ratios to compute the policy gradient

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\ &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \mathcal{R}_{s,a} \\ \nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r] \end{aligned}$$

- What do we do, if we want to apply this trick not only to one step MDP's but to multiple steps. We have to replace the instantaneous reward r with a long-term value. $Q^\pi(s,a)$

Theorem

For any differentiable policy $\pi_\theta(s,a)$,
for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$,
the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\ Q^{\pi_\theta}(s,a)]$$

- REINFORCE/ Monte-Carlo Policy Gradient:

Monte-Carlo Policy Gradient (REINFORCE)

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

That means we are in a state, we take an action and see what rewards we get from there on wards and use this as an estimate of Q. So we basically sample a whole episode.

- With this method we can see that the learning curve is very smooth, but also very slow. So in the rest of this class we will try to make it more efficient.

7.3 Actor Critic Methods

- Monte-Carlo policy gradient still has high variance. We use a critic to estimate the action-value function,

$$Q_w(s, a) \sim Q^{\pi_\theta}$$

- actor-critic algorithms maintain two sets of parameters: The critic which updates action-value function parameters w and the actor which updates policy parameters θ in direction suggested by the critic.

- actor-critic algorithms follow an approximate policy gradient:

$$\nabla_\theta J(\theta) \sim E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) Q_w(s, a)]$$

$$\Delta\theta = \alpha \nabla_\theta log \pi_\theta(s, a) W_w(s, a)$$

- The critic is solving a familiar problem: policy evaluation. So how good is a policy for the current parameters. This problem was explored in previous lectures with Monte-Carlo policy evaluation, TD learning or TD(lambda). But note, we need algorithms which solve the prediction, not the control(Q-learning) problem.

Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s,a) = \phi(s,a)^\top w$
  - Critic Updates w by linear TD(0)
  - Actor Updates θ by policy gradient

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

- So the difference between Policy iteration and actor critic, is that here we do not do greedy policy update but we follow the policy gradient.
- approximating the policy gradient introduces bias. A biased policy gradient may not find the right solution. Luckily if we choose value function approximation carefully, we can avoid introducing any bias.
- There are several tricks to reduce the variance.

Reducing Variance Using a Baseline

- We subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing expectation

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] &= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a) B(s) \\ &= \sum_{s\in\mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(s,a) \\ &= 0\end{aligned}$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- So we can rewrite the policy gradient using the advantage function $A^{\pi_\theta}(s,a)$

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]$$

We see that adding the B(s) term does not change the Expectation. The state advantage function tells us how much better is it than usual to take action a.

- The advantage function can significantly reduce variance of policy gradient method. So the critic shoudl really estimate the advantage function. For example by estimating both V^{π_θ} and $Q^{\pi_\theta}(s,a)$. using two function approximators and two parameter vectors:

$$V_v(s) \sim V^{\pi_\theta}(s)$$

$$Q_w(s,a) \sim Q^{\pi_\theta}(s,a)$$

$$A(s,a) = Q_w(s,a) - V_v(s)$$

And update both value functions by e.g. TD learning.

- But there is an easier way:

Estimating the Advantage Function (2)

- For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

- is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta}|s,a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s')|s,a\right] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s,a)\end{aligned}$$

- So we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta^{\pi_\theta}\right]$$

- In practice we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- This approach only requires one set of critic parameters v

The advantage of that is that we only need to estimate V and not also Q.

- a critic can estimate value function from many targets at different timescales. So we had MC, TD(0) and forward or backward TD(lambda). So first of all we can apply that to our critic, any evaluation algo is valid for the critic.

Critics at Different Time-Scales

- Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales From last lecture...
  - For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

  - For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

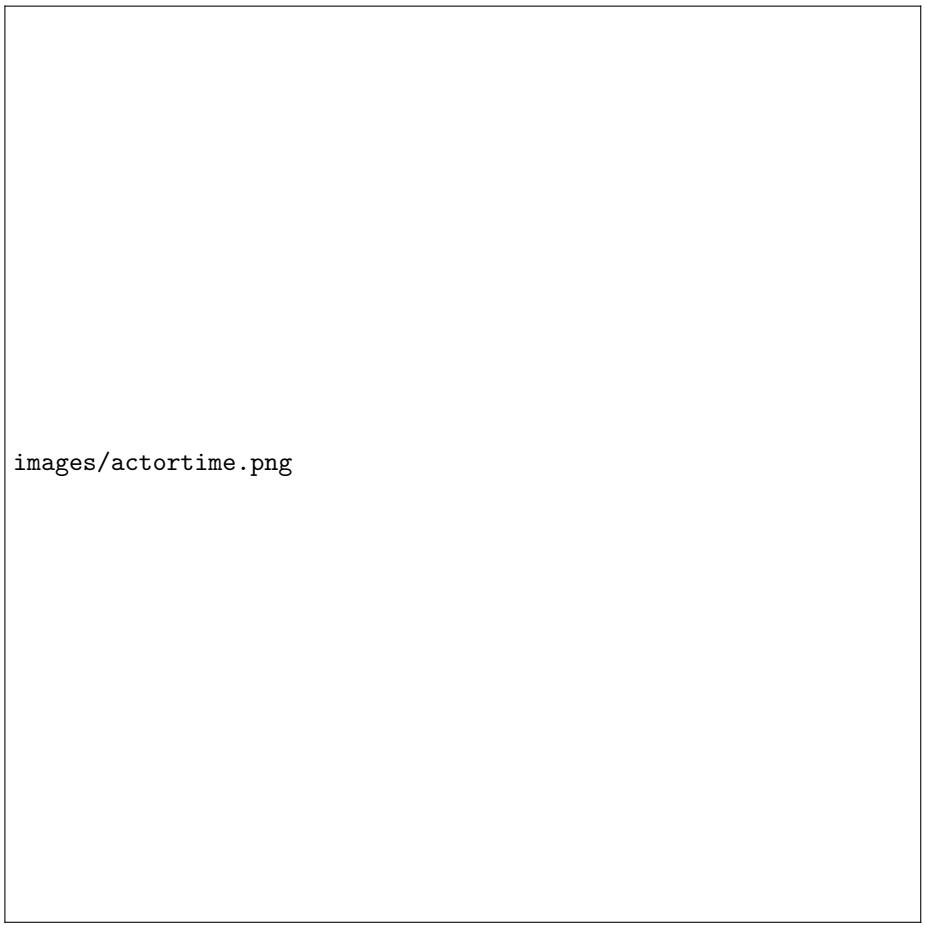
  - For forward-view TD(λ), the target is the λ-return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

  - For backward-view TD(λ), we use eligibility traces

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$
$$e_t = \gamma\lambda e_{t-1} + \phi(s_t)$$
$$\Delta\theta = \alpha\delta_t e_t$$

But we can also apply these tricks to the actor.



The MC actor policy has low bias but high variance. The TD has high bias, low variance. To have the best of both world, we use Eligibility traces.

Policy Gradient with Eligibility Traces

- Just like forward-view TD(λ), we can mix over time-scales

$$\Delta\theta = \alpha(v_t^\lambda - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- where $v_t^\lambda - V_v(s_t)$ is a biased estimate of advantage fn
- Like backward-view TD(λ), we can also use eligibility traces
  - By equivalence with TD(λ), substituting $\phi(s) = \nabla_\theta \log \pi_\theta(s, a)$

$$\begin{aligned} \delta &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \\ e_{t+1} &= \lambda e_t + \nabla_\theta \log \pi_\theta(s, a) \\ \Delta\theta &= \alpha\delta e_t \end{aligned}$$

- This update can be applied online, to incomplete sequences

- There is an important question in actor critic algorithms. Gradient ascent algorithms can follow any ascent direction A good ascent direction can significantly speed convergence. Also a policy can often be reparametrized without changing action probabilities. For example increasing score of all actions in a softmax policy. The vanilla gradient is sensitive to these reparametrizations. So in spite the fact that we approximate the true value function, we can still find the true value function.
- Here a summary:

Summary of Policy Gradient Algorithms

- The policy gradient has many equivalent forms

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ v_t\right] && \text{REINFORCE} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ Q^w(s, a)\right] && \text{Q Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ A^w(s, a)\right] && \text{Advantage Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ \delta\right] && \text{TD Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic} \\ G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic} \end{aligned}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s, a)$, $A^\pi(s, a)$ or $V^\pi(s)$

8 Integrating Learning and Planning

- We will have a look at model-based RL, Integrated architectures and simulation-Based Search. In Model Based RL we will especially see how one can plan better by having a model. Integrated architectures give us the best of both worlds, model based and no model RL.
- Before we looked at learning a policy or a value function directly from experience. Today we try to learn a model directly from experience and use planning to construct a value function or policy.
- A model in our context is an idea of an MDP.
- Model-Free RL: NO model, we learn a value function from experience. Model based RL means that we learn a model from experience and plan a value function (and/or policy) form the model.

8.1 Model Based RL

- We have a value/policy on which we act to gather experience. From that we learn a model and then plan on this model to learn the value/policy.
- Advantages: Can efficiently learn model by supervised learning methods and can reason about model uncertainty. For example in chess, we can have the rules of the game as a model which allows us to do a tree search of the next steps.
- Disadvantages: First learn a model, then construct a value function which introduces two sources of approximation error.
- What is a mode? It is a representation of an MDP ¡S,A,P,R¿ parametrized by η. We will assume that state space S and action space A are known. So a model $M =< P_\eta, R_\eta >$ represents state transitions $P_n \sim P$ and rewards $R_n \sim R$:

$$S_{t+1} \sim P_n(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = R_n(R_{t+1}|s_t, A_t)$$

  Typically we assume conditional independence between state transitions and rewards.
- For chess for example the model learning would be to learn the rules of the game. Given the rules we want to plan.
- Goal is to estimate the model M_n from experience S_1, A_1, R_2 etc. This is a supervised learning problem, because we have a dataset which tells us that if we are in S_1, A_1 we will get R_2 and land in S_2.
- So learning the reward from a state action pair is a regression problem and learning the next state from a state action pair is a density estimation problem. The loss function can be MSE(regression) and for the second we could take KL-divergence. We want to find η to minimize the empirical loss.

- Note that often the state space is not given or we might want to learn state features.
- We will look at the table lookup model in which we have a state and an action and try to predict the reward and next state seperately. Then there are also linear expectation models, linear gaussian models, gaussian process models, Deep Belief Netowrk Model.
- The model is an explicit MDP, P', R'. We count visits N(s,a) to each state action pair:

$$P'_{s,s'}{}^{a} = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t, S_{t+1} = s, a, s')$$

$$R_s^a = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t = s, a) R_t$$

  Alternatively at each time-step t, we can record experience tuple. To sample model, randomly pick tuple matching ¡s,a,.,.¿.
- Once we have a model we can solve it/plan with our favourite planning algorithm such as value iteration, policy iteration, tree search etc.
- Sample based planning is also very powerful. We use the model only to generate samples, i.e. we sample experience from the model:

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$

$$R_{t+1} = R_\eta R_{t+1}|s_t, A_t$$

  and to these samples we apply model-free RL such as Monte-Carlo control, Sarsa, Q-learning etc. The advantage of this approach is that we can sample infinite data.
- Given an imperfect model, we might not be able to learn anything useful. So our performance is limited to the optimal policy of approximate MDP. So the solution is that when the model is wrong we should use model-free RL or reason explicitly about our model uncertainty.

8.2 Integrated Architectures

- We consider two sources of experience: First of all we have real experience samples from the environment(true MDP).

$$S' \sim P^a_{s,s'}$$

$$R = R^a_s$$

  And then we also have simulated experience from the model(approximate MDP):

$$S' \sim P_\eta(S'|S, A)$$

$$R = R_\eta(R|S, A)$$

- Dyna-Architecture. WE learn a model from real experience and learn and plan our value function (and/or policy) from real and simulated experience. That means from our experience we do direct RL and we do model learning and planning from that.

Dyna-Q Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

- We can also optimize the algorithm such that it explores more.

8.3 Simulation-Based Search

- Here we look the idea of taking samples from imaginary experience. We are going to use sampling and forward search. tem Forward search algorithms select the best action by lookahead. They build a search tree with the current state at the root. Using a model of the MDP to look ahead. We do not need to solve the whole MDP, we just need to solve parts of it starting from now.
- Simulation based search is a forward search paradigm using sample-based planning. That means we simulate episodes of experience from now with the model. WE apply model free RL to simulated episodes. The reason why it is forward is because we always sample from the current state on. Once we get these trajectories we apply model-free RL.
- We simulate episodes of experience from now with the model:

$$s_t^k, A - t^k, r_{t+1}^k, ..., S_T^k \sim M_v$$

Note that we apply for example Monte-Carlo control which is then called Monte Carlo search, or if we apply Sarsa it is called TD search.

- MOnte-Carlo Search: Given a model M_v and a simulation policy π, for each action we simulate K episodes from current(real) state s_t:

$$s_t, a, r^k_{t+1}, A^k_{t+1}, ..., S^k_T{}_{k=1}^K \sim M_{v,\pi}$$

We then evaluate the actions by the mean return(Monte-Carlo evaluation):

$$Q(s_t, a) = \frac{1}{k}\sum_{k=1}^{K} G_t -> q_\pi(s_t, a)$$

We select the current(real) action with maximum value:

$$a_t = argmax_{a\epsilon a} Q(s_t, a)$$

So for example if I can go left, I simulate 100 times going left and take the average empirical expected reward.

- Here is another thing that works nicely in practice(Alpha GO).

Monte-Carlo Tree Search (Evaluation)

- Given a model $\mathcal{M}_\nu$
- Simulate K episodes from current state s_t using current simulation policy π

$$\{s_t, A^k_t, R^k_{t+1}, S^k_{t+1}, ..., S^k_T\}^K_{k=1} \sim \mathcal{M}_\nu, \pi$$

- Build a search tree containing visited states and actions
- Evaluate states $Q(s, a)$ by mean return of episodes from s, a

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(S_u, A_u = s, a) G_u \xrightarrow{P} q_\pi(s, a)$$

- After search is finished, select current (real) action with maximum value in search tree

$$a_t = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s_t, a)$$

We then use the information in our tree to make the policy better.

Monte-Carlo Tree Search (Simulation)

- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximise $Q(S, A)$
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by $\epsilon - \text{greedy}(Q)$
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$

- Advantages of MC Tree Search. it is highly selective best-first search. It evaluates states dynamically unlike e.g.DP. It uses sampling to break the curse of dimenstionality. it works for black-box models(it only requires samples. Also it is computationally efficient, anytime and parallelizable.
- There is also temporal-difference search. It is also simualation-based search. But we use TD instead of MC. MC tree search applies MC control to sub-MDP from now. TD search applies Sarsa to sub-MDP from now.
- For model-free RL, bootstrapping is helpful. TD learning reduces variance but increases bias. TD learning is usually more efficient than MC. TD(lambda) is much more efficient than MD.For simulation-based search it does the same!

- Simulate episodes from the current (real) state s_t
- Estimate action-value function $Q(s, a)$
- For each step of simulation, update action-values by Sarsa

$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

- Select actions based on action-values $Q(s, a)$
  - e.g. ϵ-greedy
- May also use function approximation for Q

- Remember for both searches, we can also use Functon approximators.
- Dyna-2: the agent stores two sets of feature weights: long-term and short-term memory. The Long term memory is updated from real experience using TD learning. The short term memory is updated from simulated experience using TD search.

sectionExploration and Exploitation

- We will have a look at multi-armed bandits and contextual bandits and lastly MDP's. Online decision making involves a fundamental choice: Exploitation(make the best decision given current information) or exploration(gather more information to get more reward). The best long-term strategy may involve short-term sacrifices to gather enough information to make the best overall decisions.
- We will look at three different exploration techniques. The first is to do random exploration(epsilon-greedy). The next is to use optimism in the face of uncertainty. There you estimate uncertainty on the value and prefer to explore states/actions with highest uncertainty. The last one is information state space usage, that is one considers agent's information as part of its state. We then do look ahead to see how information helps the reward. So you can say I am in a state where I have tried left three times and right one time. We can then ask how good it is to move into a state where the agent has no or some information.
- There are two types of exploration. There is state-action exploration which tries to systematically explore the state or action space. So we know something about the states and explore based on that. There is also parameter exploration, it parametrizes a policy by a parameter. One could then try different parameters which influence the policy. The advantage of that is we have consistent exploration and the disadvantage is that one doesn't know about the state/action space exploration. So here the exploration part is just that we try different parameters to see how it works.

8.4 Multi Armed bandit

- It's a simplification of the MDP problem. It is a tuple ¡A,R¿ where a is a known set of actions. $R^(r) = P[R = r|A = a]$ is an unknown probability distribution over rewards. At each step the agent selects an action and the environment generates a reward. The goal is to maximize the cumulative reward. So the difference is that we throw away the whole state space and the transitions.
- The action-value is the mean reward for action a,

$$q(a) = E[R|A = a]$$

. The optimal value v* is

$$v* = q(a*) = max_{a\epsilon A} q(a)$$

The regret is the opportunity loss for one step:

$$l_t = E[v* - q(A_t)]$$

The total regret is the total opportunity loss:

$$E[\sum_{\tau=1}^{t} v* - q(a_\tau]$$

. The goal is ot maximize the cumulative reward or to minimize the total regret.

- The count $N_t(a)$ is the expected number of selections for action a. The gap is the difference in value between action a and an optimal action a*: $\delta_a = v* - q(a)$
- Regret is a function of gaps and the counts

$$L_t = E[\sum_{\tau}^{t} v* - q(A_\tau]$$

$$\sum_{a \epsilon A} E[N_t(a)](v* - q(a))$$

$$= \sum_{a \epsilon A} E[N_t(a)]]\delta_a$$

A good algorithm ensures small counts for large gaps. The problem though is that the gaps are not known.

- If an algorithm forever explores it will hit linear total regret. If an algorithm never explores it will have linear total regret.
- We consider algorithms that estimate $Q_t(a) \sim q(a)$. We estimate the value of each action by Monte-Carlo evaluation:

$$Q_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} 1(A_t = a)R_t$$

So that means just the mean reward of each action. The greedy algorithm selects the action with the highest value:

$$A_t = argmax Q_t(a)$$

. Greey can lock onto suboptimal action forever and thus the total regret is linear.

- A first improvement would be to initialize values to the maximum possible, i.e.$Q(a) = r_{max}$. Then act greedily

$$a_t = argmax_{a \epsilon A} Q_t(a)$$

This encourages exploration of unknown values, but a few unlucky samples can lock out optimal action forever and thus it also has linear total regret. So the way it actually works is that it is just some kind of bias. We say we have already seen the optimal value a hundred times and then we do the same as before(MC) from there on.

- The epsilon greedy algorithm continues to explore forever. The constant epsilon ensures a minimum regret:

$$I_t >= \frac{\epsilon}{|A|} \sum_{a \epsilon A} \delta_a$$

. Thus epsilon greedy has a linear total regret. But if we pick a decay schedule for epsil on we have logarithmic asymptotic regret.

- So the goal is to find an algorithm with sublinear regret for any multi-armed bandit without knowing the rewards beforehand.

- A hard bandit problem is one with similar distributions but different means. Asymptotic total reward is at least logarithmic in number of steps.

- We will use upper confidence bounds for each action value pair. Such that $q(a) <= Q_t(a) + U_t(a)$ with high probability. This dpends on the number of times N(a) has been selected. A small N(a) implies a large U(a) (estimated value is uncertain). a large N(a) implies a small U(a) (estimated value is accurate. We then pick the action maximizing the UCB:

$$A_t = argmax Q_t(a) + U_t(a)$$

- We apply Hoeffdings inequality to our Upper bound. This gives us:

$$U_t(a) = (\frac{-logp}{2N_t(a))}^{1/2} = \frac{2logt}{N_t(a)}^{1/2}$$

- This leads to the UCB1 algorithm:

$$a_t = argmax Q_t(a) + (\frac{2logt}{N_t(a)})^{1/2}$$

which has logarithmic asymptotic regret.

- Bayesian bandits exploit prior knowledge about rewards. So we consider a distribution over action value functions with parameter w.We then use bayesian methods to compute posterior distributions over w. We can use this to guide the exploration. Either use it as UCB bound or try to do probability matching which selects an action a according to the probability that a is the optimal action. So for distributions that we do not know a lot about, there is a high chance that an action could be the max for that one.

- What is asymptotically optimal is to sample from your distributions, take the one that is maximal and use that action. That is called Thompson sampling.

8.5 Information Exploration

- So far we have seen 2 of three methods, randomized exploration and UCB algorithms.

- Exploration is useful because it gains information.Can we quantify the value of information? Can we quantify the long term gain of exploring a trejectory? How much reward a decison-maker would be prepared to pay in order to have that information, prior to making the decision. The long-term reward after getting information - immediate reward. Information gain is higher in uncertain situations. Therefore it make ssense to explroe uncertain situations more. So the tradeoff of information depends on how much budget we have. if we know the value of information, we can trade-off exploration and exploitation optimally.
- We have viewed bandits as on-step decision-making problems. But it can also be viewed as a sequential decision-making problems. At eacht step there is an information state s which summarizes ll information accumulated so far. Each action A causes a transition to a new information state .
- So lets assume a bernoulli bandit such that $R = B(\mu_a)$.We want to find the arm which ahs the highest μ_a. The information state $S =< \alpha, \beta >$ with α counts the pulls of arm a where reward was 0 and β counts the pulls of arm a where reward was 1.
- One can do a look ahead ex. for drugs. If it succeeds I would change my success distribution in a certain way and if I fail in another. So we have an infinite MDP over information states. This MDP can be solved by reinforcement learning.
- We begin with a $Beta(\alpha, \beta)$ prior over the reward function R. Each time action a is selected, we update posterior R: $Beta(\alpha + 1, Beta)$ if reward is 0 and otherwise if reward is 1. Each information state corresponds to a model beta.
- A contextual bandit uses a state. The reward is then dependent on the state and the action. We can estimate the value function with linear function approximators.
- How can we use the things learned today in the full MDP problem. The UCB approach can be generalized so can all the others. For the UCB you would take the Q-value of a state action pair and add some uncertainty part to it. But this ignores that in an MDP we will start to improve our policy. So we shoudl also take into account how much we could improve our policy .
- One successful approach to exploration/exploitation in model-based RL is to construct a model of the MDP. For unknown or poorly estimated states, we replace the reward function with the r-max value. That is we are very optimistic in the face of uncertainty and solve optimistic MDP by our favorite planning algorithm(tree search, policy iteration, value iteration etc.). This algo is called Rmax. The information based method can also be applied to MDP's.

References