# Some Common Considerations in RL

## Step size

* + 1. In the tabular case, a step size of alpha=1/tau suggest that it would take about tau experiences for a state to converge approximately to the mean of the target
    2. With function approximation, if we want to learn in about tau experiences, a general rule of thumb for setting the step-size parameter is  
        where x is a random feature vector chosen from the same distribution as input vectors. The step size is thus one over tau times the expected variance of input

## Reward

* + 1. Why might we want to change the reward function?
       1. Make the problem more efficient to solve, in both time and space
       2. Make the problem solvable
    2. Reward-modification strategy
       1. Multiplying by a scalar, i.e. R’ = cR, and the resulting Q’ = cQ
       2. Adding a scalar, i.e. R’ = R + c, and the resulting Q’ = Q + c/(1-gamma)
    3. Potential-based (reward) shaping
       1. We rewrite R as below  
           And now Q becomes

Phi(s) is a shaping function, which maps each state to a real value (potential)

The result Q’(s, a) = Q(s, a) - \Phi(s) is quite intuitive that since we map each state to a value, we remove the value of the state from our estimated utility when we leave a state

Note, potential-based shaping doesn’t change the original policy

* + - 1. Why do we define potential other than reward?
         1. Reward is accumulated along the way, which doesn’t get diminished (unless the agent receives negative reward).  
            Potential, on the other hand, is the temporal reward the agent receives when entering a state, which will be removed when the agent leaves the state
      2. Potentials are based on the knowledge of states in the environment. They help speed up reinforcement learning. The effect of potential-based shaping is equivalent to initializing the Q-value with potentials.
         1. In a tabular setting with a fixed potential function, initialization is feasible and might be simpler. But it gets tricker with function approximation and impossible with potential functions that change over time
         2. Potential-based reward shaping still provides the only Bellman-consistent mechanism of combining value functions

## Convergence conditions

* + 1. Greedy in the Limit of Infinite Exploration (GLIE)

A type of scheme which says

1. The agent should try each action in each state enough times to get an unbiased sense of the true model

2. The policy converges on a greedy policy

* + - 1. A sensible GLIE scheme gives some weight to action that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility
    1. Step-size parameter alpha
       1. Step-size parameter alpha should satisfy the following two conditions to assure convergence with probability 1 and

The first condition guarantees that the steps are large enough to eventually overcome any initial conditions or random fluctuation.

The second condition guarantees that eventually the step size becomes small enough to assure convergence

* + - 1. Constant alpha doesn’t meet the second condition, which indicates that the estimates never completely converge but continue to vary in response to the most recently received rewards. This is, however, actually desirable in a non-stationary environment.

To see that constant step-size parameter makes the action-value function continue to vary in response to the most recently received rewards. We expand the action-value function assuming alpha is constant

From which we can see

1. The weight given to R{i} decays exponentially according to the exponent on 1-alpha as time moves on.

2. The total amount of weight is (1-alpha)^n+∑alpha(1-alpha)^(n-i)=1

3. The most recent reward will always have a constant weight.

* + - 1. Sequences of step-size parameters that meet the above two conditions often converges very slowly or need considerable tuning in order to obtain a satisfactory convergence rate, which makes them often used in theoretical work, but seldom in applications and empirical research
    1. In practice, we usually neglect either GLIE or step-size conditions, and the algorithms still converge anyway.

## Strategies to balance tradeoff between exploration and exploitation

* + 1. Following methods more or less suffer the problem that it is difficult to discover alternative course of action that extend far into the future
       1. Epsilon-greedy policy  
          simply follows a greedy policy with probability 1-epsilon and takes a random action with probability epsilon

Epsilon generally starts with 1 and decays as the training proceeds. It will eventually stop at some minimum value

* + - * 1. Proof that policy improvement actually improves epsilon-greedy policy

Note that

and

so the weighted average Q^pi

must be less than or equal to the largest Q^pi

* + - * 1. Epsilon-soft policy

Where

For all states and actions

* + - 1. Softmax with temperature  
          tau is the temperature coefficient, as in simulated annealing, it’s annealed over the course of training. A high temperature causes all actions to be equiprobable, while a low temperature skews the probability toward a greedy policy
      2. Upper-confidence-bound (UCB)  
          N(s. a) denotes the number of times that action a has been selected at state s — if N(s, a)=0, then a is considered to be a maximizing action. c>0 controls the degree of exploration.

The idea of the upper confidence bound (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a’s value. The quantity being max’ed over is thus a sort of upper confidence bound on the possible true value of action a, with c determining the confidence level. Each time a is selected the uncertainty is presumably reduced: N(s, a) increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but

N(s, a) does not; since t appears in the numerator, the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

* + - * 1. The square-root term is obtained from the [Hoeffding's inequality](https://en.wikipedia.org/wiki/Hoeffding%27s_inequality)  
            Where n is number of samples used to compute the mean of X  
           Now let the exponential part equal to p, the upper bound of probability that real action value is greater than the confidence bound, then we have  
            By making p=t^{-2} (we make the upper bound U so that the probability of expected Q being greater than Q plus the upper bound U decays as time goes by), it ends up the square-root term in UCB
      1. Gradient Bandit Algorithm  
         We define a numerical preference for each action a, H(a). The action selection policy is a softmax  
          At each step, after selecting action a and receiving reward R, all the action preferences get updated by  
          where \bar R{t}, the average of all the rewards up through and including time t, serves as a baseline with which the reward is compared. The last weird term which contains pi is derived from the partial derivative of the policy, pi(a), w.r.t. the action preference, H(a)

Intuition behind GBA is given [here](https://stats.stackexchange.com/questions/343283/rl-gradient-bandit-algorithm-whats-the-intuition-behind-mathbb-1-a-a-pi)

## Bias and variance

[bias-variance trade-off](https://medium.com/mlreview/making-sense-of-the-bias-variance-trade-off-in-deep-reinforcement-learning-79cf1e83d565)

* + 1. In RL, a value estimate is biased if its expectation deviates from the real return. On the other hand, if a value estimate is of high variance, it’s noisy but on average accurate.   
       MC is unbiased in that it computes the expectation of actual rollouts. It is, however, of high variance due to the return of each trajectory may differ significantly.  
       TD is highly biased because it relies on the value estimate for the next state, which is never completely accurate. On the other hand, it’s of low variance, since there’s much less stochasticity in the reward signal

## What’s the most important feature distinguishing reinforcement learning from other types of learning?

* + 1. Trial-and error: the agent must discover which actions yield the most reward by trying them (interacting with the environment)
    2. Delayed reward: the reward of taking an action may not be seen immediately, instead it may emerge in the future. This allows the agent to think in long term
    3. RL uses training information that evaluates the action taken rather than instructs by giving correct actions. This is what creates the need for active exploration, for an explicit search for good behavior

Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible.

Purely instructive feedback indicates, on other hand, indicates the correct action to take, independent of the action actually taken (supervised learning)

# Multi-Armed Bandits

## Concept

* + 1. Bandit
       1. <A, R>

Non-associative, non-stationary

* + - 1. The incremental update rule  
          Where the step-size parameter alpha can be any function resulting in (0, 1]. The error term, (R{n+1}-Q{n}), indicates how the current reward adjusts the original action-value.

In the book, it uses R{n} instead of R{n+1}, which suggests reward obtained after taking action at time-step n. Since we don’t specify action here, I think it will be clearer to use R{n+1}, which stands for reward at time-step n+1

* + 1. Regret

The opportunity loss for one step

Where V\* is the maximum reward

* + - 1. Count

N(a), expected number of selections for action a

* + - 1. Gap

Delta a, the difference in value between action a and optimal action a\*

* + - 1. Total regret

The total opportunity loss

* + - 1. Maximize cumulative rewards = minimize total regret
    1. Optimistic initial values

A technique for encouraging exploration, in which the initial values are equal to the maximum possible action-values

* + - 1. The agent may still need to do some exploration, otherwise, it would get stuck at local optimal for a few unlucky samples.   
         A way to circumvent this issue is to sample a certain amount of times (n), updating the value using the mean of the samples, and then acting greedily. In this way, we gain some confidence with adequate large n
      2. It is not well suited to non-stationary problems because its drive for exploration is inherently temporary.

## Algorithm

* + 1. General pattern for bandit algorithm
       1. Initialize variables  
          While true  
           a = select\_action()  
           r = R(a)  
           update variables accordingly

select\_action() varies from algorithm to algorithm

variables serve to help select action

# Markov Decision Process

## Markov Process

* + 1. <S, P>
    2. The transition probability matrix, P is a matrix where P[i, j] defines the transition probability from S[i] to S[j]. Each row of the matrix sums to 1

## Markov Reward Process

* + 1. <S, P, R, gamma>

The reward is our way of communicating to the agent what we want it to achieve, not how we want it to achieve

* + 1. Define the return as the total discounted reward from time-step t  
        or

Both share the same meaning, the difference is introduced by different definition of reward

The first one uses the reward definition that says we leave state S(t), receiving reward R(t+1). This one is preferred in some cases since it leaves R(0) undefined, and thus we can slip in initial values without changing the general definition of R

The second uses the reward definition that says we enter state S(t), immediately receiving reward R(t). This is pretty intuitive in that a reward has the same timestamp as its corresponding state

* + 1. The Bellman equation

We haven’t introduced action yet, so the Bellman equation is a linear equation for the time being

* + - 1. The Bellman equation is defined
      2. Since the Bellman equation is a linear equation, it can be solved directly:  
          The computational complexity is O(n^3) for n states, which is pretty high for large MRPs. There are some alternative iterative methods more efficient for large MRPs:  
         Dynamic programming   
         Monte-Carlo evaluation  
         Temporal-Difference learning

## Markov Decision Process

A sequential decision problem for a fully observable, stochastic environment with a Markov transition model and additive rewards.

* + 1. <S, A, P, R, gamma>

Actions can be any decisions we want to learn how to make

States can be anything we can know that might be useful in making them

* + 1. Policy
       1. The general definition of policy
       2. A policy is stationary if it doesn’t change over time. Non-stationary policies mainly introduced by the fact that deadline may results in rewards changing over time
    2. The Bellman equation
       1. The Bellman expectation equation for a given policy
          1. Relationship between the state-value function and the action-value function

From action-value function to state-value function

From state-value function to action-value function

* + - * 1. The state-value function of an MDP is the expected return starting from state s, and then following policy pi
        2. The action-value function is the expected return starting from state s, taking action a, and following policy pi
        3. Notice that with a fixed policy, MDPs are essentially the identical to MRPs — the transition probability is averaged over the action distribution of the policy
      1. The Bellman optimality equation
         1. The optimal state-value function
         2. The optimal action-value function

## Why do we discount the reward?

* + 1. Avoids infinite returns in cyclic Markov processes
    2. Uncertainty about the feature. The feature rewards may be unclear at present
    3. If the reward is financial, immediate rewards may earn more interest than delayed rewards
    4. Animal/human behavior shows preference for immediate reward
    5. It is sometimes possible to use un-discounted Markov reward process, e.g. if all sequence terminates

# Planning

## Concept

* + 1. Prediction

Which takes an MDP and a policy as inputs, and produces the value function as outputs

* + 1. Control

Which takes an MDP as inputs, and produce the optimal value function and optimal policy as outputs

* + 1. Generalized policy iteration

The general idea of letting policy evaluation and policy improvement processes interacts, independent of the granularity and other details of the two processes

* + - 1. Interaction between policy evaluation and policy improvement
         1. Competition:  
             Evaluation and improvement pull in opposite directions — making the policy greedy typically makes the value function incorrect for the new policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy  
             In the long run, evaluation and improvement interact to converge on the optimal solution

## Dynamic Programming Algorithm

* + 1. What’s dynamic programming?
       1. Dynamic programming is a method for solving complex problems by breaking them down into subproblems, then solving the subproblems and combining solutions to subproblems
       2. Problems solved by dynamic programming have two properties:  
          1. Optimal substructure: Principle of optimality applies; Optimal solution can be decomposed into subproblems  
          2. Overlapping subproblems: Subproblems recur many times; Solutions can be cached and reused

Principle of optimality says that a n optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision

* + 1. Why can we apply dynamic programming to MDPs?
       1. MDPs satisfies the two properties required by dynamic programming:  
          1. The Bellman equation gives recursive decomposition  
          2. Value function stores and reuses solutions
    2. Policy iteration

Which is based on the insight that the policy pi becomes optimal long before U{i} has converged

* + - 1. Algorithm
         1. def policy\_iteration(mdp):  
             # initialization  
             U = {s: 0 for s in mdp.states}  
             pi = {s: random.choice(mdp.actions(s)) for s in mdp.states}  
               
             while True:  
             U = policy\_evaluation(pi, U, mdp)  
             unchanged = True  
             # policy improvement  
             for s in mdp.states:  
             a = argmax(mdp.actions(s), key=lambda a: expected\_utility(a, s, U, mdp))  
             if a != pi[s]:  
             pi[s] = a  
             unchanged = False  
             if unchanged:  
             return pi  
              
            def expected\_utility(a, s, U, mdp):  
             """The expected utility of doing a in state s, according to the MDP and U."""  
             return sum([p \* U[s1] for (p, s1) in mdp.T(s, a)])

Algorithm Description

initialize utility function U and policy pi

do

policy evaluation: update U according to the current policy pi

policy improvement: improve the policy pi based on the updated U

until pi isn’t changed

* + - 1. Policy evaluation
         1. def policy\_evaluation(pi, U, mdp, k=20):  
             """Return an updated utility mapping U1 from each state in the MDP to its  
             utility, using an approximation (modified policy iteration)."""  
             # reward function, transition model, discount factor  
             R, T, gamma = mdp.R, mdp.T, mdp.gamma  
             U1 = U.copy()  
             for i in range(k):  
             for s in mdp.states:  
             U1[s] = R(s) + gamma \* sum([p \* U[s1] for (p, s1) in T(s, pi[s])])  
             return U1

Math behind the above algorithm:

Heads up: The above step need to repeat k times to get a good approximation. An exact policy evaluation return the utility when it doesn’t change or when it’s smaller than some tiny constant.

Strictly speaking, policy evaluation need to get to the equilibrium where  
 It’s time-consuming (O(n^3)) to do the exact policy evaluation. Instead, the approximation given in the above implementation works fine when dealing with large state space

In this case, it’s almost the same as value iteration except that policy evaluation has a policy set in stone, which eliminates max in value iteration

* + 1. Value iteration

An algorithm based on the Bellman optimality equation

* + - 1. Algorithm
         1. def value\_iteration(mdp, epsilon=0.001):  
             """  
             param: epsilon, the maximum error allowed in the utility of any state  
             """  
             U1 = {s: 0 for s in mdp.states}  
             # reward function, transition model, discount factor  
             reward, transition, gamma = mdp.R, mdp.T, mdp.gamma  
             while True:  
             U = U1.copy()  
             # delta: maximum change in the utility of any state in an iteration  
             delta = 0  
             for s in mdp.states:  
             U1[s] = reward(s) + gamma \* max([sum([p \* U[s1] for (p, s1) in transition(s, a)])  
             for a in mdp.actions(s)])  
             delta = max(delta, abs(U1[s] - U[s]))  
             # return the utility when the change in utility between consecutive iterative is very small  
             if delta < epsilon \* (1 - gamma) / gamma:  
             return U

Algorithm Description

U = initial utility function

while the utility function has a considerably large change between consecutive iteration:

recompute U based on the Bellman optimality equation

return U

* + - 1. Why do we restrain delta less than epsilon \* (1-gamma)/gamma?
         1. It is said that   
            If ||U{i+1}-U{i}|| < epsilon(1-gamma)/gamma  
            Then ||U{i+1}-U|| < epsilon

i.e. if the update between consecutive iterations is small, then the error between estimate and the true utility function is also small

* + - 1. In the above process, the policy pi becomes optimal long before U{i} has converged
      2. In fact, we can do this in place. That is, we don’t resort to a new utility table U1, we simply apply the Bellman optimality equation in U. In that case, the order of states to update really matters. There are some heuristics for ordering states
         1. Prioritized sweeping:  
            Which keeps states in a priority queue, where element’s value is measured by Bellman error  
             At every step we update state with the maximum Bellman error
         2. Real-time dynamic programming:  
            In which we only update states that are relevant to agent at the current timestep

RTDP is guaranteed to converge for stochastic optimal path problem, in which  
 The initial value of every goal state is zero  
 All the initial values are equal to, or greater than, their optimal values  
 The goal is guaranteed to be reached by a policy from any start state  
 All rewards are negative

* + 1. Relation & comparison between value iteration and policy iteration
       1. If we modify value iteration so that it stops when the greedy policy doesn’t change and set k in policy evaluation to 1 for policy iteration, then value iteration is somewhat identical to policy iteration
       2. Policy iteration employs the Bellman expectation equation (at the policy-evaluation step) and greedy policy improvement, whereas utility iteration employs the Bellman optimality equation
       3. Both policy iteration and value iteration are widely used , and it is not clear which, if either, is better in general

Both converge much faster than the theoretical worst-case run time, particularly if they are started with good initial value functions or policies

* + 1. Limits of dynamic programming
       1. Computational limit: Dynamic programming is effective for medium-sized problem (millions of states). For large problems, it suffers Bellman’s curse of dimensionality since it considers every successor state and action

Bellman’s curse of dimensionality: the number of states(with timestamp) grows exponentially with the number of state variables

* + - 1. Full knowledge about the dynamics of the environment: It requires the knowledge of the MDP transitions and reward function. Thus, it’s model-based.

## Evolutionary methods

Evolutionary methods apply multiple static policies each interacting over an extended period of time with a separate instance of the environment. The policies that obtain the most reward(unbiased since such an estimate is based on the experience), and random variations of them, are carried out to the next generation of the policies and the process repeats

* + 1. Genetic algorithm

A variant of stochastic beam search in which successor states are generated by combining two parent states. In genetic algorithm, each state is represented by a string and is rated by the fitness function. It generates the next k states by following three steps:

1. Randomly select two pairs from the current k states for reproduction, in accordance with the probabilities calculated by the fitness function. Notice that different pairs may contain a same state and some states may just be omitted

2. Create the offspring by crossing over the parent strings at some randomly picked crossover point.

3. Mutation happens to the offspring with a small independent probability

* + 1. Simulated annealing

A local search that randomly select a successor and adopt it with probability e^{∆E/T} when it’s bad, where ∆E is the difference between the value of the successor and that of current state (so it’s negative for bad successors), T is given by an annealing schedule function, which slowly decreases over time

* + - 1. Algorithm Overview
         1. def exp\_schedule(k=20, lam=0.005, limit=100):  
             """One possible schedule function for simulated annealing"""  
             return lambda t: (k \* math.exp(-lam \* t) if t < limit else 0)  
              
            def simulated\_annealing(problem, schedule=exp\_schedule()):  
             """[Figure 4.5] CAUTION: This differs from the pseudocode as it  
             returns a state instead of a Node."""  
             current = Node(problem.initial)  
             for t in range(sys.maxsize):  
             T = schedule(t)  
             if T == 0:  
             return current.state  
             neighbors = current.expand(problem)  
             if not neighbors:  
             return current.state  
             next = random.choice(neighbors)  
             delta\_e = problem.value(next.state) - problem.value(current.state)  
             if delta\_e > 0 or probability(math.exp(delta\_e / T)):  
             current = next

Algorithm description

For t in time bound

Calculate the temperature T based on t

If T is 0, which means the temperature reaches a low-energy state, return current state

Randomly pick a state in its neighbors, adopt it when it’s a better solution or with probability e^{∆E/T} when it’s bad

* + - 1. When T goes to ∞, simulated annealing is like a random walk around its neighbor. When T goes to 0, it’s like hill climbing
    1. Pros
       1. Effective when the policy space is small, or can be structured so that good policies are easy to find, or if a lot of time is available for search
       2. Have advantage on problems in which the learning agent cannot sense the complete state of its environment
    2. Cons
       1. Ignore much of the useful structure of the reinforcement learning problem:   
          1. They do not use the fact that the policy they are searching for is a function from states to actions  
          2. They do not notice which states an individual passes through its life time, or which actions it selects

Although in some cases this information can be misleading (e.g., when states are misperceived), but more often it should enable more efficient search

# Model-Free Prediction

## Monte-Carlo method

A model-free method, which learns directly from episodes of experience — such episodes must terminate

* + 1. Incremental update rule for on-policy MC  
        Where G{t} is the actual return, which is calculated backward for each episode. The error term, (G{t}-V(s)), indicates how the return adjusts the value function for the current state
       1. Incremental update rule for off-policy MC (ordinary importance sampling)  
           Where W\_t is the weight. Typically, for importance sampling, it’s the ratio of the trajectory under the target policy to that under the behavior policy

The way W gets updated may cause early stopping. When some pi(A(k)|S(k)) is 0, all previous transitions (before timestamp k) are dropped since they won’t actually appear under the target policy.

* + 1. Incremental update rule for off-policy MC (weighted importance sampling)  
        Where C(s, a) is the sum of weights across different episodes, which is updated as below

Notice that here use action value instead of state value since C is defined based on both s and a.

Also since action value defines

* + 1. Ordinary importance sampling vs weighted importance sampling
       1. Ordinary importance-sampling estimator is unbiased but it has potential infinite variance
       2. Weighted importance-sampling estimator is biased but it usually comes with low variance (converging to zero assuming bounded returns). In practice, it’s generally preferred

## Temporal-difference learning

A model-free method, which learns directly from episodes of experience — such episodes could be incomplete

* + 1. TD resorts to the idea called bootstrapping so as to learn from incomplete episodes. That is, it resorts to the estimate of the next state value to update the value function instead of computing the real return backward as it’s done by Monte-Carlo methods.

Dynamic programming also uses bootstrapping

* + 1. Incremental update rule  
        Where R(s)+gammaV(s’) is called the TD target. The TD error  
        indicates how the TD target should adjust the value function for the current state
    2. Why do we derive the TD target from the successive state, why not from previous state?
       1. The successor is closer to the terminal state, and hence always bears more certainty than the current state, which ensures TD learning converges

## TD(lambda)

* + 1. On-policy
       1. We define the n-step return  
           Then we have n-step TD learning

TD learning is exactly 1-step return, while MC method is infinite-step return

* + - 1. The lambda-return combines all n-step returns  
          The forward view of TD(lambda)

TD(0) is identical to TD learning

TD(1) is identical to MC method

* + - 1. Define an eligibility trace for every state s  
          Then we have the backward view of TD(lambda)  
          Where delta is the TD target

Eligibility traces are initialized to be 0

Lambda determines the rate at which eligibility trace falls

* + - * 1. TD(lambda) is slightly different from other general policy evaluation strategies in that it updates all eligibility traces and states at each step

Note that the delta used to update each action-value function stays the same in the above process. This ensures the current TD error back propagate to all previous states

* + - * 1. Eligibility traces employs both frequency heuristic and recency heuristic to deal with credit assignment problem — it tends to assign more credit to the most frequent and most recent states

Frequency heuristic: assign credit to the most frequent states

Recency heuristic: assign credit to the most recent states

* + - 1. The forward view provides theory, whereas backward view provides mechanism which allows TD(lambda) update online from incomplete sequences
    1. Off-policy
       1. Importance sampling
          1. Importance sampling ratio

State value

Action value

For off-policy one-step TD algorithms that use action value instead of state value, such as Q-learning, the importance ratio is 1 since t+1 > t + 1 - 1

Importance sampling ratio starts from t in the case of state value, whereas, for action value, that starts from t + 1. Such difference introduced by the fact that action value has specified action at t

## MC vs TD

Long story short, TD is preferred

* + 1. In practice, TD usually converge faster than MC
    2. TD can learn before knowing the outcome. That makes TD an online algorithm.  
       MC, on the other hand, must wait until end of episode so that it can calculate returns for each state. That makes MC an offline algorithm
    3. TD can learn without the final outcome. That suggests TD can work in continuing(non-terminating) environments  
       MC, on the other hand, can only learn from complete sequences. That suggests MC only works for episodic (terminating) environments
    4. Bias/variance trade-off
       1. MC has high variance, zero bias. Therefore MC  
          1. has good convergence properties (even with function approximation)  
          2. is not very sensitive to initial value

High variance is caused by the return G{t} depending on many random transitions (from t onwards)

It has zero bias since the return is calculated from real trajectories.

* + - 1. TD has low variance, some bias. Therefore TD  
         1. is more efficient than MC (thanks to low variance)  
         2. is more sensitive to initial value (thanks to bias)  
         3. TD(0) converges to the optimal state-value function V\* (but not always with function approximation)

Low variance is for the TD target depends only on one transition

Some bias is caused by the bias of the TD target since the TD target is just an estimate

* + 1. MC best fits to the observed return, whereas TD exploits Markov property so as to solve the MDP that best fits the data.   
       As a result, TD is usually more efficient in Markov environment, and MC, on the other hand, usually more effective in non-Markov environment

## Unified view of reinforcement learning

* + 1. The following figure shows the difference among DP, TD and MC

DP bootstraps but does not sample

TD bootstraps and samples

MC doesn’t bootstrap but samples

## Expected vs. sample updates

* + 1. The expected update for a state-action pair is  
        The corresponding sample update is

The expected and sample updates are identical when

there is only one possible next state (i.e., p(s’, r|s, a) is 1 for some s’, r, and 0 for the rest)

alpha in sample update is 1

* + 1. The expected update is an exact computation, resulting a new Q whose correctness is limited by the correctness of the Q at successor states. The sample update is in addition affected by sampling error.
    2. Sample updates are superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly since expected updates have high computational cost on those problems. In addition, thanks to the frequent updates, values backed up from the successor will be more accurate in sample updates

## Does TD lead to the same result as MC?

* + 1. No. The reason is well explained [here](#Row_171_MC_best_fits_to_the_observed_r). For a concrete example, considering we have below data:  
       A-0, B-0  
       B-1  
       B-1  
       B-0  
       Where A and B are states, the number following are rewards. We can easily compute V(B)=0.5. As for A, things becomes slightly different. If we use TD, we’ll end up with V(A)=0.5; with MC, we’ll have V(A)=0

# Model-Free Control

## Algorithm

* + 1. General pattern for control algorithm
       1. While true  
           sample episodes/transitions using policy  
           do policy evaluation to update value function  
           do policy improvement to update policy

There are two common policy patterns

1. Policy is responsible for exploration

2. Policy is always greedy based on the current value function, and we take into account exploration when we’re taking actions

* + 1. Monte-Carlo control

An on-policy, offline algorithm using MC method to do policy evaluation

* + 1. SARSA

An on-policy, possibly online algorithm using TD to do policy evaluation

* + 1. Q-learning

An off-policy, possibly online algorithm using TD to do policy evaluation

* + - 1. The update rule  
          Where a\* is an action sampled from another policy. A general choice is  
          And the corresponding update rule is
    1. Expected SARSA
       1. The update rule for one-step Expected SARSA  
           Where pi could be either the behavior policy (in which case, it’s on-policy) or another target policy (in which case, it’s off-policy).
       2. The update rule for n-step Expected SARSA  
           Where the n-step return G is defined by
       3. When the target policy is the greedy policy, Expected SARSA is exactly Q-learning.
       4. Expected SARSA may completely dominate both SARSA and Q-learning

## Q & A

* + 1. Why do we use action-value function instead of state-value function in model-free control?
       1. When we use state-value function, the policy is updated by   
           in which the transition probabilities are involved. That causes the algorithm model-based.  
          On the other hand, with help of action-value function, the policy is simply updated by   
           in which no model information is required. Thus, using action-value function makes the algorithm model-free.
    2. What’s the difference between on-policy and off-policy algorithms?
       1. On-policy algorithms use the learned policy to generate behavior, whereas off-policy algorithms have two different policies: the target policy and the behavior policy

The target policy: the policy being learned about

The behavior policy: the policy used to explore and generate behavior in off-policy algorithms

* + 1. Why do off-policy algorithms work?
       1. Importance sampling transforms the expectations from the behavior policy to the target policy and thereby ensures the convergence of the target policy
    2. Why is off-policy important?
       1. Learn from observing humans or other agents
       2. Reuse experience generated from old policies
       3. Learn about optimal policy while following exploratory policy
       4. Learn about multiple policy while following one policy
    3. What’s the advantage of model-free control algorithms over dynamic programming?
       1. Model-free control learns optimal behavior directly from interaction with the environment, with no model of the environment’s dynamics
       2. Model-free control can be used with simulation or sample models
       3. It is easy and efficient to focus model-free control on a small set of the states
    4. Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as SARSA?
       1. No. Because SARSA selects the next action before the update of the action-value function, whereas Q-learning does after the update.

## Relationship between DP and TD

* + 1. TD is a sampling version of DP

# Function Approximation

## Feature construction

* + 1. Polynomials
       1. For a feature vector  
           each order-n polynomial-basis feature is

## Variance

* + 1. Variance leads to step size varying significantly from time to time. Large step size results in unreliable moves, while small step size slow the learning. Sophisticated optimizers such as momentum stochastic, etc. help ease variance issues

## Stochastic gradient descent

* + 1. Define loss function  
        And the difference operator for theta  
        Then we have

Q^pi is the target. In practice, it’s decided by the specific algorithm

For MC, it’s the return

For TD(0), it’s the TD target

For TD(lambda), it’s the lambda-return

* + - 1. delta varies from algorithm to algorithm.  
         For SARSA, it’s  
          For expected SARSA, it’s  
          For n-step SARSA, it’s
      2. For off-policy, we multiply the difference by importance sampling ratio, and thus have  
          Where the importance sampling ratio is

This is weighted importance sampling

* + 1. Note that when we use the TD target or lambda-return as the target, we introduce some bias. Although a linear function approximator may still converge under this circumstance, a non-linear function approximator such as neural network diverges
    2. For TD(lambda), the eligibility trace becomes vector defined by  
        And the difference operator is  
       1. TD(lambda)
       2. SARSA(lambda) with binary feature and linear function approximation

This provides an example dealing with the terminal case for action-value function approximator

## Average reward

An undercounted optimality framework said to be more appropriate than discounted framework

* + 1. We define the average reward when following a policy as  
        Where mu is the state distribution under policy pi.
    2. In the average-reward setting, returns are defined in term of the difference between rewards and the average reward, namely differential returns  
        thereby, the differential TD error is defined by
    3. Differential SARSA

delta is computed before average reward

* + 1. Differential n-step SARSA

## Condition

* + 1. A training method must be suitable for non-stationary, non-iid data

RL data is non-stationary since the policy/environment changes over time

## Type of value function approximation

* + 1. state -> state-value approximation
    2. state, action -> action-value approximation
    3. state -> action-value approximations for all action

Most common choice for its efficiency

## Convergence

* + 1. Convergence of Prediction Algorithm

MC, TD are off-policy when the experiences fed to them are generated by another policy

In the on-policy linear case, MC and TD converge to different points

* + - 1. In function approximation case, MC methods converge to the representable value function that is closest to the true value function — the projection of the true value function on the function-approximation subspace. In other words, MC methods find weights that minimize the difference between the true value function and value function approximator
      2. TD methods converge to the point which minimize the Mean Squared Bellman Error  
          As a convention, we define the Bellman operator as  
          The Bellman error delta used in the Mean Squared Bellman Error is  
          or  
          The Bellman error is 0 when

Table lookup 1-step bootstrapping methods, such as TD or DP, converge to the point where the Bellman error is 0

Bootstrapping function-approximation methods, however, converge to the point where the projection of Bellman error on the function-approximation subspace is 0

* + 1. Convergence of Control Algorithm

(√) = oscillate around near-optimal value function

The reason why MC control and SARSA oscillate with linear function approximator and does not work with non-linear function approximator is that policy gets updated in the control algorithm

* + - 1. The reason why control algorithms may fail to converge in function approximation cases is that, for function approximation methods, there is no guarantee that a single improvement on one state improves the overall policy since any update for weights has an influence on value estimates for all state(-action)

## Linear least squares prediction

* + 1. It is possible, in prediction, to directly calculate the weights for linear function approximator based on the observation that the expected update must be zero at the minimum of the loss function:

## Importance sampling

* + 1. In function approximation, if we apply the importance sampling ratio to the TD target in the mean square root loss, we end up having the ordinary importance sampling. If we apply it to the TD error, we end up with the weighted importance sampling

Reference: “Weighted importance sampling for off-policy learning with linear function approximation”

## Q & A

* + 1. Why do we bother to use function approximation to represent value function?
       1. Traditional table-based approach is inefficient and even ineffective to handle the case where the state space is huge

Inefficiency: too slow to learn the value of each state individually

Ineffectiveness: too many information to store in memory, unable to generalize to unvisited states

# Policy Gradient Methods

## Policy representation

Which introduces continuity into policy and thereby avoid an infinitesimal change in theta causing the policy to switch from one action to another

* + 1. softmax policy   
       1. The score function for the softmax policy  
           Which says how much these feature (nabla part) than usual
    2. Gaussian policy

Where sigma may be pre-fixed or parametrized

* + - 1. The score function for the Gaussian policy  
          It multiplies how much the action than usual by the features (nabla part), then scale it by variance

## Q & A

* + 1. What’s the advantage/disadvantage of policy-based RL against value-based RL?
       1. Advantages
          1. Better convergence properties: value-based methods can oscillate even diverge in some cases, where policy-based methods are guaranteed to converge at least to a local optimum if we just directly follow the policy gradient
          2. Effective in high-dimensional or continuous action spaces: value-based methods require a maximization operation to select an action at each step. This operation could be intimidating when action space is large or continuous. Policy-based methods circumvent this operation by adjusting the parameters of the policy directly, and thereby incrementally learn what’s the best action to take at each state
          3. Can learn stochastic policy: deterministic policy characterized by maximizing value function just doesn’t work in cases, such as rock-paper-scissors and some partially observable environment where two different states may seem the same to the agent. It is necessary to act stochastically in those cases to break the tie
       2. Disadvantages
          1. Naive policy-based methods converge very slowly, and have high variance.
          2. Typically converge to a local rather than global optimum
    2. Why do we go through all the troubles to obtain expectations?
       1. Expectation is the reason why we can apply sampling to approximate the result

# Integrating Learning and Planning

## Model

* + 1. Representation
       1. State transition
       2. Reward
       3. Typically we assume conditional independence between state transitions and rewards
    2. Two types of model
       1. Distribution model

A distribution model consists of the probabilities of next states and rewards of possible actions

* + - * 1. Dynamic programming requires a distribution model to do expected updates
      1. Sample model

A sample model produces single transitions and rewards generated according to the probabilities of next states and rewards of possible actions

* + - * 1. A sample model simulates interacting with the environment to do sample updates. Therefore, it generally works with reinforcement learning algorithms
    1. Models
       1. Table lookup model
          1. Calculate from experience  
             state transitions  
              rewards
          2. Store each experience tuple (s, a, r, s’), then latter when we want to sample model, we randomly pick a tuple match (s, a, ., .)
       2. Linear expectation model
       3. Linear Gaussian model
       4. Gaussian process model
       5. Deep belief network model

## Planning

Improve the value/policy based on the model the agent learns

* + 1. Value iteration
    2. Policy iteration
    3. Tree search
    4. [Sample-based planning](omnioutliner:///open?focus=lSyxD0l5f_x&row=di7dh1vsA1A)

Which sample experiences from model and then feed them to a model-free control algorithm

* + - 1. Sample-based planning methods are often more efficient

## Algorithm

* + 1. General pattern for model-based algorithm
       1. Learn model  
          Do planning
    2. Background planning

In which planning simulates experience for value updates

* + - 1. Sample-based planning
         1. Sample experience from model  
            Apply to samples model-free RL (MC control, SARSA, Q-learning, etc.)
      2. Dyna-Q

Which uses the current model to generate more experience for Q network

* + - * 1. Initialize Q function and Model  
           While True  
            Take an action based on Q, observe reward r, and state s’  
            Update Q and Model using the transition (s, a, r, s’)  
            Repeat n times // planning updates  
            sample from Model transitions whose s and a are previously observed/taken  
            update Q based on the simulated transitions

The planning stage speeds up the learning process

* + - * 1. Dyna-Q struggles when the model is incorrect caused by stochastic environment, defective function approximator for imperfect generalization, change of environment, etc. Especially, when the environment changes to be better, Dyna-Q may not even be able to detect such modeling error
        2. Dyna-Q shares similarity with simulation-based search in that both feed simulated experience to model-free control algorithm.
      1. Dyna-Q+

Which encourages actions untried for a long time by adding extra reward to those transitions when updating Q

* + - * 1. If the model reward for a transition is R, and the transition has not been tried in tau times steps. Then planning updates are done as if that transition produced a reward of R+k(√tau), for some small k. More specifically, the update rule at the planning stage becomes
        2. Another modification is that Dyna-Q+ algorithm allows to select actions which have never been tried before. Such actions are assumed to initially lead back to the same state with a reward of zero, i.e., Model(s, a)=(0, s), if a has never been tried before.
      1. Backward focusing
         1. Backward focusing modifies Dyna-Q so that the states preceding the updated state are sampled to be update
      2. Prioritized sweeping

Prioritized sweeping takes advantage of the fact that if a state-action value is changed dramatically, then it is likely that state-actions leading to this state will have big change

* + - * 1. Initialize Q function and Model  
           Initialize a priority queue PQueue   
           While True  
            Take an action based on Q, observe reward r, and state s’  
            Update Model using the transition (s, a, r, s’)  
            Compute absolute value of Q error P for s, a  
            If P is greater than some threshold, insert s, a into PQueue with priority P  
            Repeat n times, PQueue is not empty // planning updates s, a = PQueue.pop  
            Sample transition (s, a, r, s’) from Model  
            Update Q based on the simulated transitions  
            For each s-, a- leading to s  
            Compute absolute value of Q error P for s-, a-  
            if P is greater than some threshold, insert s, a into PQueue with priority P
        2. One of prioritized sweeping’s limitations is that it uses expected updates, which in stochastic environments may waste lots of computation on low-probability transitions. Sample updates in many case get closer to the true value function with less computation despite the variance introduced by sampling.

Although prioritized sweeping updates along transitions, like sampling updates, but it does so based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

* + - 1. Trajectory sampling
         1. Trajectory sampling simulates trajectories according to the model learned when following the current policy.
         2. Trajectory sampling puts more considerations on those states frequently visited when following the current policy and ignores vast uninteresting states.

This makes the learning process faster but may impair the ultimate performance in the long run.

* + - * 1. Trajectory sampling gains a great advantage for large problems, in particular for problems in which only a small subset of the state-action space is visited under the policy
      1. Real-Time Dynamic Programming (RTDP)
         1. At each step, RTDP applies the expected value-iteration update operation to the current state, and select a greedy action to move to the next state.   
            It can also update the values of an arbitrary collection of other states at each step (e.g., it can update the values of states visited in limited-horizon look-ahead search from the current state.)

RTDP can cooperate with [Learning Real-Time A\*](omnioutliner:///open?focus=gwVusVQ8jnd&row=mJmGoOCyunE)

* + - * 1. For problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided:  
           1. The initial value of every goal state is zero  
           2. There exists at least one policy that guarantees that a goal state will be reached with probability one from any start state  
           3. All rewards for transitions from non-goal states are strictly negative  
           4. All the initial values are equal to, or greater than, their optimal values (which can be satisﬁed by simply setting the initial values of all states to zero).

Tasks having these properties are examples of stochastic optimal path problems, which are usually stated in terms of cost minimization instead of reward maximization.

* + 1. Decision-time planning

In which planning focuses on a particular state and selects a single action

* + - 1. Rollout algorithm

Rollout algorithms estimate action values by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy

* + - * 1. The better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely to be. This is because a rollout takes the best action on the current state evaluated by the rollout policy, i.e., it differs from the rollout policy only at the current state. Since the action selected at the current state is the best, the resultant policy should be as good as, or better than the rollout policy.
        2. The computation time needed by a rollout algorithm depends on:  
           1. The number of actions that have to be evaluated for each decision  
           2. The number of time steps in the simulated trajectories needed to obtain useful sample returns  
           3. The time it takes the rollout policy to make a decision (i.e. the time an action is taken to act)  
           4. The number of simulated trajectories needed to obtain good Monte Carlo action-value estimates
      1. Simulation-based search

Which simulates tree search by sampling. That is, instead of recursively calculating the expected value for each state, we generate episodes of experience from now onward with the model, and then feed them to model-free RL to compute value for each state

* + - * 1. Simulate episodes of experience from now onward with Model  
           Apply model-free RL to simulated episodes to make an action

It uses sampling to approximate the expected value.

* + - 1. Monte Carlo Tree Search (MCTS)
         1. There are two set of nodes: Visited nodes on which simulations have been performed Unvisited nodes on which no simulation is performed  
            Based on the above definition, we further partition visited nodes into two sets Fully expanded nodes whose children are all visited  
             Not-fully expanded nodes(aka. leaf nodes) some of whose potential children are not visited (i.e., not yet added to the tree)
         2. There are two different policies: Tree policy, which picks actions within the part of the tree comprised of visited nodes states Rollout policy, which simulations/rollouts are performed based on

The tree policy could be any policy that takes into account both exploration and exploitation, a typical choice is Upper Confidence Bound applied to trees (UPC)

The rollout policy could be a uniform random policy or a shallow network

* + - * 1. While time remains Selection: perform the tree policy from root traversing down to a leaf node Expansion: add some unvisited node reached from the selected leaf node via unexplored actions Simulation: run a simulation(according to the rollout policy) from the newly-added child Backpropagation: back propagate the simulation results to the root  
           After the environment transitions to a new state, MCTS run again, starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS, all remaining nodes are discarded, along with the action values associated with them.

Alpha Zero doesn’t perform any simulation at all, instead, it evaluates the unvisited node with a CNN network

Monte-Carlo control is applied during simulation. Moreover, we could substitute the Monte-Carlo algorithm with some TD algorithm to form TD search

* + - * 1. Advantage of MCTS  
            It is a highly selective best-first search since, in the tree policy, we act greedily (with a certain rate of exploration)  
            It evaluates states dynamically because we only focus on the current state.  
            It avoids the curse of dimensionality using sampling  
            It is computationally efficient, anytime, parallelizable

An anytime algorithm is an algorithms that could return a solution even if it is interrupted before it ends. The more time it keeps running, the better solution it's expected to obtain

* + - 1. TD search
         1. TD search is almost the same as Monte Carlo tree search, except that at simulation state, it doesn’t run a complete episode. Instead, it updates the unvisited node simply using a TD evaluation
    1. Dyna-2
       1. Dyna-2 stores two set of feature weights  
           long-term memory  
           short-term memory

Long-term memory is updated from real experience using TD learning

Short-term memory is updated from simulated experience using TD search

* + - 1. The value function is sum of long and short-term memories
    1. Background planning uses simulated experience to gradually improve policy or value function, while decision-time planning uses simulated experience to select an action for the current state.   
       These two can be blended together if needed.

## Advantage/disadvantage of model-based RL

* + 1. Advantage:  
        When the state space is large and moving from one state to another state completely change the value function/policy (for example chess), it is hard to learn a value function or policy directly. On the other hand, the model (the rule of chess) may be quite straightforward and compact, thus learning a model then then planning on it may be a better choice.  
        Model can be learnt efficiently by supervised learning methods (using real feed back (s, a, r, s’) to supervise the learning of the model)  
        Model can tell us what we know and don’t know of the world so that we can manually choose to explore the world
    2. Disadvantage:  
        We need to learn a model and then construct a value function. So there are two places to which error may be introduced

## Q & A

* + 1. What kind of role does experience play in a planning agent?
       1. Improve the model
       2. Improve the value function or policy
       3. Make decision (decision time planning)