Reinforcement learning: The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence. Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning. Basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Reinforcement learning is also different from what machine learning researchers call unsupervised learning, which is typically about finding structure hidden in collections of unlabeled data. Reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-o↵ between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. Characteristics: 1) There is no supervisor, only a reward signal. 2) Feedback is delayed, not instantaneous. 3) Time really matters (sequential, non i.i.d data). 4) Agent's actions affect the subsequent data it receives.
Elements of reinforcement learning: 1) Reward: A reward Rt is a scalar feedback signal. Indicates how well agent is doing at step t. The agent's job is to maximise cumulative reward. Reinforcement learning is based on the reward hypothesis. Reward hypothesis: All goals can be described by the maximisation of expected cumulative reward. The Reward Function is an incentive mechanism that tells the agent what is correct and what is wrong using reward and punishment. The goal of agents in RL is to maximize the total rewards. Sometimes we need to sacrifice immediate rewards in order to maximize the total rewards. 2) Sequential decision making: Goal: select actions to maximise total future reward. Actions may have long term consequences. Reward may be delayed. It may be better to sacrifice immediate reward to gain more long-term reward. a) Action: Choose a set of action according to the problem characteristics. b) Reward: Actions at any time step t have reward Rt. c) Policy: Choose action according to reward and formalize this called as the policy. 3) History: The history is the sequence of observations, actions, rewards Ht = O1; R1; A1; :::; At−1; Ot; Rt. All observable variables up to time t. What happens next depends on the history: The agent selects actions. The environment selects observations/rewards. 4) State: State is the information used to determine what happens next Formally, state is a function of the history: St = f (Ht). Types of states: a) Environment state: The environment state Set is the environment's private representation i.e. whatever data the environment uses to pick the next observation/reward. The environment state is not usually visible to the agent. Even if Set is visible, it may contain irrelevant information. b) Agent state: The agent state Sat is the agent's internal representation i.e. whatever information the agent uses to pick the next action i.e. it is the information used by reinforcement learning algorithms. It can be any function of history: Sat = f (Ht). c) Information state: An information state (a.k.a. Markov state) contains all useful information from the history. Once the state is known, the history may be thrown away i.e. The state is a sufficient statistic of the future. The environment state Set is Markov. The history Ht is Markov.

Major components of an RL agent: 1) Policy: Agent’s behavior function. It is a map from state to action. A policy defines the learning agent’s way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus–response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action. 2) Value function: Value function is a prediction of future reward. Used to evaluate the goodness/badness of states. Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. 3) Model: A model predicts what the environment will do next. The fourth and final element of some reinforcement learning systems is a model of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners—viewed as almost the opposite of planning. 4) Reward signal: A reward signal defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the reward. The agent’s sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent. In a biological system, we might think of rewards as analogous to the experiences of pleasure or pain. They are the immediate and defining features of the problem faced by the agent. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

Categorizing RL agents: 1) Value based: No policy. The goal of a value-based Reinforcement Learning approach is to optimize a value function V. (s). In this strategy, the agent anticipates a long-term return of the current policy states. 2) Policy based: No value function. In this reinforcement learning model, you strive to come up with a policy that allows you to get the most reward in the future by doing actions in each state. 3) Actor critic: Policy and value function both are present. 4) Model Free: Policy and / or value function and no model. On the other hand, model-free algorithms seek to learn the consequences of their actions through experience via algorithms such as Policy Gradient, Q Learning, etc. In other words, such an algorithm will carry out an action multiple times and will adjust the policy (the strategy behind its actions) for optimal rewards, based on the outcomes. 5) Model based: Policy and / or value function and model. Model-based, as it sounds, has an agent trying to understand its environment and creating a model for it based on its interactions with this environment. In such a system, preferences take priority over the consequences of the actions i.e. the greedy agent will always try to perform an action that will get the maximum reward irrespective of what that action may cause .Model based: You must develop a virtual model for each environment in this Reinforcement Learning approach. The agent learns how to perform in that particular setting.

Learning and planning: 1) Learning: The environment is initially unknown. The agent interacts with the environment. The agent improves its policy. 2) Planning: A model of the environment is known. The agent performs computations with its model (without any external interaction). The agent improves its policy.
Exploration and exploitation: Reinforcement learning is like trial-and-error learning. The agent should discover a good policy. From its experiences of the environment. Without losing too much reward along the way. Exploration finds more information about the environment. Exploitation exploits known information to maximise reward. It is usually important to explore as well as exploit. Deciding the best course after going through all of the courses outlines one by one is might be the ideal solution for him. In reinforcement learning, this is an exploration where one gathers information for assessing the scenario that may lead him to a better decision in the future. After the exploration, he may decide to learn from a specific course. This is called exploitation in reinforcement learning where one can take the optimal decisions with the highest possible outcome given current acquired knowledge or information. Exploration is a necessary step though it is a labor intensive and time-consuming process. In consequence, it raises questions: how long we should explore? when we should start exploiting? how much we should exploit? Investigating these questions provides us with what we actually seek: "identifying the best option to exploit at the same time exploring a sufficient number of options". This is the explore-exploit dilemma in reinforcement learning.
Prediction and control: 1) Prediction: Evaluate the future. Given a policy. 2) Control: optimise the future. Find the best policy. The difference between prediction and control is to do with goals regarding the policy. The policy describes the way of acting depending on current state, and in the literature is often noted as $\pi(a|s)$, the probability of taking action a when in states. A prediction task in RL is where the policy is supplied, and the goal is to measure how well it performs. That is, to predict the expected total reward from any given state assuming the function $\pi(a|s)$ is fixed. A control task in RL is where the policy is not fixed, and the goal is to find the optimal policy. That is, to find the policy $\pi(a|s)$ that maximises the expected total reward from any given state. A control algorithm based on value functions (of which Monte Carlo Control is one example) usually works by also solving the prediction problem, i.e. it predicts the values of acting in different ways, and adjusts the policy to choose the best actions at each step. As a result, the output of the value-based algorithms is usually an approximately optimal policy and the expected future rewards for following that policy.
Markov decision process: Markov decision processes formally describe an environment for reinforcement learning. Where the environment is fully observable. Every RL problem can be converted to MDP's: 1) Optimal control primarily deals with continuous MDPs. 2) Partially observable problems can be converted into MDPs. 3) Bandits are MDPs with one state. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. In a Markov decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for St and Rt depends only on the immediately preceding state and action. The MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment.
Goals and rewards in MDP: In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, the reward is a simple number, Rt 2 R. Informally, the agent's goal is to maximize the total

amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run.
Returns and episodes: In general, we seek to maximize the expected return, where the return, denoted Gt, is defined as some specific function of the reward sequence. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Tasks with episodes of this kind are called episodic tasks. In episodic tasks we sometimes need to distinguish the set of all nonterminal states, denoted S, from the set of all states plus the terminal state, denoted S+. The time of termination, T, is a random variable that normally varies from episode to episode.
Policies and value functions: Almost all reinforcement learning algorithms involve estimating value functions—functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Formally, a policy is a mapping from states to probabilities of selecting each possible action. The value functions vpie and qpie can be estimated from experience.
Optimal policies and optimal value functions: There is always at least one policy that is better than or equal to all other policies. This is an optimal policy. Although there may be more than one, we denote all the optimal policies by vpiestart.
Extension to MDP's: 1) Infinite and continuous MDP's: Countably infinite state and/or action spaces. Continuous state and/or action spaces. Closed form for linear quadratic model (LQR). Continuous time. Requires partial differential equations. Hamilton-Jacobi-Bellman (HJB) equation. Limiting case of Bellman equation as time-step -> 0.

Dynamic programming: Dynamic sequential or temporal component to the problem. Programming optimising a "program", i.e. a policy. A method for solving complex problems. By breaking them down into subproblems. Solve the subproblems. Combine solutions to subproblems. Dynamic Programming is a very general solution method for problems which 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. 3) Markov decision processes satisfy both properties: Bellman equation gives recursive decomposition. Value function stores and reuses solutions. It assumes full knowledge of the MDP. Applications: Scheduling algorithms, string algorithms, graph algorithms, graphical models, bioinformatics. The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases.
Iterative policy evaluation: Problem: evaluate a given policy π. Solution: iterative application of Bellman expectation backup. Using synchronous backups, At each iteration k + 1, For all states s E S, Update vk+1(s) from vk(s'), where s' is a successor state of s. Once a policy, pie, has been improved using vpie to yield a better policy, piestar, we can then compute vpiestart and improve it again to yield an even better . We can thus obtain a sequence of monotonically improving policies and value functions. Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.
Policy improvement: Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function vpie for an arbitrary deterministic policy pie.
Policy evaluation: First we consider how to compute the state-value function vpie for an arbitrary policy pie. This is called policy evaluation in the DP literature. We also refer to it as the prediction problem.

Value iteration: One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to vpie occurs only in the limit. In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called value iteration. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps. Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep.
Asynchronous dynamic programming: A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive Asynchronous DP backs up states individually, in any order. For each selected state, apply the appropriate backup. Can significantly reduce computation. Guaranteed to converge if all states continue to be selected. Three simple ideas for asynchronous dynamic programming: In-place dynamic programming, Prioritised sweeping and Real-time dynamic programming.
Value function space: Consider the vector space V over value functions. There are |S| dimensions. Each point in this space fully specifies a value function v(s). What does a Bellman backup do to points in this space? We will show that it brings value functions closer and therefore the backups must converge on a unique solution. We will measure distance between state-value functions u and v by the infinity-norm.
Convergence of Iterative Policy Evaluation and Policy Iteration: The Bellman expectation operator Tπ has a unique fixed point. vπ is a fixed point of Tπ (by Bellman expectation equation). By contraction mapping theorem Iterative policy evaluation converges on vπ Policy iteration converges on v∗.
Convergence of value function: The Bellman optimality operator T∗ has a unique fixed point. v∗ is a fixed point of T∗ (by Bellman optimality equation). By contraction mapping theorem. Value iteration converges on v∗.
Reinforce algorithm: REINFORCE belongs to a special class of Reinforcement Learning algorithms called Policy Gradient algorithms. A simple implementation of this algorithm would involve creating a Policy: a model that takes a state as input and generates the probability of taking an action as output. A policy is essentially a guide or cheat-sheet for the agent telling it what action to take at each state. The policy is then iterated on and tweaked slightly at each step until we get a policy that solves the environment. The policy is usually a Neural Network that takes the state as input and generates a probability distribution across action space as output. Each policy generates the probability of taking an action in each station of the environment. The agent samples from these probabilities and selects an action to perform in the environment. At the end of an episode, we know the total rewards the agent can get if it follows that policy. We backpropagate the reward through the path the agent took to estimate the “Expected reward” at each state for a given policy. Algorithm steps: 1) Inititialize a random policy. 2) Use the policy to iterate definite number of steps. 3) Calculate the discounted reward for each step by backpropagation. 4) Calculate expected reward G. 5) Adjust weights of Policy (back-propagate error in NN) to increase G. 6) Repeat from 2.
Monte carlo reinforcement learning: MC methods learn directly from episodes of experience. MC is model-free: no knowledge of MDP transitions / rewards. MC learns from complete episodes: no bootstrapping. MC uses the simplest possible idea: value = mean return. It can only be applied to episodic MDP’s. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks.

That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed.
Monte carlo prediction: We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return—expected cumulative future discounted reward—starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods. In particular, suppose we wish to estimate vpie(s), the value of a state s under policy pie, given a set of episodes obtained by following pie and passing through s. Each occurrence of state s in an episode is called a visit to s. Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the first visit to s. The first-visit MC method estimates vpie(s) as the average of the returns following first visits to s, whereas the every-visit MC method averages the returns following all visits to s. These two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties. Both first-visit MC and every-visit MC converge to vpie(s) as the number of visits (or first visits) to s goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of vpie(s) with finite variance.
Monte carlo control: Monte Carlo estimation can be used in control, that is, to approximate optimal policies. In GPI(generalised policy iteration) one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy pie0 and ending with the optimal policy and optimal action-value function: pie0 ->e qpie0 ->I pie1. E = Evaluation and I = Improvement. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each qpiek exactly, for arbitrary piek. Policy improvement is done by making the policy greedy with respect to the current value function. The theorem assures us that each piek+1 is uniformly better than piek, or just as good as piek, in which case they are both optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function. In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.
Temporal difference learning: TD methods learn directly from episodes of experience. TD is model free: no knowledge of MDP transitions / rewards. TD learns from incomplete episodes, by bootstrapping. TD updates a guess towards a guess. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome.
TD prediction: Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy pie, both methods update their estimate V of vpie for the nonterminal states St occurring in that experience. A simple every-visit Monte Carlo method suitable for nonstationary environments is $V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big],$ Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to V (St) (only then is Gt known), TD methods need to wait only until the next time step. The simplest TD method makes the update
$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$.
Advantages and disadvantages: 1) TD can learn before knowing the final outcome. TD can learn online after every step. MC must wait until end of episode before return is known. 2) TD can learn without the final outcome. TD can learn from incomplete sequences. MC can only learn from

complete sequences. TD works in continuing (non-terminating) environments. MC only works for episodic (terminating) environments. 3) MC has high variance, zero bias. Good convergence properties (even with function approximation). Not very sensitive to initial value. Very simple to understand and use. 4) TD has low variance, some bias. Usually more efficient than MC. TD(0) converges to vπ(s) (but not always with function approximation). More sensitive to initial value. 5) MC converges to solution with minimum mean-squared error. TD(0) converges to solution of max likelihood Markov model. 6) TD exploits Markov property. Usually more efficient in Markov environments. MC does not exploit Markov property. Usually more effective in non-Markov environments. 7) MC does not bootstrap. DP bootstraps. TD bootstraps. MC samples. DP does not sample. TD samples.
TD and MC: TD(1) is roughly equivalent to every-visit Monte-Carlo. Error is accumulated online, step by-step. If value function is only updated offline at end of episode. Then total update is exactly the same as MC.
Offline equivalence of forward and backward TD: Updates are accumulated within episode but applied in batch at the end of episode.
Online equivalence of forward and backward TD: TD(λ) updates are applied online at each step within episode Forward and backward-view TD(λ) are slightly different. NEW: Exact online TD(λ) achieves perfect equivalence. By using a slightly different form of eligibility trace.
Epsilon greedy exploration: Simplest idea for ensuring continual exploration. All m actions are tried with non-zero probability. With probability 1 − epsilon choose the greedy action. With probability epsilon choose an action at random.
Greedy and epsilon greedy policies: A greedy policy means the Agent constantly performs the action that is believed to yield the highest expected reward. Obviously, such a policy will not allow the Agent to explore at all. In order to still allow some exploration, an ε-greedy policy is often used instead: a number (named ε) in the range of [0,1] is selected, and prior selecting an action, a random number in the range of [0,1] is selected. if that number is larger than ε, the greedy action is selected — but if it's lower, a random action is selected. Note that if ε=0, the policy becomes the greedy policy, and if ε=1, always explore.
E-greedy: Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. The epsilon-greedy, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.
Sarsa (on policy TD control): As with Monte Carlo methods, we face the need to trade off exploration and exploitation. The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate qpie(s, a) for the current behavior policy pie and for all states s and actions a. This can be done using essentially the same TD method described above for learning vpie. Now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$ This update is done after every transition from a nonterminal state St. It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate qpie for the behavior policy pie, and at the same time change pie toward greediness with respect to qpie. The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use "-greedy or "-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy.
Off policy learning: Evaluate target policy π(a|s) to compute vπ(s) or qπ(s| a). While following behaviour policy μ(a|s) {S1; A1; R2; … ; ST} ∼ μ. Learn from observing humans or other agents. Re-use experience generated from old policies π1; π2; :::; πt−1. Learn about optimal policy while following exploratory policy. Learn about multiple policies while following one policy.

Qlearning: We now consider off-policy learning of action-values Q(s, a). No importance sampling is required Next action is chosen using behaviour policy At+1 ∼ µ(·, St). But we consider alternative successor action A' ∼ π(·, St). And update Q(St, At) towards value of alternative action. In this case, the learned action-value function, Q, directly approximates q*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. The rule (6.8) updates a state–action pair, so the top node, the root of the update, must be a small, filled action node. The update is also from action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next action" nodes with an arc across them.
Online policy: Typically the experiences are collected using the latest learned policy, and then using that experience to improve the policy. This is sort of online interaction. The agent interacts with the environment to collect the samples. In on-policy reinforcement learning, the policy πk is updated with data collected by πk itself. We optimise the current policy πk and use it to determine what spaces and actions to explore and sample next. That means we will try to improve the same policy that the agent is already using for action selection. Policy used for data generation is called behaviour policy. Behaviour policy == Policy used for action selection. Examples: Policy Iteration, Sarsa.
Off policy: In the classic off-policy setting, the agent's experience is appended to a data buffer (also called a replay buffer) D, and each new policy πk collects additional data, such that D is composed of samples from π0, π1, . . . , πk, and all of this data is used to train an updated new policy πk+1. The agent interacts with the environment to collect the samples. Off-policy learning allows the use of older samples (collected using the older policies) in the calculation. To update the policy, experiences are sampled from a buffer which comprises experiences/interactions that are collected from its own predecessor policies. This improves sample efficiency since we don't need to recollect samples whenever a policy is changed. Behaviour policy ≠ Policy used for action selection. Examples: Q- learning.
Offline reinforcement learning: Offline reinforcement learning algorithms: those utilize previously collected data, without additional online data collection. The agent no longer has the ability to interact with the environment and collect additional transitions using the behaviour policy. The learning algorithm is provided with a static dataset of fixed interaction, D, and must learn the best policy it can using this dataset. The learning algorithm doesn't have access to additional data as it cannot interact with the environment. This closely resembles the standard supervised learning problem statement, and we can regard D as the training set for the policy. Offline reinforcement learning algorithms hold tremendous promise for making it possible to turn large datasets into powerful decision making engines. No behaviour policy. Example batch reinforcement learning.
Double Q learning: One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them Q1(a) and Q2(a), each an estimate of the true value q(a), for all a 2 A. We could then use one estimate, say Q1, to determine the maximizing action A⇤ = argmaxa Q1(a), and the other, Q2, to provide the estimate of its value, Q2(A⇤) = Q2(argmaxa Q1(a)). This estimate will then be unbiased in the sense that E[Q2(A⇤)] = q(A⇤). We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate Q1(argmaxa Q2(a)). This is the idea of double learning. The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning. If the coin comes up tails, then the same update is done with Q1 and Q2 switched, so that Q2 is updated. The two approximate value functions are treated completely symmetrically.

Generalization: A generalisation problem occurs when the training and testing context sets are different, and the policy then learns to rely on features of the training environments which may change at test time. Conceptualising generalisation this way leads us to classify generalisation methods into three categories: Those that try and increase the similarity between the training and testing data and objectives, those that explicitly handle the difference in features between training and testing, and those that handle RL-specific generalisation problems or optimisation improvements.