1. Reinforcement learning

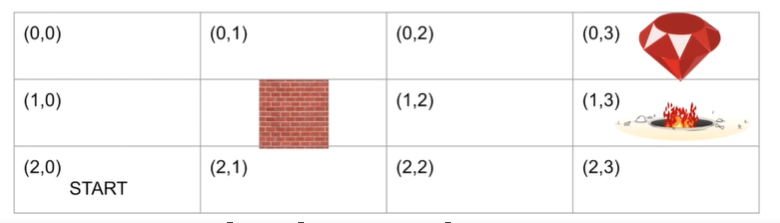
* Supervised learning – static function
* RNN – still a static function (input varies over ‘time’)
* RL – more like a loop
* A goal to achieve
* Time
* Plan for the future
* In supervised learning, training set must include labels (assigned by humans)
* E.g. Supervised driving – given an image, can you give it a target?

1. RL approach

* **RL uses goals rather than targets**
* E.g. solve a maze
* Goal: find the exit
* Do not need to label the correct direction for each maze position
* Magically, RL will figure out what to do during the preceding steps leading up to the goal (i.e. **planning for the future**)

1. RL terminology

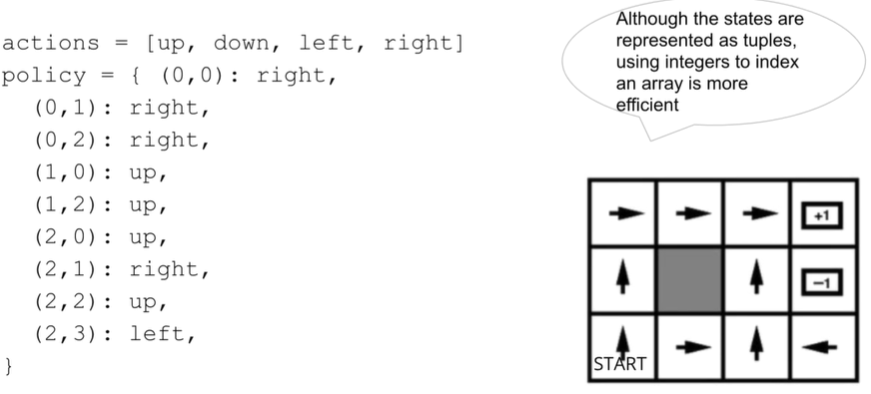
* Main objects – **agent and environment**
* E.g. Tic tac toe
* Environment: the computer program that implements the tic-tac-toe game
* This itself may contain some form of AI that acts as the other player
* A bunch of if-statements?
* Imagine thee game has an API
* The agent interfaces with the game (via the API) – picking the best move
* **Episode**:
* With tic-tac-toe, I will eventually win/lose, and then I can restart
* ML algorithms learn using data
* Playing tic-tac-toe multiple times allows us to collect more data to learn from
* Colloquially, you might call these games / rounds / matches, but in RL we call them episodes
* **An environment is episodic if it ends and you can start a new episode**
* No relationship between one episode and the next
* There are non-episodic environments too, e.g. stock market
* It can go on forever, there is no end
* E.g.: online ads – agent should choose the right ad to maximize revenue -> no end to an online ad service
* Infinite horizon
* **States, actions, rewards**
* Describe how the agent and environment interact
* E.g. tic-tac-toe
  + State – for each location, what’s there? X/O/empty
  + Action – where to place the next X/O
  + Reward – a number, received at each step of the game (e.g., +1 for win, -1 for lose, 0 for draw)
* E.g. Maze
  + State: position in maze
  + Action: direction you can go (up/down/left/right)
  + Reward: ?
  + +1 for solving the maze, 0 otherwise
* Is this good?
  + Suppose the agent plays 10000 eps and never finds the exit (assume ep ends at 100 steps)
  + The agent learns that no matter what it does, it still gets the same reward (0)
* No incentive to do better
* Better reward: assign -1 upon arrival in every state
* Reward is now maximized by completing the maze as fast as possible
* Remove any bias you have about the connotations of the word ‘reward’
  + Reward does not necessarily means ‘good thing’ in RL
  + A real number: +ve, -ve, or 0
  + The agent will try to maximize its rewards (-100 > -1million)
* E.g. Breakout: states, actions, rewards
* Several options for state
* One (actually real) option is to read the game’s RAM
* Can be thought of as a proxy to the previously defined ‘perfect’ state
* In contemporary applications, we simply use screenshots of the game
* RL agent must interpret images of the game, just as we humans do!
* CNN would be useful here >< Complication: with just a single screenshot, we have no concept of movement
* Important: the state need not be only what I observe at a single moment
* The state can be derived from current and past observations
* In the famous DQN paper, they used 4 consecutive frames to represent the state
* Action: just the different buttons / inputs on the joystick control pad
* Reward: +1 every time you remove a block
* **Spaces**:
* High-level concepts -> math that helps solve RL problems
* Describe state spaces and action spaces – set
* **State space** – set of all possible states
* **Action space** – set of all possible actions
* Canonical RL example – Gridworld
  + State space = {(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3)}
  + Action space = {up, down, left, right}



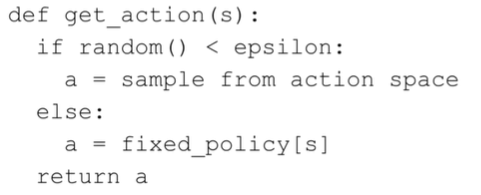
* Large state spaces
* State space is complicated
* E.g. tic-tac-toe: Action space is simple: just the different places you can put your X/O
* E.g. breakout:
  + Technically, it’s the screen resolution \* 2^24
  + Usually images, like time series, are considered continuous-valued
  + They appear discrete because computers have finite precision, values must be quantized
  + In some environments, actions can also be continuous / infinite
* **Summary**:
* Agent and environment
  + Environment: the world or the game you are teaching your agent to win
  + Agent: your computer program, the one that will learn
* Episode: one round / match of a game
* States, actions, rewards
* Reward: a number that the agent tries to maximize (+ve, -ve, 0)
* Action: what the agent does in the environment
* State: what we observe in the environment, but can be derived from one or more observations
* Last state of an episode – terminal state
* State space and action space – set of all states / actions

1. States, actions, rewards, policies

* How do we encode states and actions in code?
* Rewards are just numbers, no need to encode them
* **States**:
* Can be discrete or continuous
  + Discrete e.g.: Tic-tac-toe – state is a specific configuration of the board
  + Continuous e.g.: robot with sensors – camera, microphone, gyroscope, GPS, proximity sensor, etc.
* Digression to supervised learning
  + If targets are discrete, encode them as 0, 1, …, K-1
  + If targets are continuous, store them in a vector (images are stored in tensors)
* In RL: generally, state will be stored as tensor with 1 or more dimensions
* **Policies**
* How do you represent a policy in math?
* Policy: **what the agent uses to determine what action to perform** (given a state)
* Policy yields an action given only the current state
* Does not use any combination of past states or any rewards
* The state theoretically could be made up of past observations or rewards >< unconventional
* **Policy as a function or mapping**
* Input is the state s (dictionary key)
* Output is an action a (dictionary value)
* Mathematical representation?
* An example policy



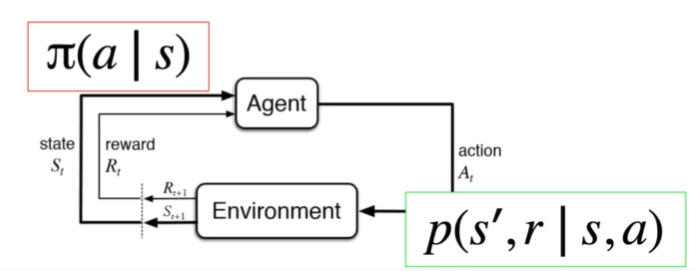
* This approach is somewhat limited
* What if we have an infinite state space?
* We cannot have an infinite number of dictionary keys
* Doesn’t allow us to explore the environment
* Think of training an agent like teaching a baby
* Must try new things to build intuition
* Makes sense to policies to be stochastic / random
* A more general way to represent policies is with probability
* **Policy as probability**
* Allows agent to have a degree of randomness (or not)
* One method is called **epsilon-greedy** (common value of 0.1)



* Continuous state spaces – s is a D-dimensional vector
* Policy parameters – W (shape of W is D x |A|)
* How do we get prob from a vector? Softmax function
* For a given state s, we can calculate the prob we should perform each action,
* To choose an action: sample from or take the argmax
* Is only the state needed?
* How can an agent make an intelligent decision based only on the current state?
* If you think in terms of SL, you may insist that this requires a target (how else would an agent know what to do for this state?)
* **In RL, an agent can learn to plan for the future by gaining experience**
* Instead of an explicit target, it will try to **maximize future rewards**

1. Markov decision process (MDP)

* Problems, frameworks, solutions
* Goal: come up with a framework
* Using that, we can more accurately define the problem, and find a solution
* The **Markov assumption**
* In RL, the main assumption we make is the Markov assumption
* Normally, this is discussed in the context of sequence modeling
* E.g.: predict whether tomorrow weather will be rainy, sunny, or cloudy
* The Markov assumption: tomorrow’s weather depends only on today’s weather, but not on any earlier day
* E.g.: predict the next word in a sentence
  + Preceding word is ‘lazy’
* What is the next word?
* In general, we say that **the state at time t depends only on the state at time t-1**
* By itself, the Markov assumption is weak
* **Recall: we can make the state whatever we want**
* Can consist of 3 or 4 words / observations
* MDP: describes a RL system
* Previously, we defined the Markov assumption in terms of state only
* **In an MDP, we describe the environment with the state-transition probability**
* **State-transition**
* is the most general
* But sometimes (e.g., maze example), the reward is deterministic
* No need to model it with prob.
* State transition prob:
* Reward function:
* What is the usefulness of the state-transition probability?
  + In complex games, the state space is infeasible to enumerate – how can we calculate this prob?
* In algorithms such as Q-learning, it’s not used at all
* So what’s the point? Think of MDPs as a stepping stone
* Allows us to build our way up to practical RL solutions
* **State-transition probability**?
* State may not capture the complete info about the game
* In tic-tac-toe, there is another player – you can’t deterministically predict their move
* Physical systems involve chaos theory – even if you know the physical laws of motion, you still cannot deterministically predict the future
* The further you try to predict, the more inaccurate your predictions
* We refer to state-transition prob as **environment dynamics**
* The language of probability
* By representing both agent and environment as mathematical objects (probs) we can describe a system using equations, which then allows us to solve those equations (need a problem to find a solution)



1. The return

* **Maximizing rewards**
* Rewards may be structured differently in different environments
* Tic-tac-toe: +1 for winning, -1 for losing
* Maze: -1 for every step
* What exactly does it mean to maximize reward?
* Does it mean ‘maximize the reward I get on the next step’?
* Does it mean ‘maximize total reward over entire episode’?
* Future rewards
* A more accurate statement would be: the agent wants to **maximize the sum of future rewards**
* We can’t maximize the rewards we already got
* We don’t want to only maximize the reward on the next step
* The agent would not be incentivized to do anything useful – each step yields the same reward (e.g. a maze gives you -1 on every step)
* By trying to maximize the sum of future rewards, the agent plans for the future
* E.g. preparing for a math exam
* You don’t receive any reward until you finish the exam – your reward signal is your exam grade
* Imagine the actions required to maximize the reward – study/ homework/ forego socializing -> Those do not sound very rewarding at all
* Immediate gratification does not work
* Must use long-term planning to maximize future rewards
* **The return**
* **Sum of future rewards**
* Some people also call it the **utility**
* Infinite horizon MDP – a game that never end
* Will your return be inf?
* Not if we use **discounting**
* Is used for episodic tasks too
* We call the **discount factor** – usually a number close to 1 (0.9, 0.99, etc.)
* Works like money – I want to receive $100 now, not in 10 years
* Recursiveness
* The return can be **defined in terms of a later return**
* Return at time t = Reward at time t+1 + gamma \* return at time t+1

1. Value functions and the Bellman equation

* Expressing the problem as an equation
* Let’s start with expected values (the mean)
* E.g. model the heights of 1000 students as a Gaussian distribution
* Average height to be 70 inch, standard deviation to be 4 inch
* The expected value is NOT the value you expect to get
* The expected value has a precise definition for both discrete and continuous random variables
* Discrete: -> weighted sum of all the possible values of a random variable where the weights are the probability of that value
* Continuous:
* Coin example:
* Suppose p(heads) = 0.6 -> biased coin
* Heads = 1, tails = 0
* E(X) = 0.6 \* 1 + 0.4 \* 0 = 0.6
* Why expected values?
* The reward (and hence return, since it’s the sum of rewards) is a random variable
* If I play the same game for 100 eps with the same policy and same environment, I will get different rewards each time
* Both environment dynamics and policy are probabilistic
* Makes sense not to think of a single return, but the expected value of the return
* Maximize the expected sum of future rewards
* Value function
* Expected return – the value function
* Conditioned on the state s at time t
* Value of state s
* Recursiveness:
  + Return can be defined recursively
* Value function can also be defined recursively
* Why do this?
* In order to find a solution, we must have a problem to solve
* Express the expected value in terms of prob. distributions
* **Bellman Equation**
* Implications
* We only used the rules of math (prob.) to arrive here
* Its physical meaning is deeper: and come from different physical processes
* One represents the agent, the other is the environment
* Problem and solution
* We now have a problem -> solution
* There are multiple possible policies - some good, some bad
* How can we tell which are good and which are bad?
* Check value functions
* Finding the value function for a given policy is called the prediction problem
* How is this simple
* Just multiplication and addition
* is just a function we wrote in code – we know what it is
* Assume we know too
* System of linear equations
* If we have 3 states, the Bellman equation would boil down to the expression below (b’s and c’s are just constants)

1. What does it mean to learn?

* In RL, there are 2 main types of tasks:
* Prediction problem: Given , find the corresponding value function V(s)
* Control problem: Find the optimal policy , which yields maximum V(s)
* Action-value
* V(s) = state-value function
* Q(s, a) = action-value function (usually we just call it a ‘Q-table’)
* Bellman equation for Q
  + Note: no sum over action, because it is given
* Space complexity
* How much space it takes to store V(s) and Q(s)
* Assume a finite set of discrete states and actions
* We have |S| states and |A| actions
* V(s) can be stored in an array of size |S| -> linear complexity
* Q(s, a) can be stored in a 2D array of size |S|x|A|-> quadratic complexity
* The optimal policy
* The optimal policy is the best policy – maximizes the value (for all states)
* We can say is better than if V(s) for is greater than V(s) for for all states s in the state space
* The best policy and the best value function
* The best value function
* The best policy
* Relationship between Q and V
* The optimal action-value
* From Q\* we can easily obtain V\*
* What about the action-value?
* If we find the optimal action-value, it is very easy to choose the best action given any state
* The optimal policy is not unique – multiple policies can lead to best value
* Suffice to find one
* Finding the optimal policy
* Can we use the definition directly? Yes
* Control problem
* Suppose we’re playing Gridworld or tic-tac-toe
* State space and action space are finite
* Naïve search will suffice

A white text with black text

Description automatically generated with medium confidence

* Impractical

1. Solving the Bellman equation with RL – Prediction problem

* Recap of simple / naïve solutions to RL
* Prediction – find V(s)
* Control – find
* Prediction: if we know all the probs -> simple linear algebra problem
* Control: exhaustively loop through all possible policies, return the one that yields the best V(s)
* Are these approaches realistic?
* Prediction problem: we need to know
* Not realistic
* The state space is also extremely large – cannot enumerate all states and their transition probs
* Control: can we enumerate all possible policies?
  + Consider we have |S| possible states and |A| possible actions
  + Total # of policies is -> exponential growth
* Would not work at all if |S| or |A| is infinite
* The solution
* Recall: the expected value is the mean
* To calculate the expected value, we must know the distribution
* Key point: we can estimate the mean with the sample mean
* As N -> inf, estimate becomes more accurate
* Application to RL
* Value function -> simply the expected return
* Sample many returns to estimate the value of each state
* G(t) means generic random variable – return at time t
* g(i, s) means a sample – the ith time reaching state s
* where do the samples come from?
* How do we sample a return?
* Every time we play an ep, even if we use the same policy in the same environment, we get a different result
* Both the policy and environment dynamics are probabilistic
* Simply playing the game works
* Pseudocode – Monte Carlo approach
* Prediction: given a policy, find the value function
* High-level idea: playing an ep yields a series of states and corresponding rewards

A screenshot of a computer

Description automatically generated

* Calculating returns
* Go backward and use the recursive definition
* since there are no future states
* Value of a terminal state is 0

A screenshot of a computer program

Description automatically generated

* Pseudocode for prediction
* Just do the same thing 100s or 1000s of times to collect samples

A screen shot of a computer code

Description automatically generated

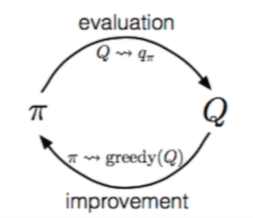
* For each state, take the average return of the samples

A close-up of a list

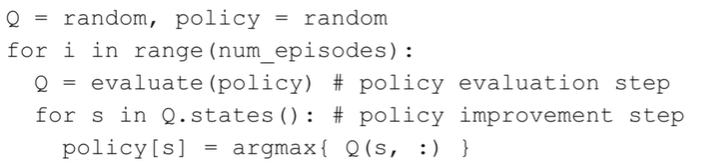
Description automatically generated

1. Solving the Bellman equation with RL – Control problem

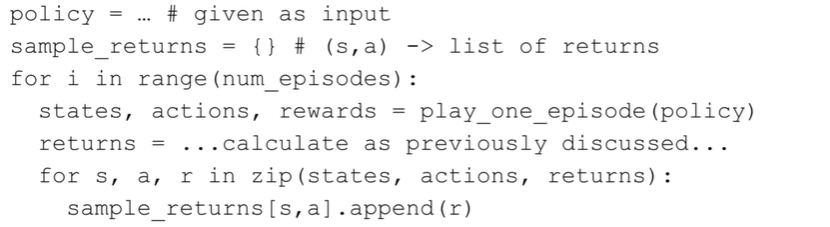
* General pattern: for prediction, we want V(s)
* For control, we want Q(s, a)
* Select the best action to perform
* Policy iteration and policy improvement
* Given a policy, we can use Monte Carlo to evaluate the value function V(s) or Q(s,a)
* Given the action-value, we can choose what we currently believe to be the best action, using the argmax
* These 2 facts are interdependent
* Repeat in a loop: find Q(s,a) given , find as the argmax over Q(s,a)



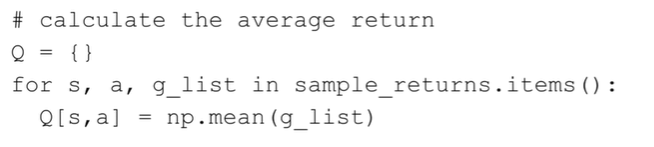
* Its been proven that this process leads to a monotonic improvement in the policy
* By repeating the process until convergence, we arrive at thee optimal policy
* Monte Carlo approach



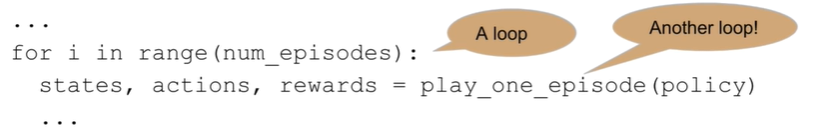
* Evaluating Q(s, a)
* Earlier, we looked at evaluating V(s)
* To evaluate Q(s, a) we must keep track of states, actions, and rewards
* We end up with triples:



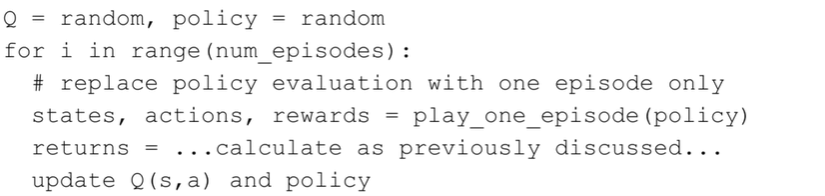
* Take the sample mean again, where the key is now (s, a)



* Improving our solution
* Our solution works, but it’s not ideal
* V(s) has to store |S| values
* Q(s, a) has to store |S| x |A| values
* Monte Carlo sampling – the more samples you collect, the more accurate your estimate
* Since Q(s, a) has more values to estimate, we must obtain many more samples to get an accurate estimate
* Problem #2: nested loops
  + Very slow



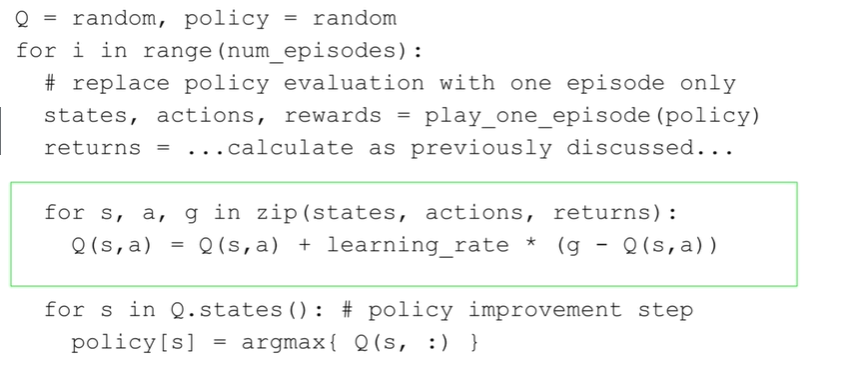
* Generalized policy iteration
* Idea: in evaluation step, instead of playing multiple episodes to obtain a Monte Carlo estimate of the value, just play one episode
* We’ll only get a single series of (states, actions, returns)
* Use this to update a single running copy of Q(s, a) and policy



* Is calculating sample mean efficient?
* Efficient sample means
* Taking the sum of N values is O(n)
* Instead, let’s see how we can **calculate the N’th sample mean from the (N-1)’th sample mean**
* This not only looks like **gradient descent**, it is!

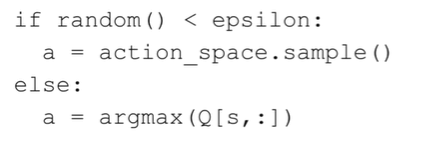
Where:

* + – decays over time
* Rewrite the same equation in terms of Q(s, a) and g (the sample)
* Note: this is an assignment, not an equality
* We only have 1 copy of Q – all the samples we collect on each iteration come from different policies (because policy is updated each step)
* The samples which we take the sample mean of do not come from the same distribution
* Intuition:
  + Oldest samples come from oldest policies – they matter less
  + Newest samples come from newest policies – they matter more
* Constant learning rate
* 1/N learning rate gives us the standard (equally weighted) average
* All samples matter equally
* We don’t want that (old samples should matter less)
* Constant learning rate gives us the exponentially decaying average
* Pseudocode



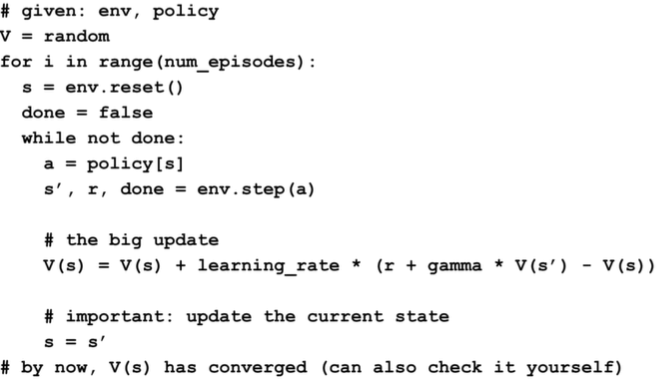
1. Epsilon-Greedy

* Policy as a distribution
* Advantages of treating policy as a distribution
* Entire MDP is just 2 probabilities
* Allows us to express MDP mathematically (and find a solution)
* Important: in order to find the best action, we must know the result of performing those actions
* Problem
* Currently, we take the argmax to determine the action – but Q will not change that much
* Suppose we have 3 actions:
* Assume Q initialized to
* Assume all rewards are +ve
* Maybe is updated to 2 after a few episodes
* We’ll always choose this action over the other 2
* We’ll never know the true values of because we can never use those actions
* The explore-exploit dilemma
* The explore-exploit dilemma
* The more samples you collect, the more accurate your estimate >< time and money wasted
* The dilemma: we must balance exploration and exploitation
* Explore: collect more data to determine which machine is best
* Exploit: play only the best machine, to make more money
* Epsilon-greedy
* We have a small prob (a hyperparameter called) of choosing a random action
* Otherwise we will perform the greedy action (argmax over Q(s, : ))
* As we collect more data, Q will become more accurate
* Code:

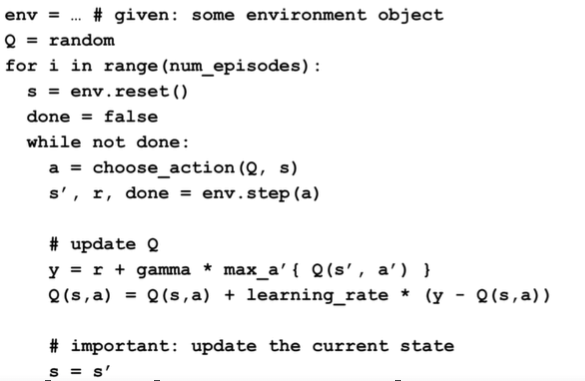


1. Q-learning

* Recap:
* Define relevant terms: agent, environment, state, action, reward, etc.
* Mathematical structure: MDP (Markov decision process)
* Prediction problem: find the value given a policy
* Control problem: find the optimal policy
* If we know the prob, these are easy
* If we don’t we can use Monte Carlo
* Monte Carlo limitations
* In order to calculate returns, we must wait until the ep ends
* This is by definition: the return is the sum of rewards until the end of the episode
* What if we have very long episodes, or environments with no terminal state?
* Even if the episode ends eventually, MC is still not ideal because the agent must perform suboptimally for a long time before improvement
* Temporal difference methods
* We’ll again make use of the recursive structure of the return
* Helped us create and solve the Bellman equation
* Monte Carlo (MC) is an approximation to an expected value problem
* Temporal difference (TD) is an approximation to MC
* An approximation of an approximation
* Monte Carlo update trick
* We converted the usual expression for sample mean into a gradient descent expression (using only algebra)
* Instead of keeping old samples around and adding them up every time, we can simply calculate the new estimate from the old estimate
* The gradient descent perspective
* Let’s test this theory by defining J as the squared error between a sample (g) and prediction V(s)
* The gradient: we can ignore the 2 because it can be absorbed into the learning rate
* Gradient Ascent vs. Gradient descent
* There is no difference between the expressions for gradient ascent / descent (the -ve sign is simply in a different place)
* + sign is more natural when derived from sample mean
* - sign is more natural when derived from gradient descent or loss
* Combining these ideas
* Idea 1: updating the value function using the exponentially decaying average is the same as gradient descent
* Idea 2: the return can be defined recursively
* Estimate the return
* Instead of using the real return (g) as the target, let’s estimate it
* Collect only the next reward (r), and estimate the rest (V(s’))
* V(s’) is the expected value of g’
* Now we only have to wait one step to update the model, instead of waiting until the ep ends
* Bootstrapped estimate of the return
* Pseudocode (prediction problem)



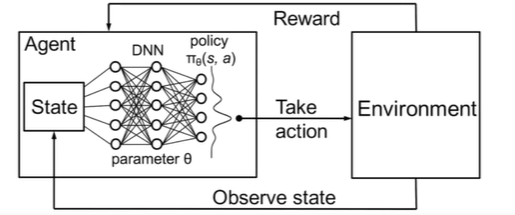
* Semi-gradient
* Target is
* Prediction is V(s)
* In supervised learning, we are given the target as part of the dataset
* Here, we are predicting the target itself
* Both V(s) and V(s’) are model predictions
* This is not a true gradient, but a semi-gradient
* Q-learning (Control problem)
* We look at Q rather than V
* Imagine the innermost part of the loop – we must do 2 things
  + Choose an action
  + Update Q
* Important: it doesn’t matter what action (a’) we take next
* Advantage: we don’t have to wait until we determine the next action to update Q
* Q-learning is an off-policy algorithm (the update may not match the actions taken)
* Pseudocode



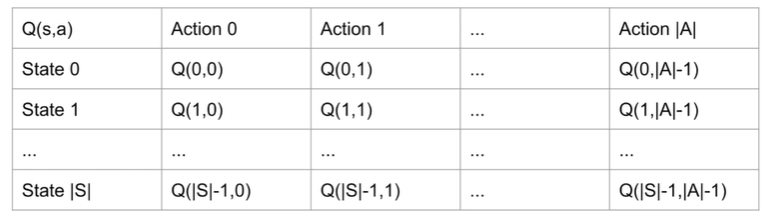
* Summary:
* Monte Carlo won’t work for infinitely long episodes
* The return can be defined recursively -> approximate it with the next reward and next value
* RL begins to look like supervised learning, except the target is partly a model estimate
* The update is gradient descent on the Q-table
* We can update Q on every step
* Online learning because the agent learns as data is collected

1. Deep Q-learning – part 1

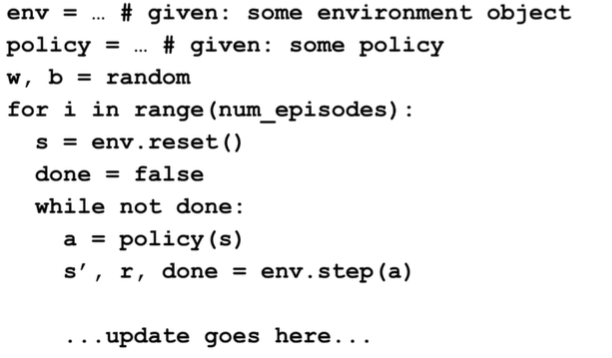
* Q-learning using neural networks -> deep Q-learning
* Q-learning involves finding the optimal Q\*(s,a) and the corresponding optimal policy
* So far, states/actions are categorical (encoded by integers starting at 0)



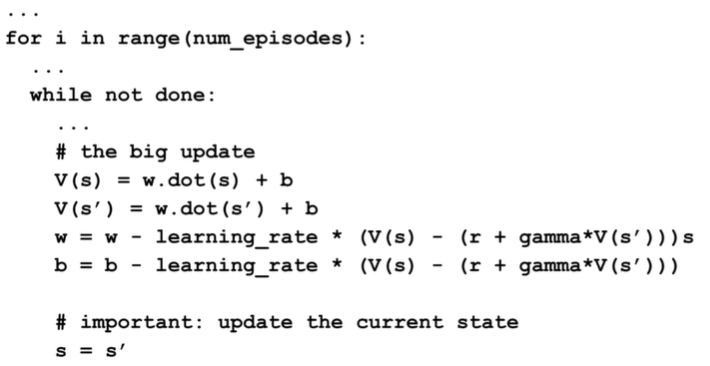
* Q-table: Q(s, a) – states and actions -> tabular methods



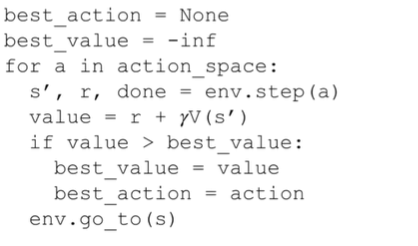
* Infinite state or action spaces
* Q-tables are no longer an option
* One option: use binning to force a finite discrete set of states / actions
* A more flexible approach would be to use ML (function approximation)
* We call these approximation methods (as opposed to tabular methods)
* In DQN, the state space (possibly) infinite but the action space is discrete
* Suppose our state is a vector s, and we have 2 actions
* Combine all weights into a weight matrix and all bias terms into a bias vector
* Q value given the state s over all actions
* If state has D components and we have K actions, then
  + W is of shape D x K
  + b is a vector of length K
* Notation abuse colon (:) means ‘select all elements of this dimension’
* How to update?
* When discussing temporal difference and MC learning, we updated V(s) and Q(s, a) directly
* They were just values in a table
* Now we need to update W and b
* **V(s) – prediction problem**
* Supervised learning perspective
* **Target**:
* If s’ is terminal, target = r
* **Prediction**:
* What we should **update**: W and b
* Even though V(s’) and V(s) depend on the parameters, we only differentiate wrt V(s)
* Instead of updating V directly, we are updating W and b
* Pseudocode:



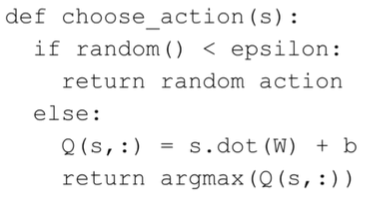
* Update



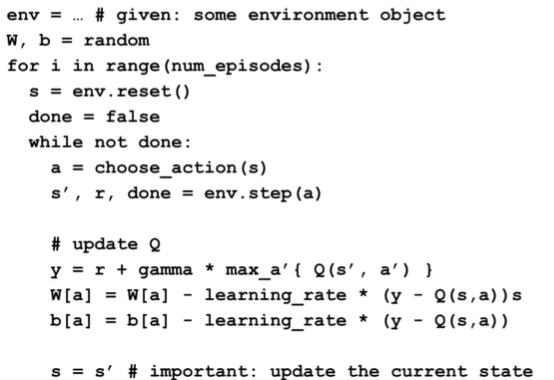
* Prediction to Control
* Approximating V(s) is not that useful
* What if we are using V(s) to help us choose an action?
* Normally, we would take the argmax of Q(s, a) over all a
* V(s) is not indexed by any action
* Using V(s) for policy improvement
* Pretend we have a very special environment, where we can try an action and go back to the previous state
* In real environments, we cannot simply go back to state s
* Not ideal solution



* **Use Q(s, a) instead**
* In this case, the **target** is
* If s’ is terminal, target is still only the reward r
* **Prediction** is Q(s, a)
* **Parameters to update** are still W and b
* Updating Q(s, a): **Only the components of W and b corresponding to the action taken (a) are updated** (i.e., W[a] and b[a])
* Equivalent to saying that the **error for actions not taken is zero** (important for implementation)
* Q-learning with function approximation
  + First, let’s look at choose\_action function

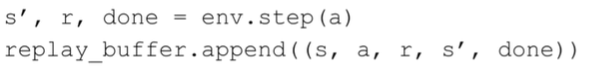


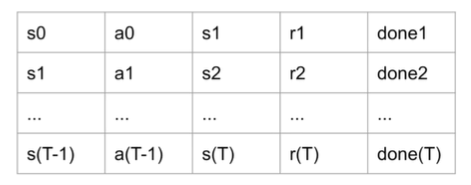
* Q-learning pseudocode



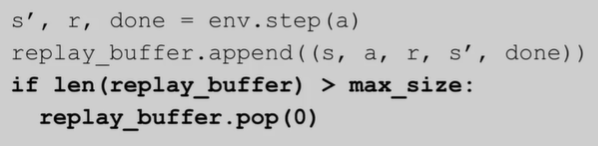
1. Deep Q learning – part 2

* Plug and play?
* Is it not trivial to simply replace the linear regression with a neural network (applying the rule ‘all ML interfaces are the same’)
  + Old:
  + New:
* This seems better since it doesn’t obvious we can make the assumption that Q is linearly dependent on s
* **Updating Q(s, a)**
* Conceptually, we only need to continue using gradient descent
* represents all params of the model here
* Gradient will be complicated, so we can use automatic differentiation
* Problem
* Training neural networks is inherently unstable
  + After awhile, you realize you can’t just choose any learning rate, any optimizer, any model architecture, etc.
  + Sometimes the cost will explode or model won’t train at all
  + Neural networks are sensitive whereas linear regression usually is not
* Temporal difference learning is itself unstable
  + Target is not a real target, it involves a model prediction
  + Loss is not a real loss, etc.
  + Everything is an approximation
* What happens when you combine the instability of neural networks with the instability of TD learning?
* Barely works
* That’s not to say there won’t be instances where you can just plug in a neural network and make it work
* But it won’t be as easy as linear regression
* Deep Q-learning
* There have been a number of approaches to DQN that have been developed over the years -> we’ll look at one of them
* The experience replay buffer / experience replay memory
* Previously, we used stochastic gradient descent
  + One sample at a time
* Batch gradient descent is more stable
  + Multiple samples at a time
* Sometimes, (e.g., in Tensorflow), people still call it SGD when working with batches
* Try different batch sizes (e.g., 1, 32, …, N) for intuition
* When using the full dataset, your loss should decrease monotonically
* Sample correlation
* We don’t like samples to be correlated
* Shuffle data on each epoch
* We’ll see how we can ‘randomize’ the replay buffer, so that we don’t always see (s, a, r, s’) tuples in the order they were encountered (high correlation)
* Replay buffer
* Conceptually, it’s just a list that stores (s, a, r, s’, done) tuples
* We can call them transitions
* In code

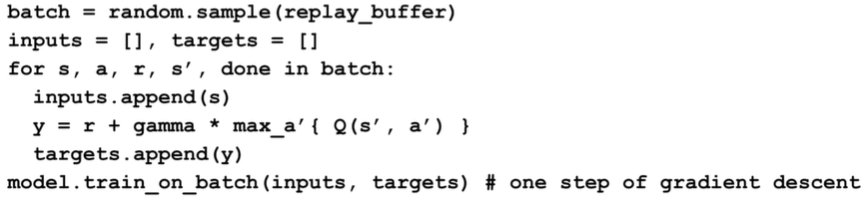




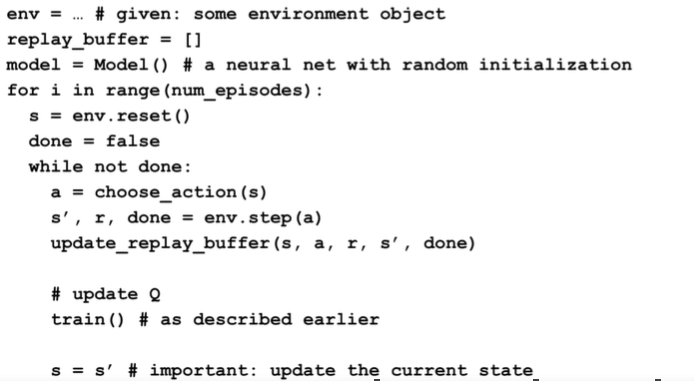
* Fixed size
* At some point, the transitions in the replay buffer will become too old (will correspond to a policy very different from what we now follow)
* In addition to adding new transitions, we will also remove old transitions when the buffer reaches its max size



* Batch update – what we want to do conceptually
* Sample a batch of transitions (tuples of (s, a, r, s’, done)) from the replay buffer
* Populate inputs and targets
* Do one step of gradient descent on the data

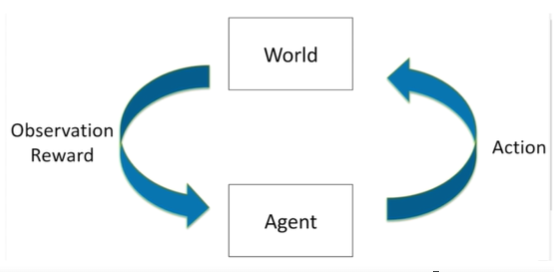


* **Pseudocode** (DQN)



1. How to learn RL

* Section summary
* What does it take to learn RL for real?
* Difficult to understand RL abstractly from a single section in a single course



* RL vs. (un)supervised learning
* Not a problem if you just want to use APIs
* It becomes a problem when you want to stop using APIs
* The turning point appears when you start RL
* No API for RL (at least not yet)
* Nearly impossible if you don’t have experience implementing ML algorithms – RL is just not a good place to start
* Learning RL for real
* Take a full course (or courses) on RL
* Learn tabular RL before approximation methods
* Learn about the 3 basic approaches
  + Dynamic programming
  + Monte Carlo
  + Temporal difference
* Spend time on implementation
* Then graduate to approximation methods (with linear models)
* Then apply deep learning
* Deep RL is very hard to get right, even if you are an expert programmer
* Subtle bugs everywhere