Value-based Deep Reinforcement Learning Methods

CS 4641 B: Machine Learning (Summer 2020)

Miguel Morales

07/13/2020

Admin

- Homework 4 has been released. It covers the two previous lectures.
- Instructions for the Project presentation have been released. Check Piazza for details.
- Make sure to contribute to Piazza/Lecture discussions.
- The two remaining lectures we will cover state-of-the-art deep reinforcement learning methods.
- Grading of HW1, HW2, Project Proposal is complete.
- Course website has been updated to reflect some of these: https://mimoralea.github.io/cs4641B-summer2020/
- Recommended Deep Learning specific courses:
  - https://course.fullstackdeeplearning.com/
  - https://atcold.github.io/pytorch-Deep-Learning/
  - https://cs230.stanford.edu/lecture/
  - http://cs231n.stanford.edu/

Outline

Reinforcement Learning with Function Approximation

Deep Reinforcement Learning

Value-based Methods

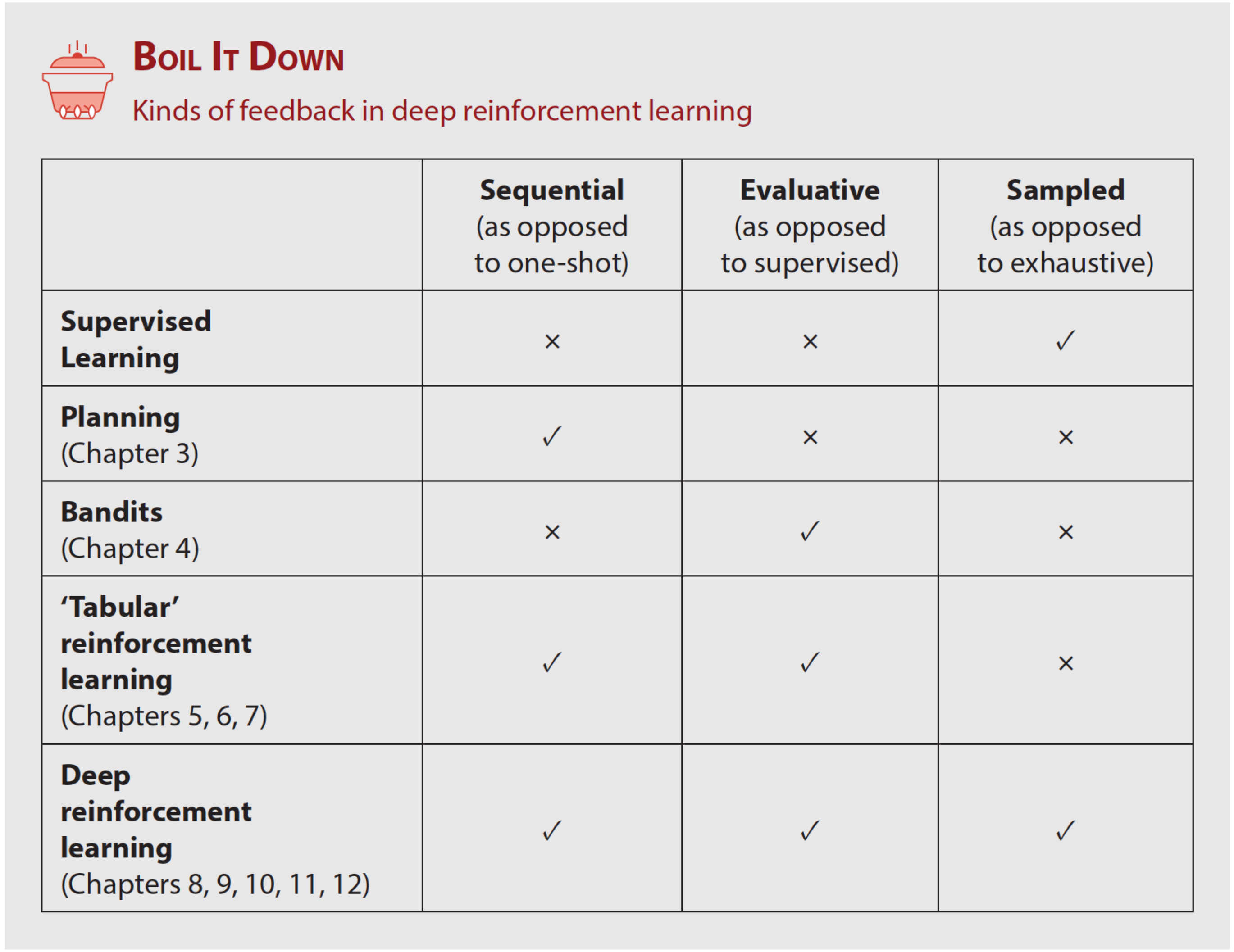
Outline

Reinforcement Learning with Function Approximation

Deep Reinforcement Learning

Value-based Methods

Recall the kinds of feedback RL agents learn from



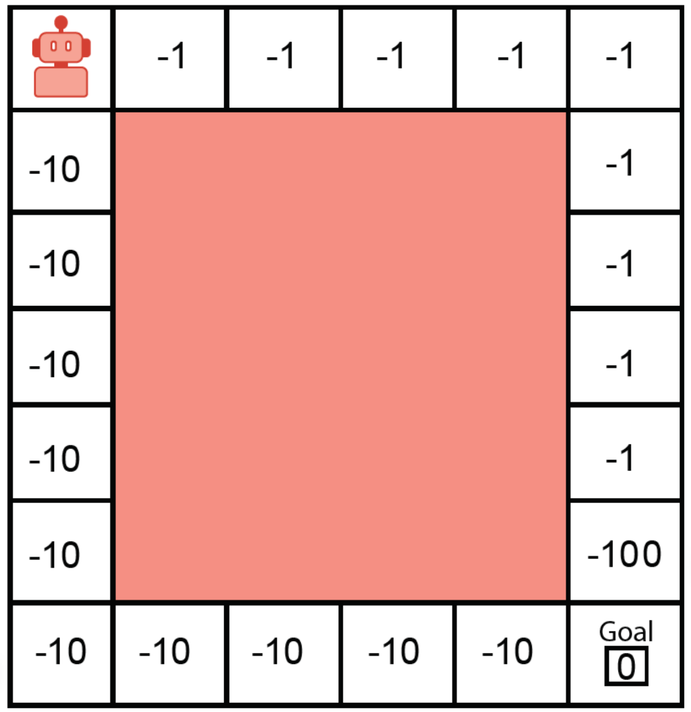
Boil It Down

Kinds of feedback in deep reinforcement learning

	Sequential (as opposed to one-shot)	**Evaluative** (as opposed to supervised)	**Sampled** (as opposed to exhaustive)
Supervised Learning	×	×	✓
Planning (Chapter 3)	✓	×	×
Bandits (Chapter 4)	×	✓	×
'Tabular' reinforcement learning (Chapters 5, 6, 7)	✓	✓	×
Deep reinforcement learning (Chapters 8, 9, 10, 11, 12)	✓	✓	✓

Recall what sequential feedback is

(1) Consider this environment in which one path looks obviously better than the other even after several steps.



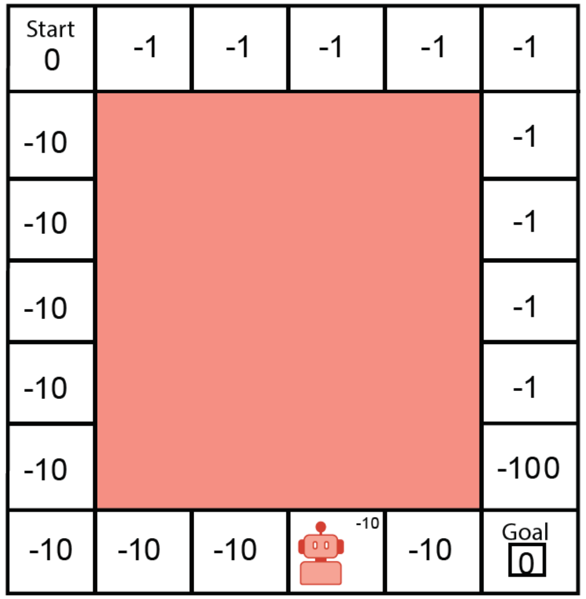
(2) But before the agent can complete this "better–looking" path, it will get a high penalty.

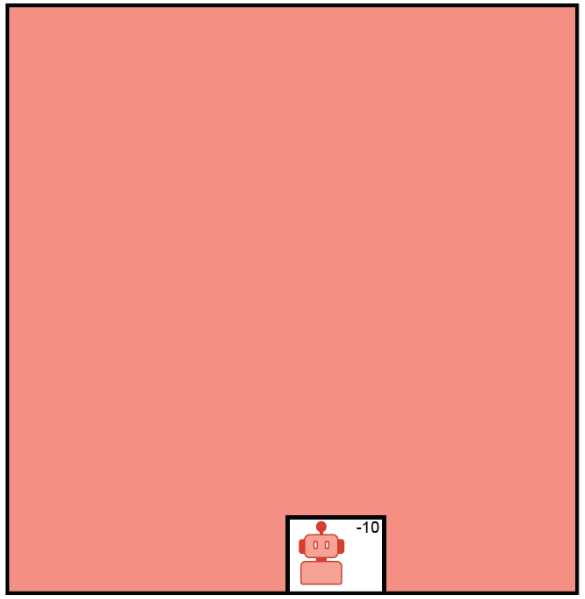
(3) This is the challenge of sequential feedback, and one of the reasons we use value functions to decide on actions, and not merely rewards.

Georgia Tech
CREATING THE NEXT

Recall what evaluative feedback is

(1) To understand the challenge of evaluative feedback you must be aware that agents don't see entire maps such as this one.





(2) Instead, they only see the current state and reward such as this one.

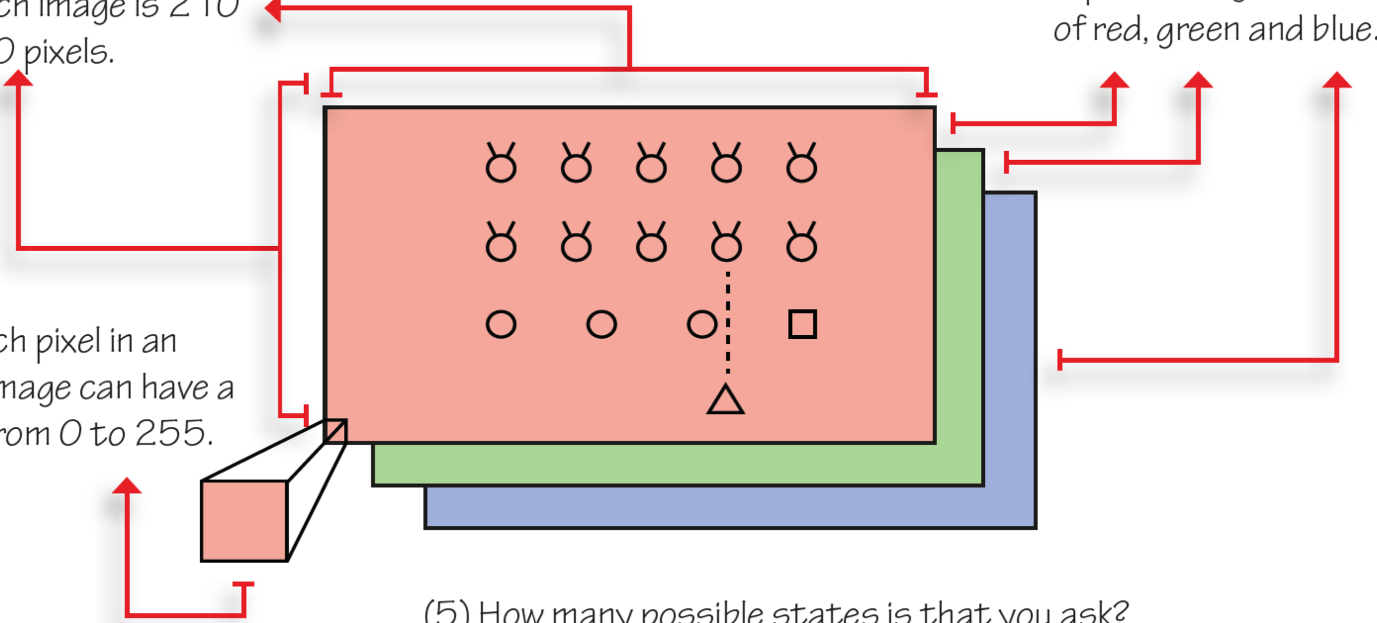
(3) So, is that -10 bad? Is it good?

Recall what sampled feedback is

(1) Imagine you are feeding your agent images as states.

(2) Each image is 210 by 160 pixels.

(3) With 3 channels representing the amount of red, green and blue.



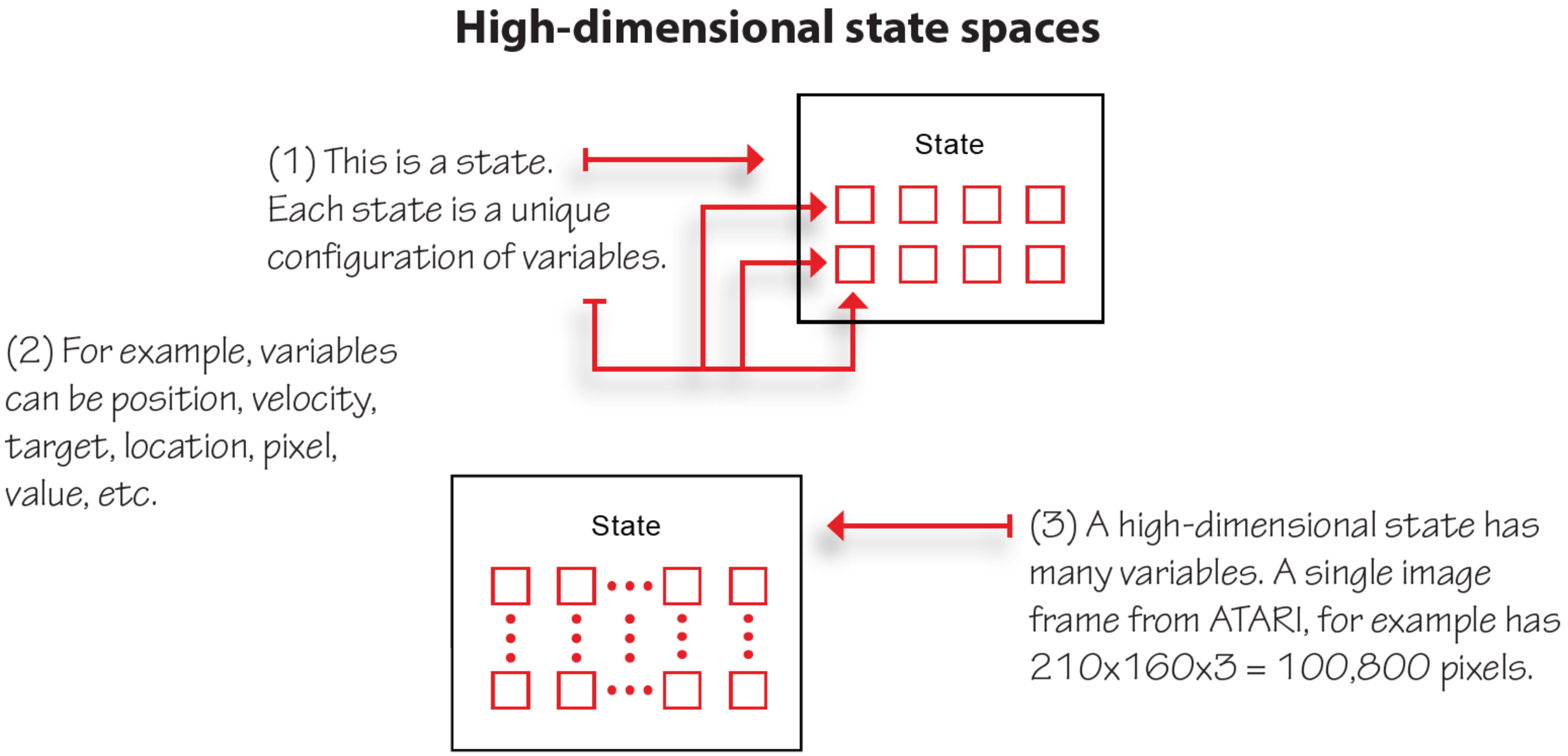
(4) Each pixel in an 8-bit image can have a value from 0 to 255.

(5) How many possible states is that you ask?

(6) That's $(255^3)^{210 \times 160} = (16{,}581{,}375)^{33{,}600}$ = a lot!

(7) For giggles, I ran this in Python and it returns a 242,580-digit number. To put it in perspective, the known, observable universe has between 10^{78} and 10^{82} atoms, which is an 83-digit number at most.

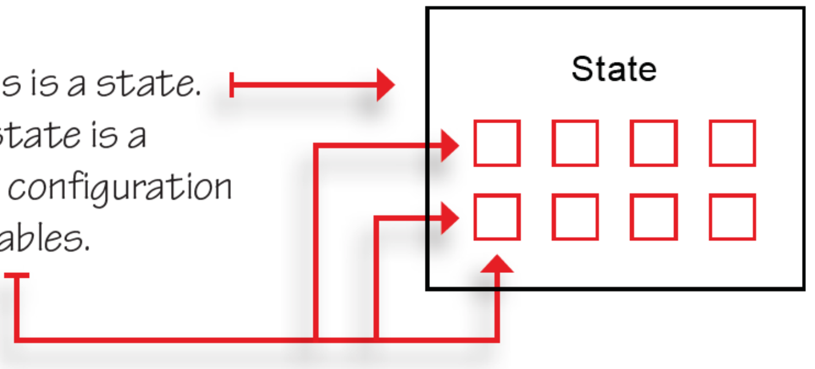
Why using function approximation?


High-dimensional state spaces
State
(1) This is a state. Each state is a unique configuration of variables.
(2) For example, variables can be position, velocity, target, location, pixel, value, etc.
State
(3) A high-dimensional state has many variables. A single image frame from ATARI, for example has 210x160x3 = 100,800 pixels.

Why using function approximation?

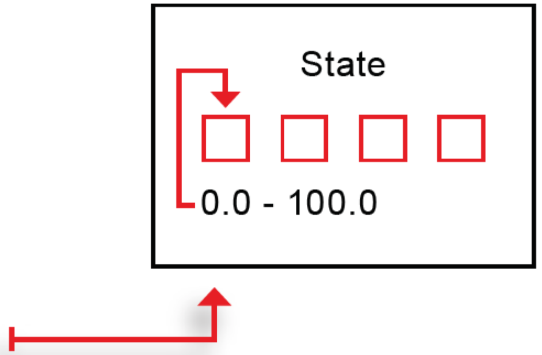
Continuous state spaces

(1) This is a state. Each state is a unique configuration of variables.



(2) Variables can be position, velocity, target, location, pixel, value, etc.

(3) A continuous state-space has at least one variable that can take on an infinite number of values. For example, position, angles, altitude, are variables that can have infinitesimal accuracy: say, 2.1, or 2.12, or 2.123, and so on.



Why using function approximation?

Refresh My Memory

Algorithms such as Value Iteration and Q-learning use tables for value functions

Value iteration is a method that takes in an MDP and derives an optimal policy for such MDP by calculating the optimal state-value function, v*. To do this, value iteration keeps track of the changing state-value function, v, over multiple iterations. In value iteration, the state-value function estimates are represented as a vector of values indexed by the states. This vector is stored with a lookup table for querying and updating estimates.

A state-value function

(1) A state-value function is indexed by the state, and it returns a value representing the expected reward to go at the state.

State	0	1	2	3	4	5
V	-3.5	1.4	0.2	1.1	-1.5	3.4

The Q-learning algorithm does not need an MDP and does not use a *state-value function*. Instead, in Q-learning, we estimate the values of the optimal *action-value function*, q*. Action-value functions are not vectors, but, instead, are represented by matrices. These matrices are 2-d tables indexed by states and actions.

An action-value function

States

Actions

Q	0	1	2	3
0	-1.5	-0.2	1.2	5.7
1	4.2	-2.1	2.7	6.1

An action-value function, Q, is indexed by the state and the action, and it returns a value representing the expected reward to go for taking that action at that state.

Boil It Down

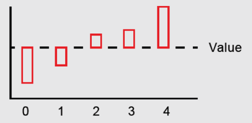
Function approximation can make our algorithms more efficient

In the cart-pole environment, we want to use generalization because it is a more efficient use of experiences. With function approximation, agents learn and exploit patterns with less data (and perhaps faster).

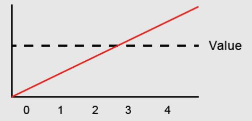
A state-value function with and without function approximation

(1) Imagine this state-value function. V =[-2.5, -1.1, 0.7, 3.2, 7.6]

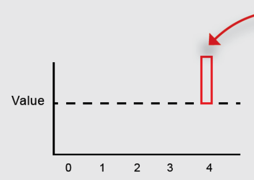
(2) Without function approximation, each value is independent.



(3) With function approximation the underlying relationship of the states can be learned and exploited.

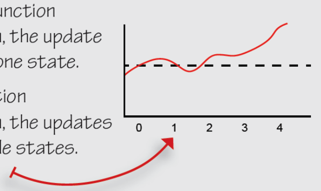


(4) The benefit of using function approximation is particularly obvious if you imagine these plots after just a single update.



(5) Without function approximation, the update only changes one state.

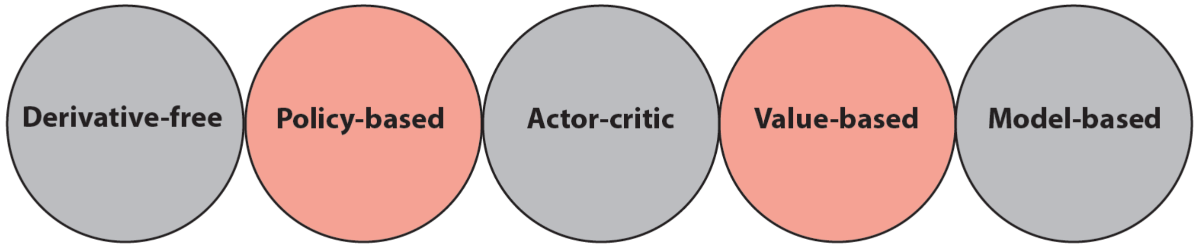
(6) With function approximation, the updates change multiple states.



(7) Of course, this is a simplified example, but it helps illustrate what's happening. What would be different in 'real' examples?
First, if we approximate an action-value function, Q, we would have to add another dimension.
Also, with non-linear function approximator, such as a neural network, more complex relationship can be discovered.

Types of RL approaches with FA

- Derivative-free
  - Using black-box optimization methods, such as Genetic Algorithms.
- Policy-based
  - Training a policy network.
- Actor-Critic
  - Training a policy and a value network.
- Value-based
  - Training a value network.
- Model-based
  - Training a transition and/or reward function network.



Outline

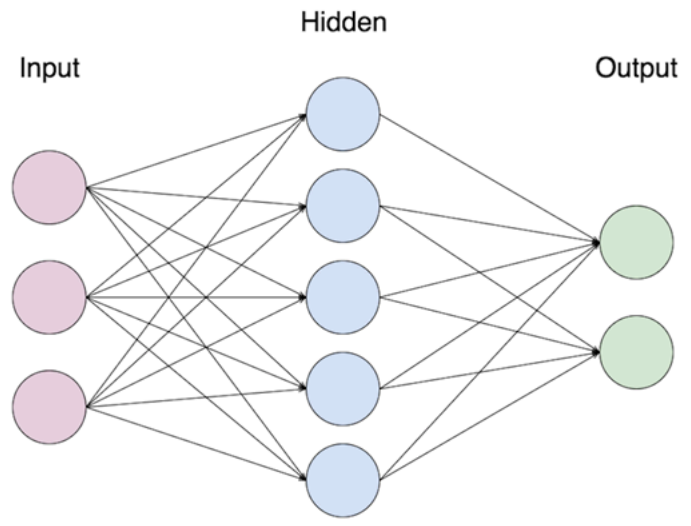
Reinforcement Learning with Function Approximation

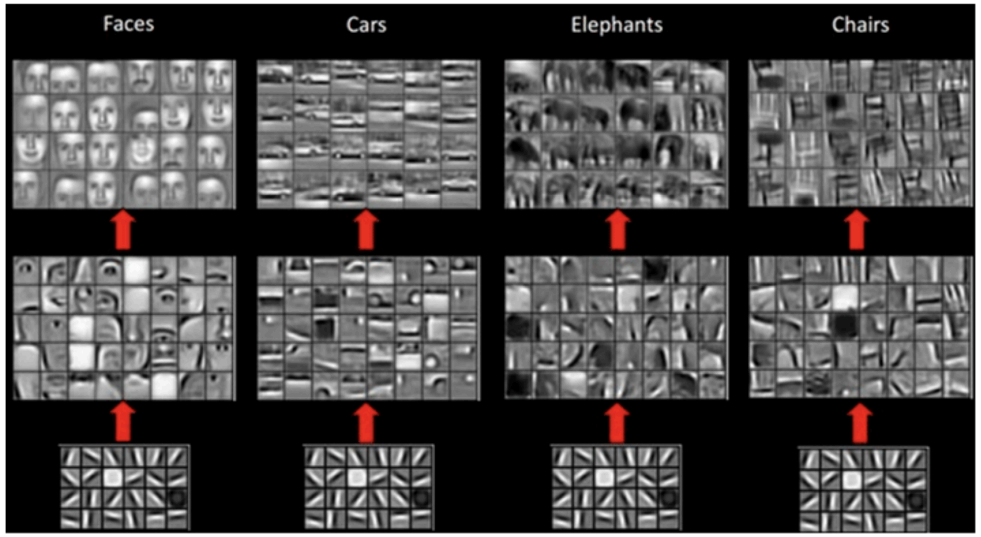
Deep Reinforcement Learning

Value-based Methods

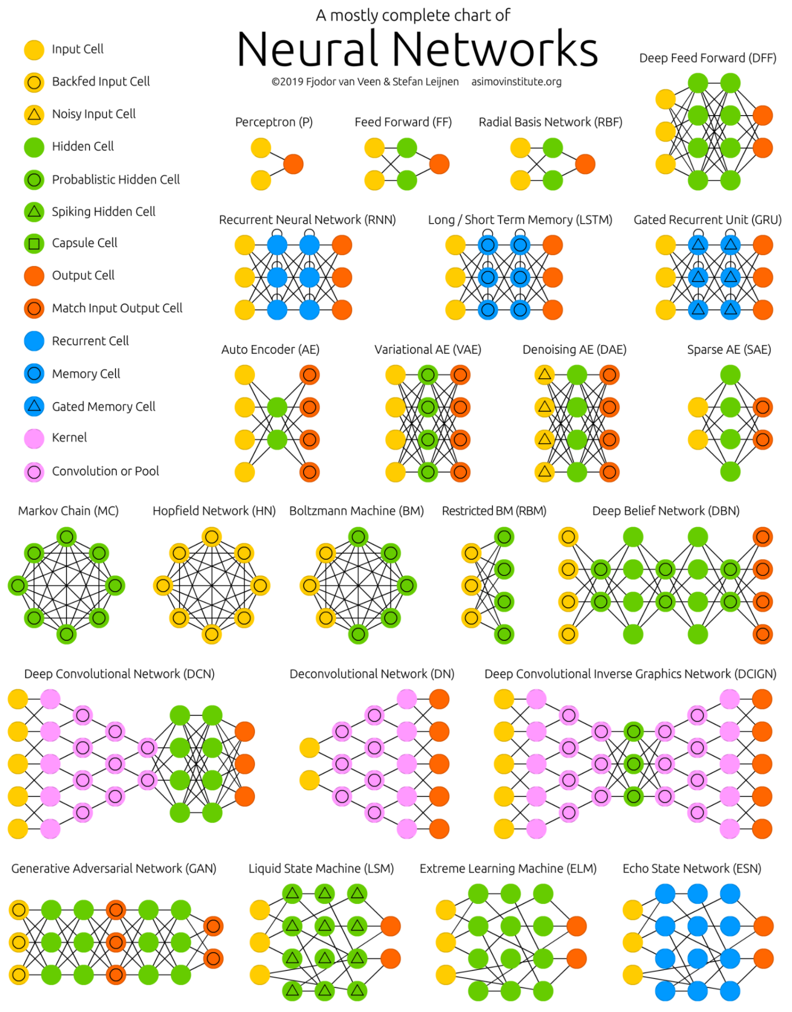
Neural Networks

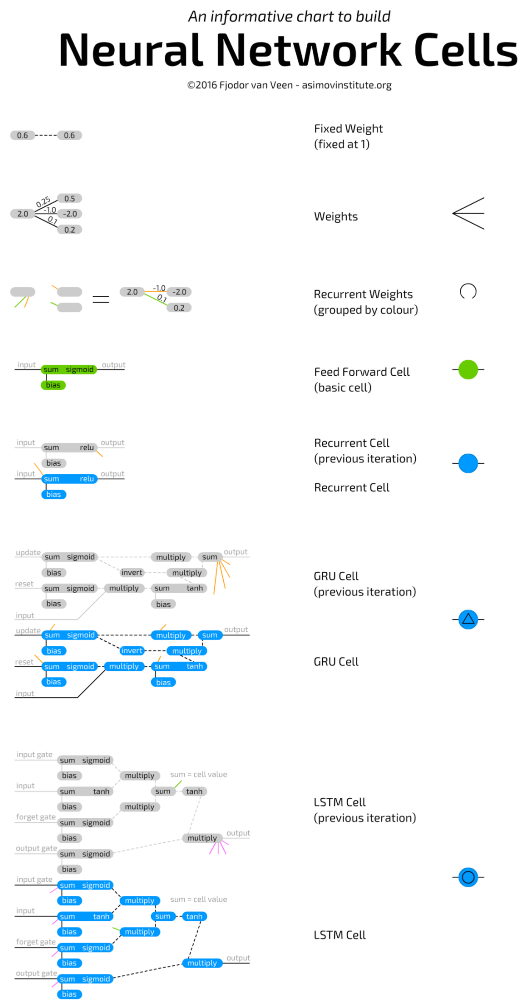
- Neural networks are a biologically-inspired programming paradigm which enables a computer to learn from observational data.
- NN allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts.
- The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones.
- The more the number of layers in that hierarchy, the "deeper" the network, thus, deep learning.
- Deep learning is a powerful set of techniques for learning in deep neural networks.
- Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, natural language processing, and of course, reinforcement learning.





So much to learn!





Additional resources:

- https://www.asimovinstitute.org/neural-network-zoo-prequel-cells-layers/
- https://www.asimovinstitute.org/neural-network-zoo/
- https://www.deeplearningbook.org/
- http://neuralnetworksanddeeplearning.com/

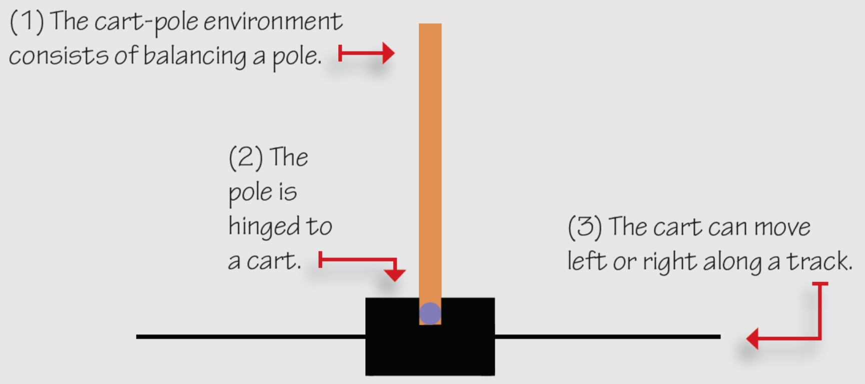
The cart-pole environment

Concrete Example

The Cart-Pole environment

The cart-pole environment is a classic in reinforcement learning. The state space is low-dimensional but continuous, making it an excellent environment for developing algorithms; training is fast, yet still somewhat challenging, and function approximation can help.

This is the cart-pole environment


(1) The cart-pole environment consists of balancing a pole.
(2) The pole is hinged to a cart.
(3) The cart can move left or right along a track.

Its state space is comprised of four variables:

- The cart position on the track (x axis) with a range from -2.4 to 2.4
- The cart velocity along the track (x axis) with a range from -inf to inf
- The pole angle with a range of ~-40 degrees to ~ 40 degrees
- The pole velocity at the tip with a range of -inf to inf

There are two available actions in every state:

- Action 0 applies a -1 force to the cart (push it left)
- Action 1 applies a +1 force to the cart (push it right)

You reach a terminal state if:

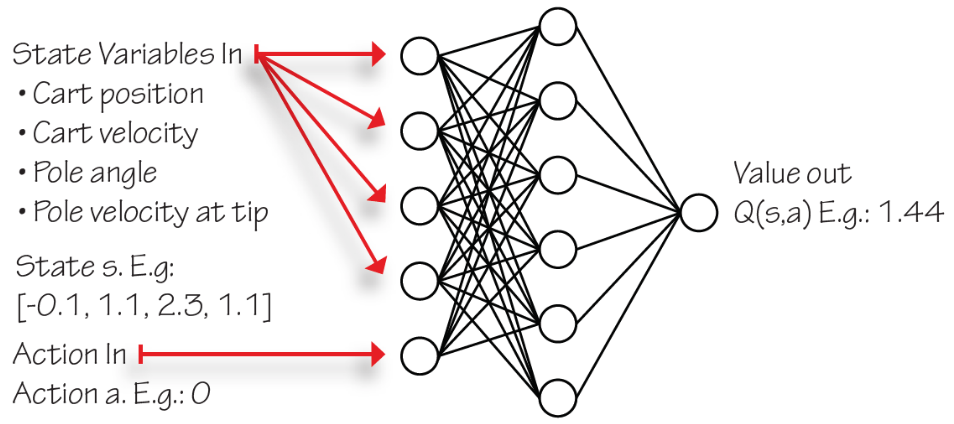
- The pole angle is more than 12 degrees away from the vertical position
- The cart center is more than 2.4 units from the center of the track
- The episode count reaches 500 time steps (more on this later)

The reward function is:

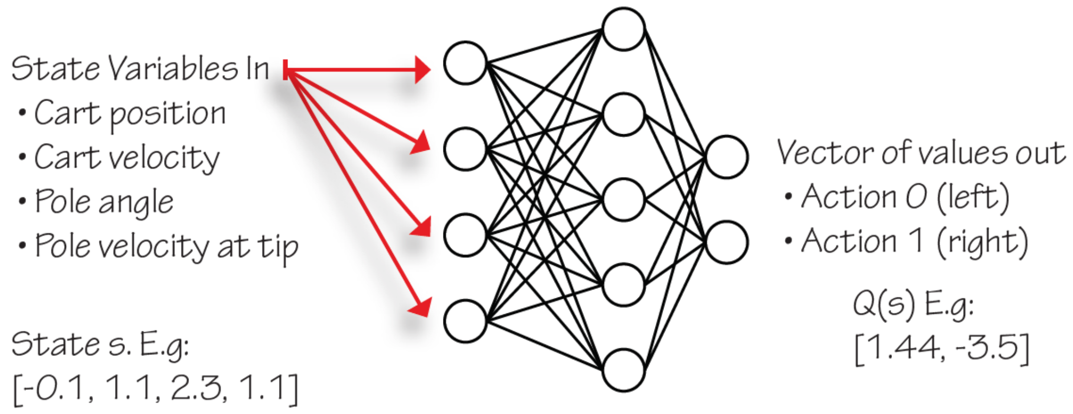
- +1 for every time step

Let’s use neural networks to solve the cart-pole environment

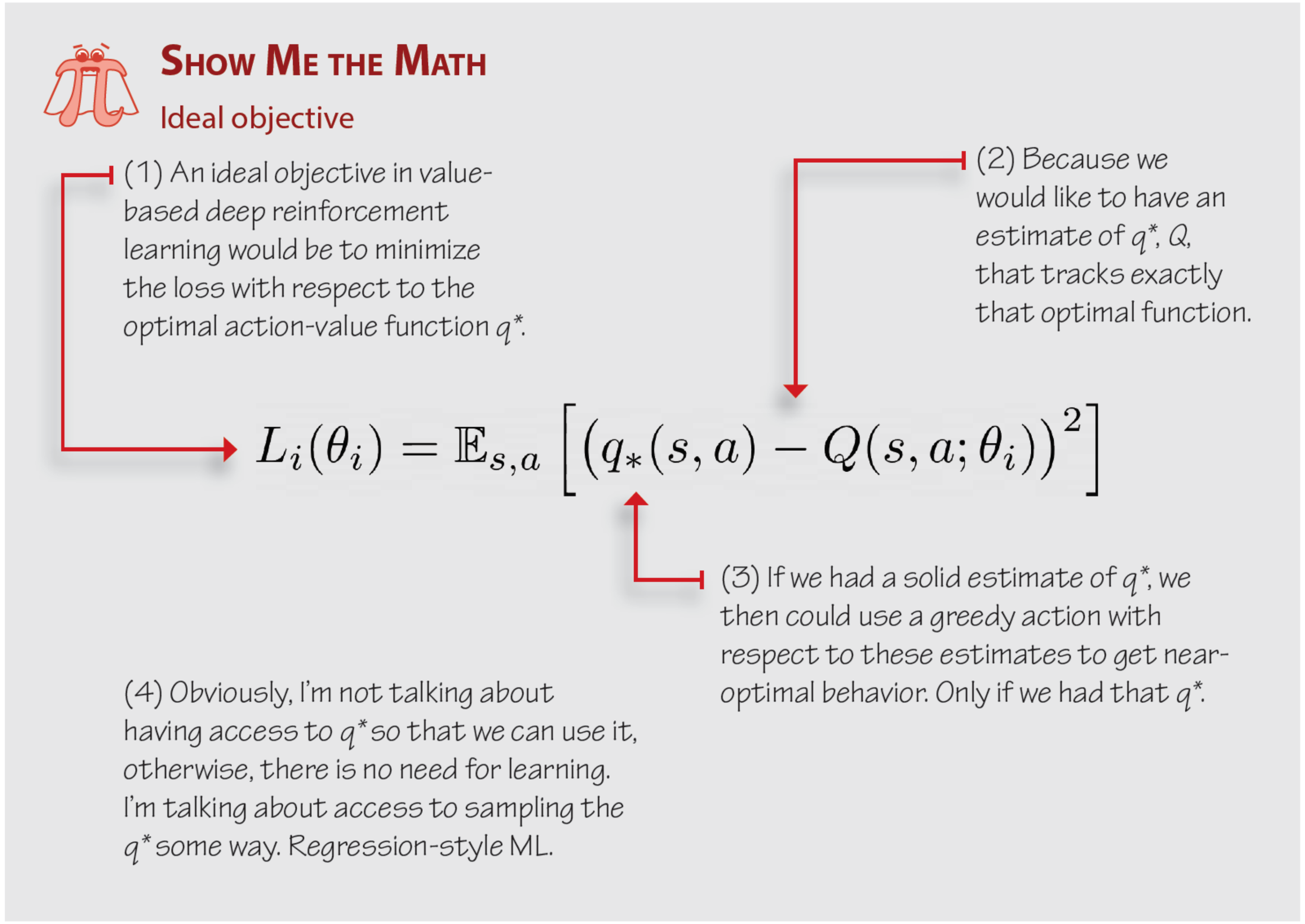
State-action-in-value-out architecture



State-in-values-out architecture



But what's a good objective to train on?



Show Me the Math

Ideal objective

(1) An ideal objective in value-based deep reinforcement learning would be to minimize the loss with respect to the optimal action-value function q^*.

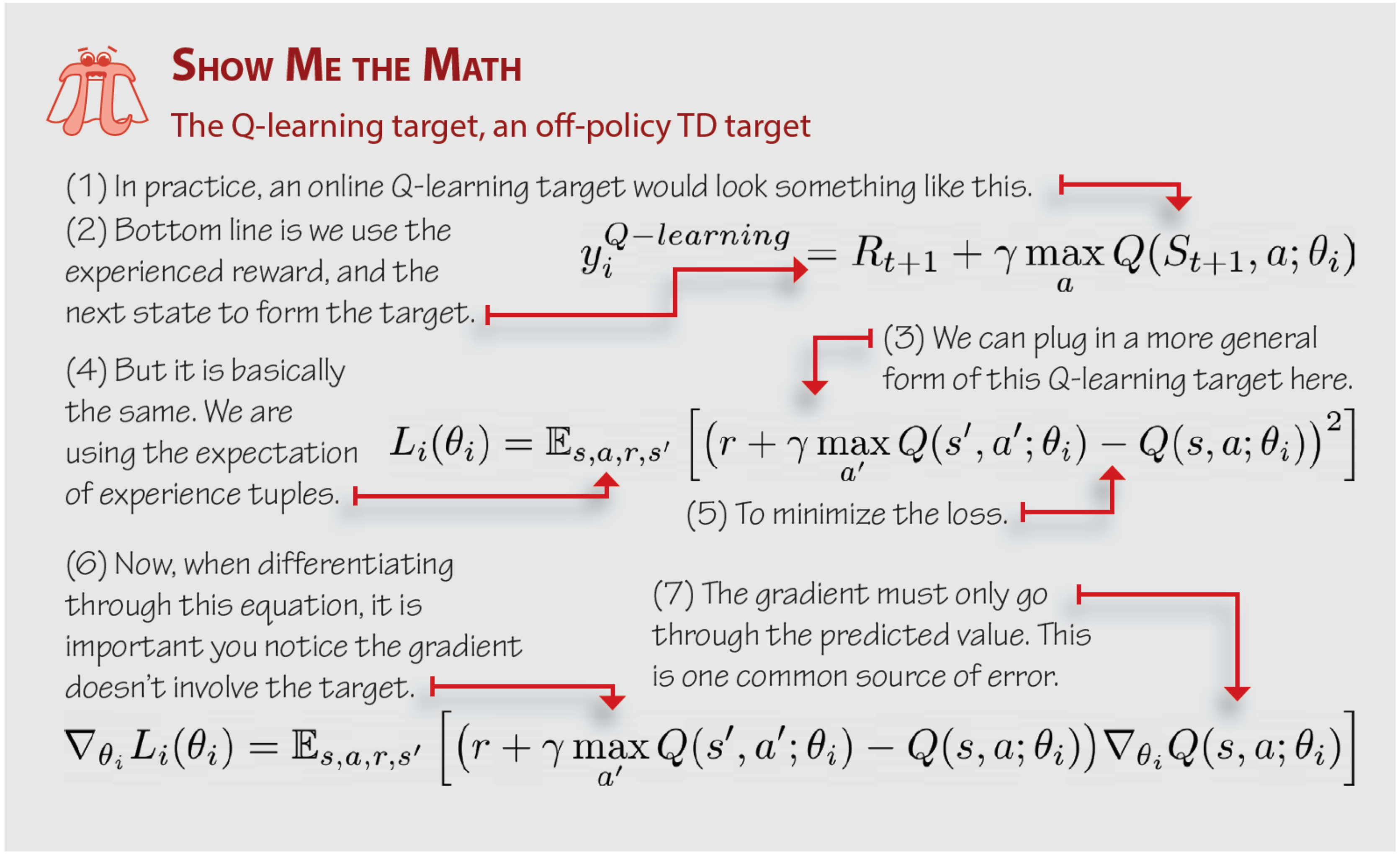
(2) Because we would like to have an estimate of q^*, Q, that tracks exactly that optimal function.

$$L_i(\theta_i) = \mathbb{E}_{s,a}\left[\left(q_*(s,a) - Q(s,a;\theta_i)\right)^2\right]$$

(3) If we had a solid estimate of q^*, we then could use a greedy action with respect to these estimates to get near-optimal behavior. Only if we had that q^*.

(4) Obviously, I'm not talking about having access to q^* so that we can use it, otherwise, there is no need for learning. I'm talking about access to sampling the q^* some way. Regression-style ML.

How about Q-learning?



Show Me the Math

The Q-learning target, an off-policy TD target

(1) In practice, an online Q-learning target would look something like this.

(2) Bottom line is we use the experienced reward, and the next state to form the target.

$$y_i^{Q-learning} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_i)$$

(3) We can plug in a more general form of this Q-learning target here.

(4) But it is basically the same. We are using the expectation of experience tuples.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\right)^2\right]$$

(5) To minimize the loss.

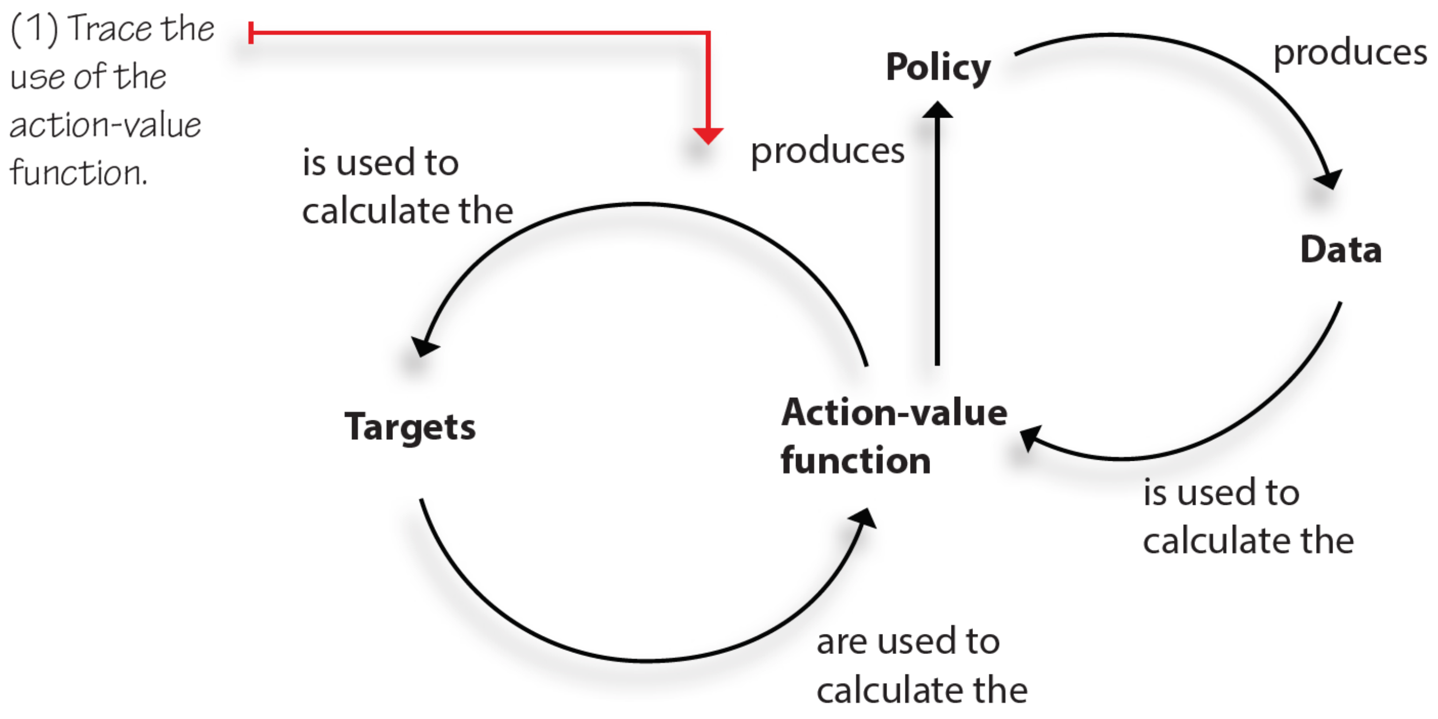
(6) Now, when differentiating through this equation, it is important you notice the gradient doesn't involve the target.

(7) The gradient must only go through the predicted value. This is one common source of error.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

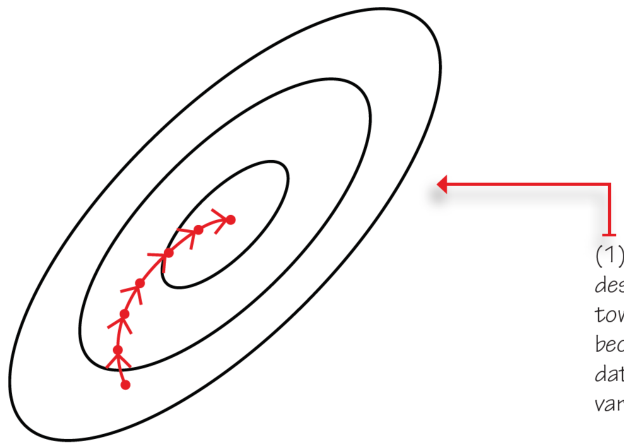
Notice our first challenge

Circular dependency of the action-value function



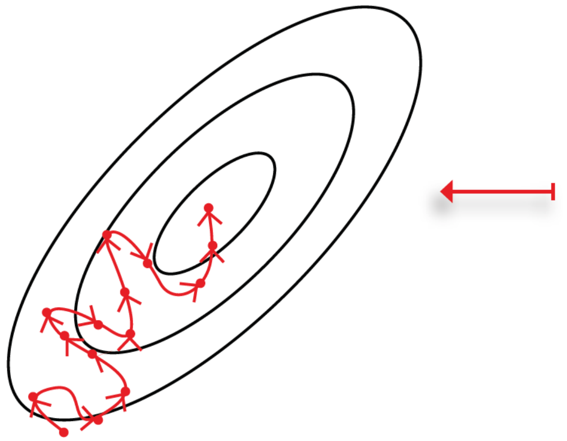
What optimization method to use?

Batch gradient descent



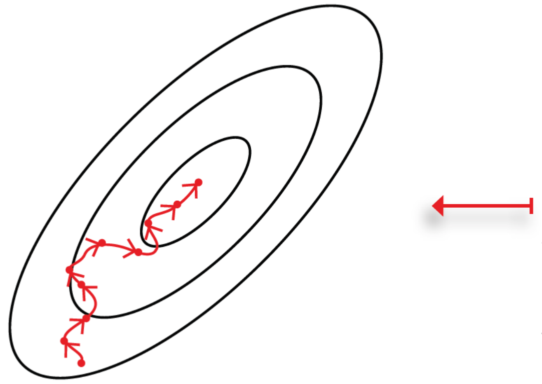
(1) Batch gradient descent goes smoothly towards the target because it uses the entire dataset at once, so lower variance is expected.

Stochastic gradient descent



(1) With stochastic gradient descent every iteration we step only through one sample. This makes it a very noisy algorithm. It wouldn't be surprising to see some steps taking us further away from the target, and later back towards the target.

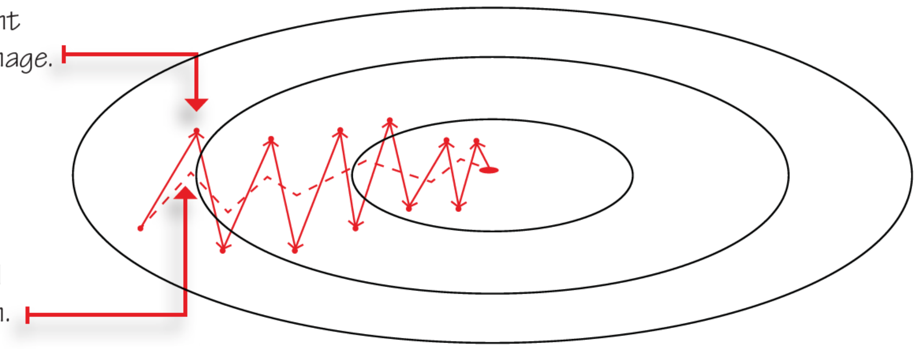
Mini-batch gradient descent



(1) In mini-batch gradient descent we use a uniformly sampled mini batch. This result in noisier updates, but also faster processing of the data

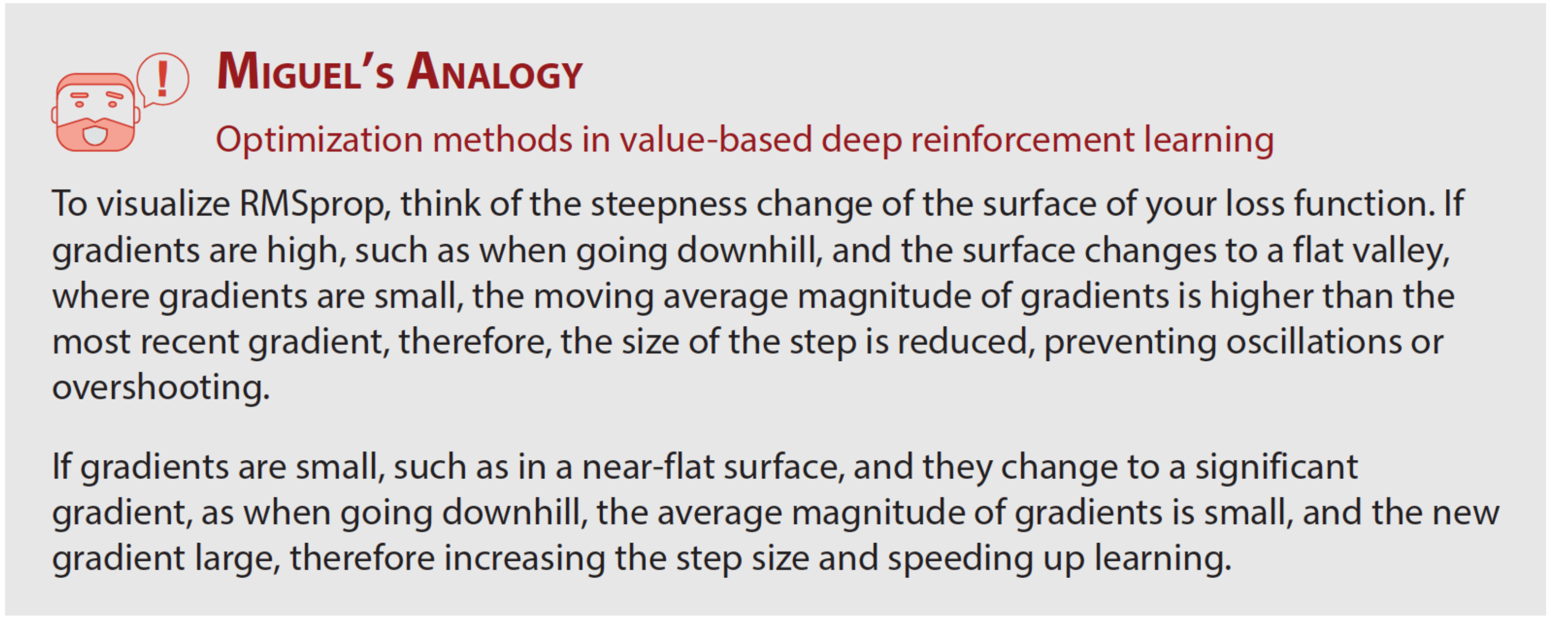
Mini-batch gradient Descent vs Momentum

(1) Mini-batch gradient descent from the last image.



(2) This would be momentum.

Momentum

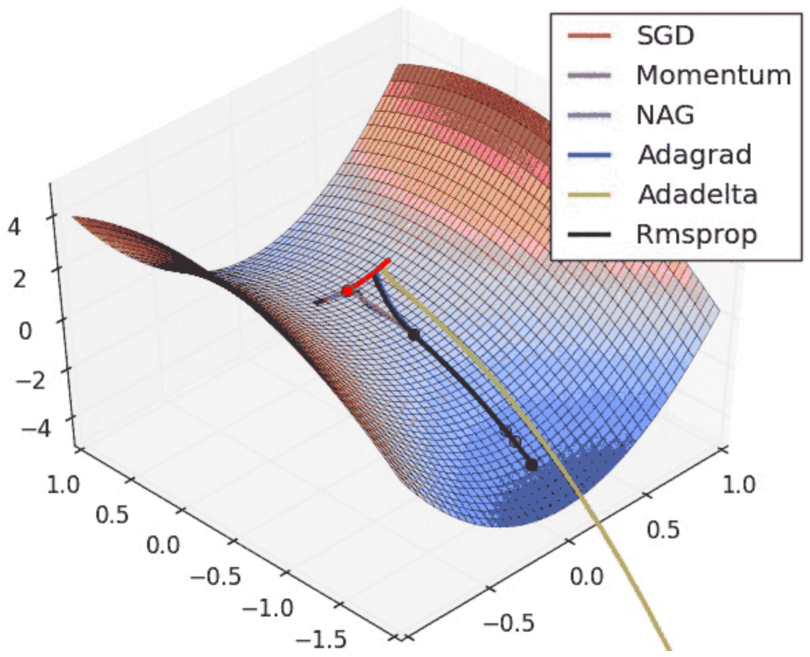


MIGUEL'S ANALOGY

Optimization methods in value-based deep reinforcement learning

To visualize RMSprop, think of the steepness change of the surface of your loss function. If gradients are high, such as when going downhill, and the surface changes to a flat valley, where gradients are small, the moving average magnitude of gradients is higher than the most recent gradient, therefore, the size of the step is reduced, preventing oscillations or overshooting.

If gradients are small, such as in a near-flat surface, and they change to a significant gradient, as when going downhill, the average magnitude of gradients is small, and the new gradient large, therefore increasing the step size and speeding up learning.



Additional readings:

- https://cs231n.github.io/neural-networks-3/

NFQ: Neural Fitted Q-Iteration

It's In The Details

The full Neural Fitted Q-Iteration (NFQ) algorithm

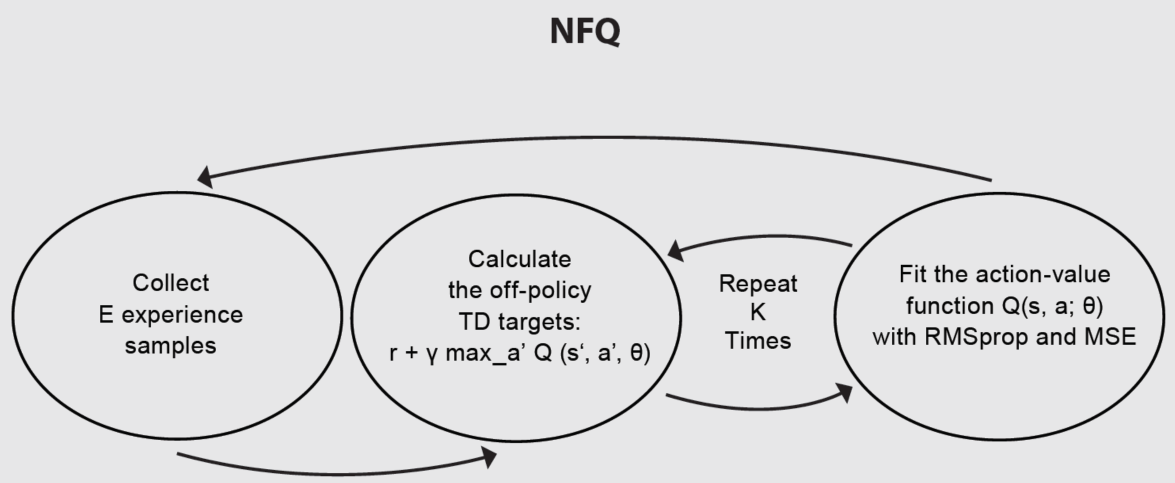
Currently, we have made the following selections, we:

- Approximate the action-value function *Q(s,a; θ)*.
- Use a state-in-values-out architecture (nodes: 4, 512,128, 2).
- Optimize the action-value function to approximate the optimal action-value function *q*(s,a)*.
- Use off-policy TD targets (*r + γ*max_a'Q(s',a'; θ)*) to evaluate policies.
- Use an epsilon-greedy strategy (epsilon set to 0.5) to improve policies.
- Use mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0005.

NFQ has three main steps:

1. Collect E experiences: (s, a, r, s', d) tuples. We use 1024 samples.
2. Calculate the off-policy TD targets: *r + γ*max_a'Q(s',a'; θ)*.
3. Fit the action-value function *Q(s,a; θ)*: Using MSE and RMSprop.

Now, this algorithm repeats steps 2 and 3 *K* number of times before going back to step 1. That's what makes it "fitted"; the nested loop. We'll use 40 fitting steps *K*.


NFQ
Collect E experience samples
Calculate the off-policy TD targets: r + γ max_a' Q (s', a', θ)
Repeat K Times
Fit the action-value function Q(s, a; θ) with RMSprop and MSE

Outline

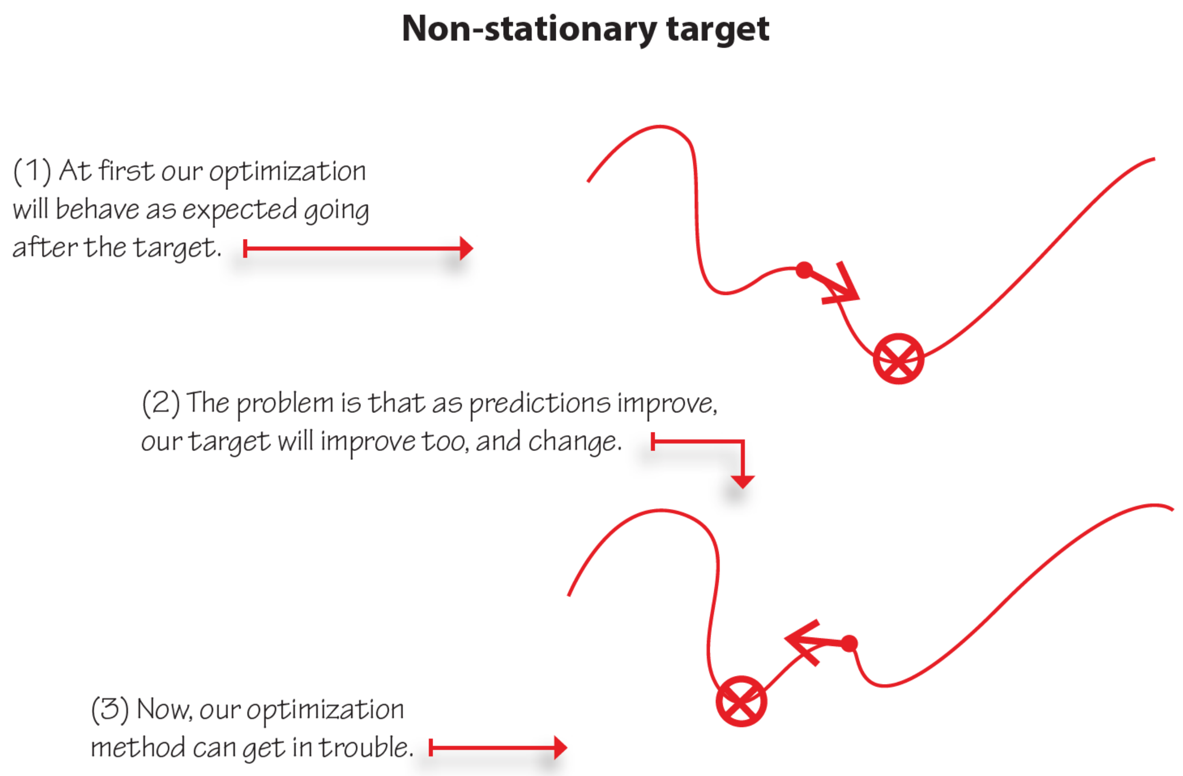
Reinforcement Learning with Function Approximation

Deep Reinforcement Learning

Value-based Methods

Issue #1

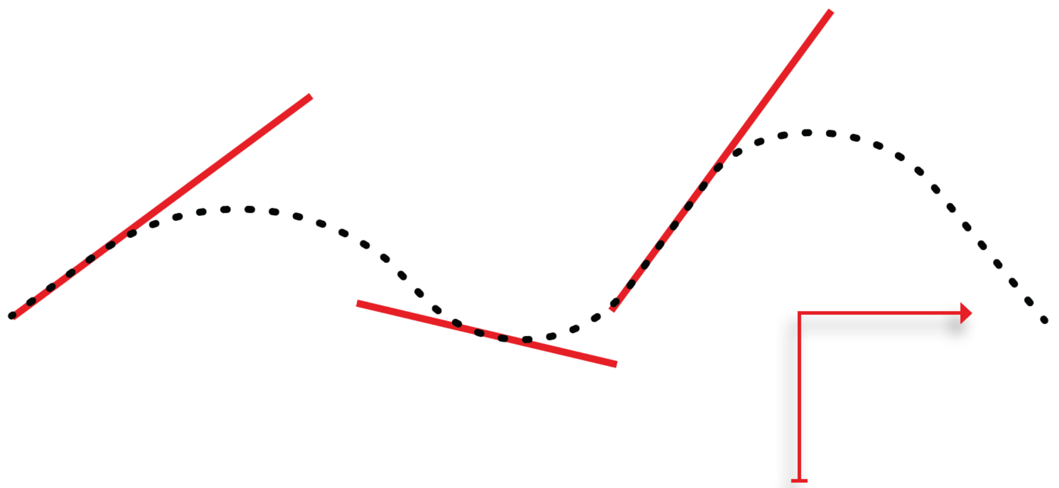
- Because we are using a powerful function approximator, we can generalize across state action pairs, which is excellent, but that also means that the neural network adjusts the values of all similar states at once.
- Think about this for a second, recall that our target values depend on the values for the next state, which we can safely assume are like the states we are adjusting the values of in the first place.
- We are creating a non-stationary target for our learning updates. As we update the weights of the approximate Q-function, the targets also move and make our most recent update outdated.
- Thus, training becomes unstable very quickly.


Non-stationary target
(1) At first our optimization will behave as expected going after the target.
(2) The problem is that as predictions improve, our target will improve too, and change.
(3) Now, our optimization method can get in trouble.

Issue #2

- We are not holding the IID assumption and that is a problem because optimization methods assume the samples to be independent and identically distributed (IID).
- But we are training with almost the exact opposite:
  - Samples on our distribution are not independent because the outcome of a new state “s’” is dependent on our current state “s.”
  - Samples are not identically distributed because the underlying data generating process, which is our policy, is changing over time.

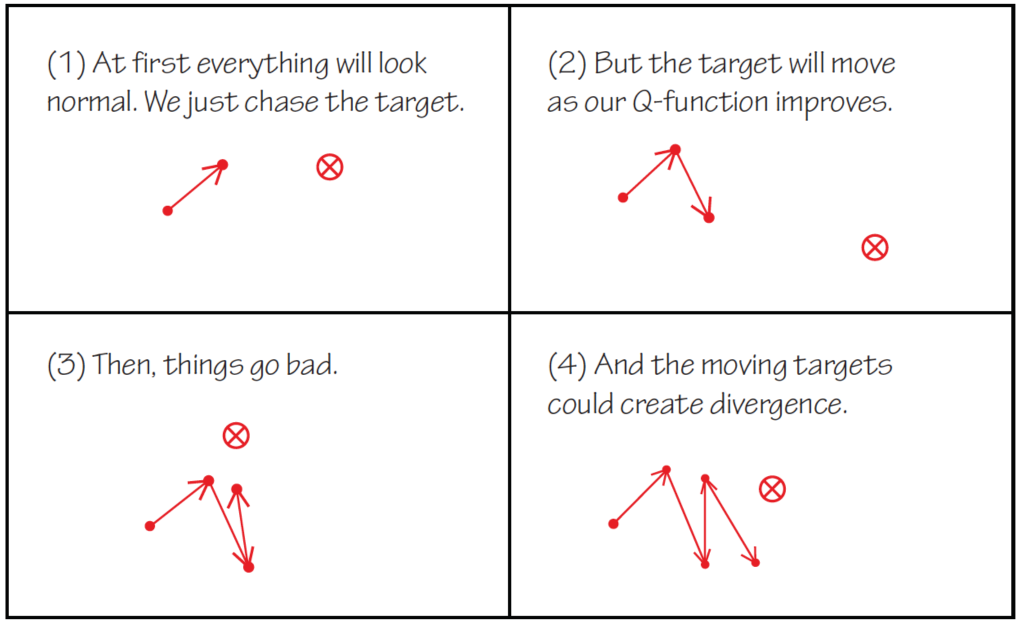
Data correlated with time



(1) Imagine we generate these data points in a single trajectory. Say the y axis is the position of the cart along the track, and the x axis is the step of the trajectory. You can see how likely it is data points at adjacent time steps will be similar making our function approximator likely to overfit to that local region.

Target networks

Q-function optimization without a target network



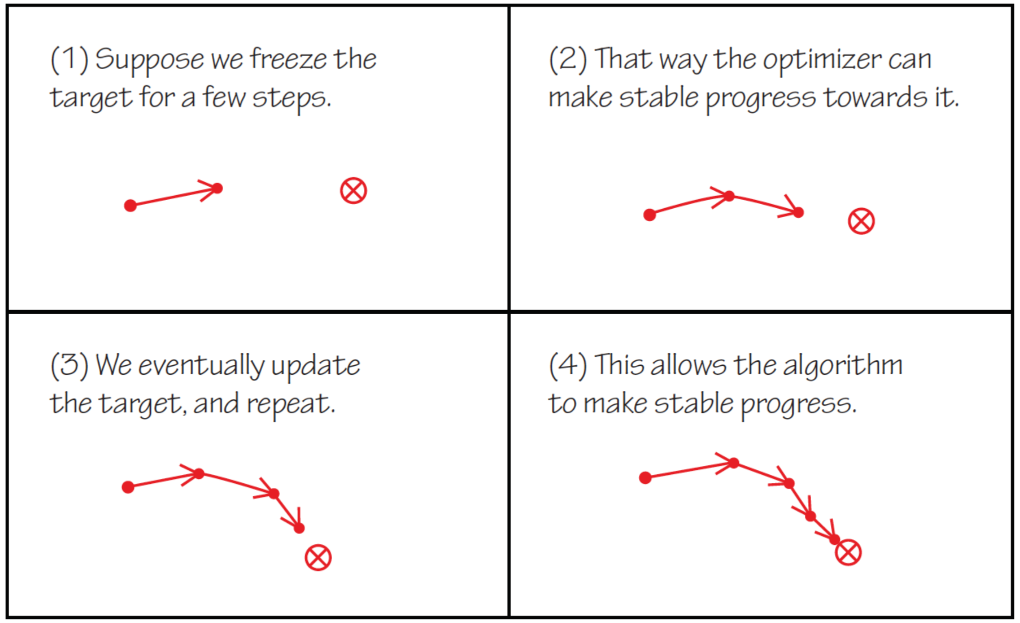
(1) At first everything will look normal. We just chase the target.

(2) But the target will move as our Q-function improves.

(3) Then, things go bad.

(4) And the moving targets could create divergence.

Q-function approximation with a target network



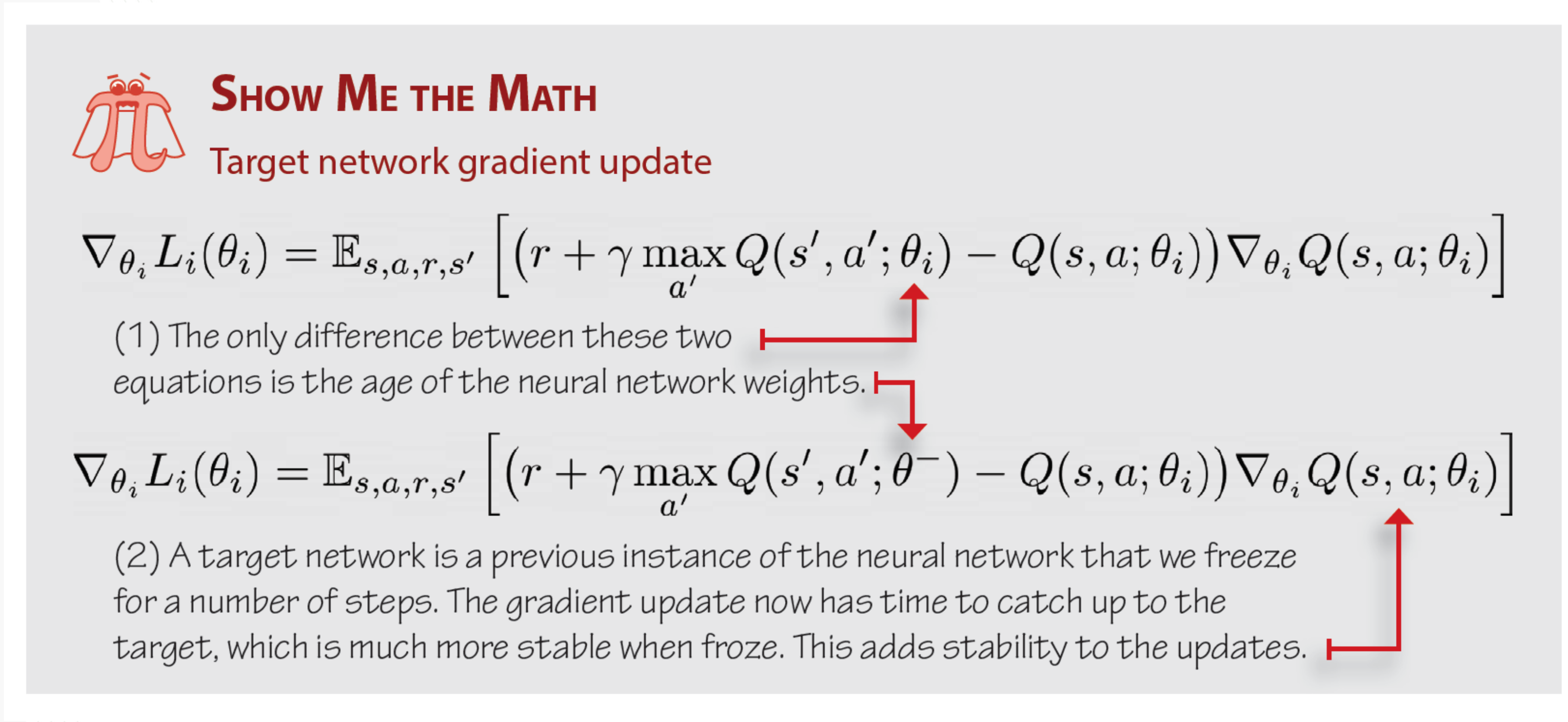
(1) Suppose we freeze the target for a few steps.

(2) That way the optimizer can make stable progress towards it.

(3) We eventually update the target, and repeat.

(4) This allows the algorithm to make stable progress.

Target networks equations



Show Me the Math

Target network gradient update

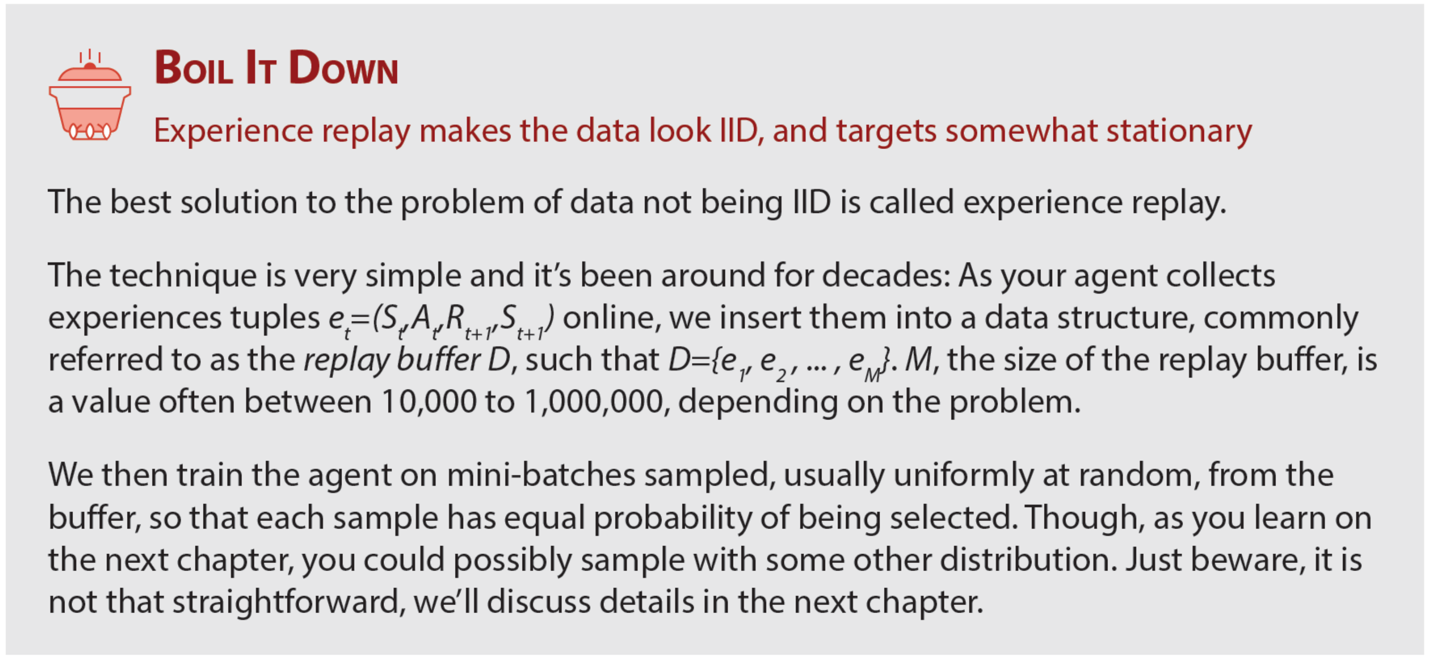
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

(1) The only difference between these two equations is the age of the neural network weights.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

(2) A target network is a previous instance of the neural network that we freeze for a number of steps. The gradient update now has time to catch up to the target, which is much more stable when froze. This adds stability to the updates.

Replay buffers



Boil It Down

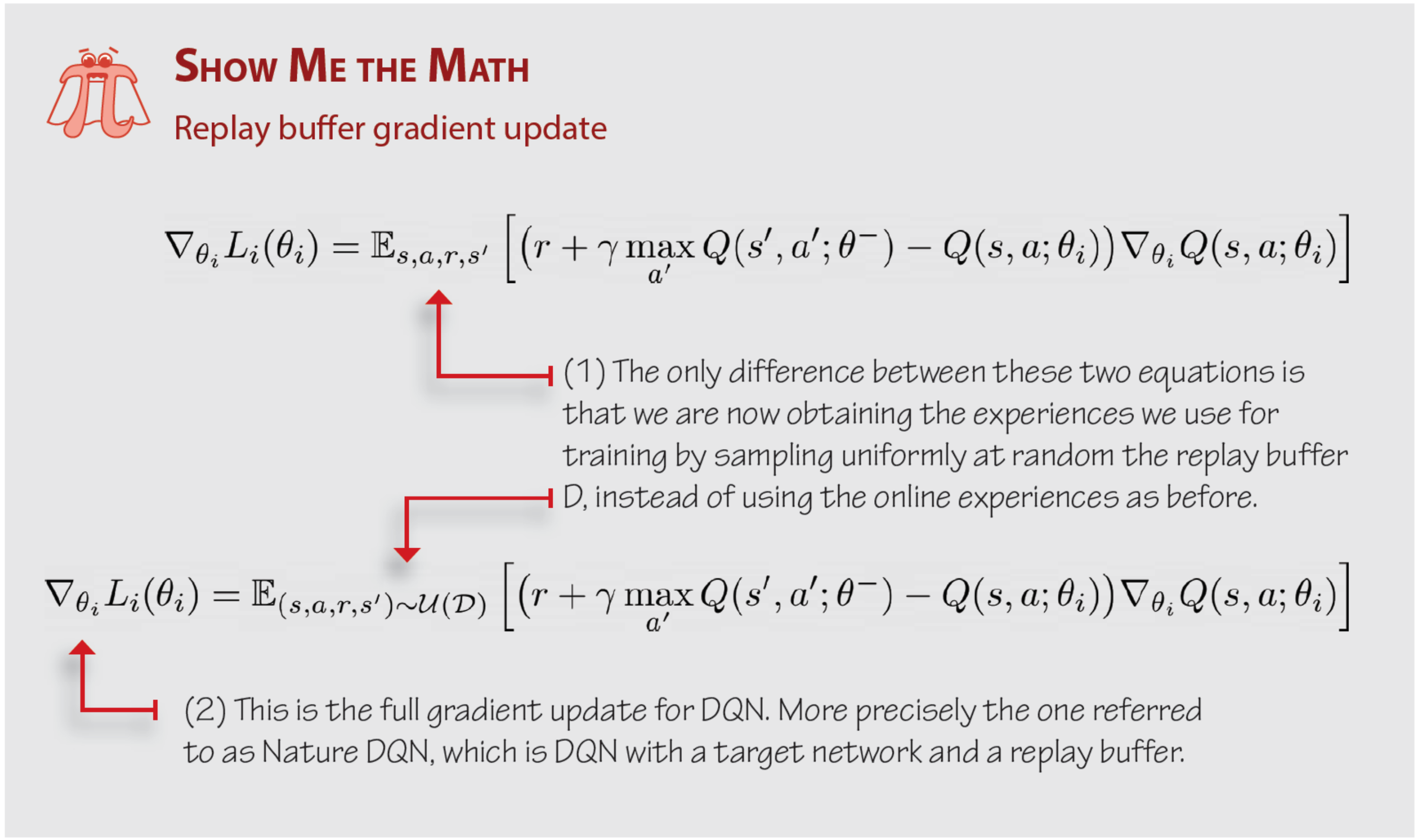
Experience replay makes the data look IID, and targets somewhat stationary

The best solution to the problem of data not being IID is called experience replay.

The technique is very simple and it's been around for decades: As your agent collects experiences tuples $e_t=(S_t,A_t,R_{t+1},S_{t+1})$ online, we insert them into a data structure, commonly referred to as the *replay buffer D*, such that $D=\{e_1, e_2, \ldots, e_M\}$. *M*, the size of the replay buffer, is a value often between 10,000 to 1,000,000, depending on the problem.

We then train the agent on mini-batches sampled, usually uniformly at random, from the buffer, so that each sample has equal probability of being selected. Though, as you learn on the next chapter, you could possibly sample with some other distribution. Just beware, it is not that straightforward, we'll discuss details in the next chapter.

Replay buffers equations



Show Me the Math

Replay buffer gradient update

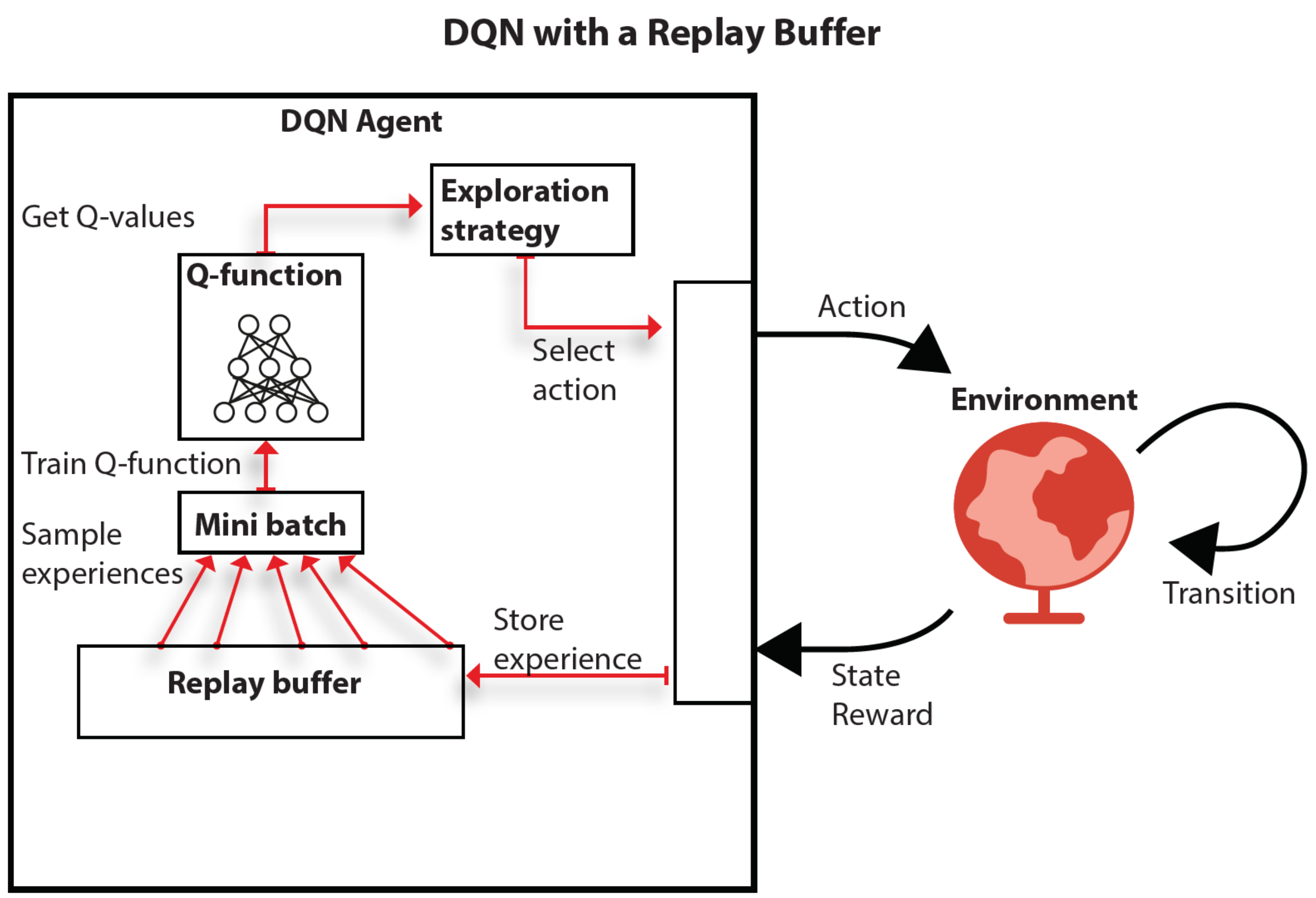
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

(1) The only difference between these two equations is that we are now obtaining the experiences we use for training by sampling uniformly at random the replay buffer D, instead of using the online experiences as before.

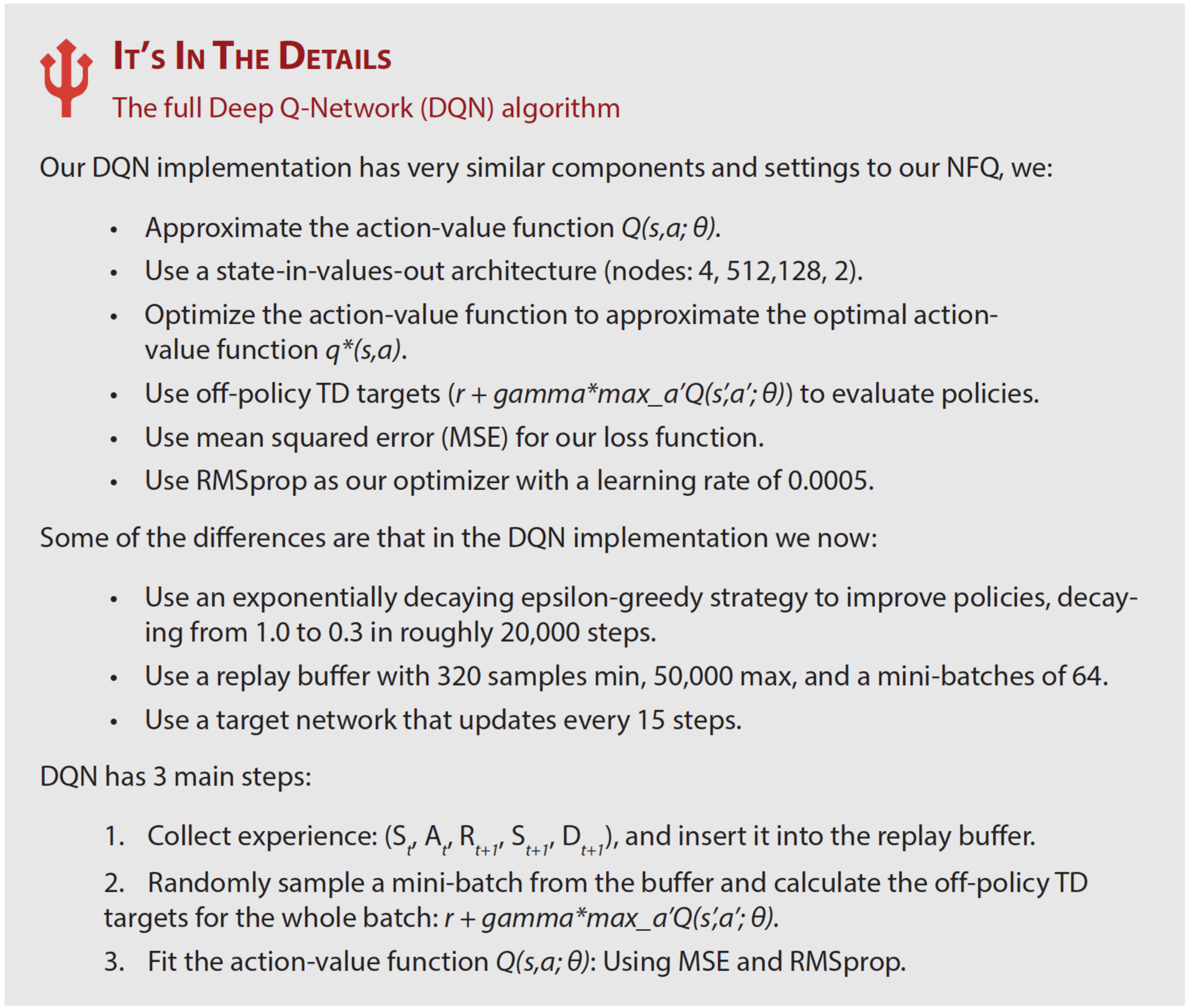
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

(2) This is the full gradient update for DQN. More precisely the one referred to as Nature DQN, which is DQN with a target network and a replay buffer.

DQN: Deep Q-Networks with a Replay Buffer



Training DQN in the cart-pole environment



It's In The Details

The full Deep Q-Network (DQN) algorithm

Our DQN implementation has very similar components and settings to our NFQ, we:

- Approximate the action-value function *Q(s,a; θ)*.
- Use a state-in-values-out architecture (nodes: 4, 512,128, 2).
- Optimize the action-value function to approximate the optimal action-value function *q*(s,a)*.
- Use off-policy TD targets (*r + gamma*max_a'Q(s',a'; θ)*) to evaluate policies.
- Use mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0005.

Some of the differences are that in the DQN implementation we now:

- Use an exponentially decaying epsilon-greedy strategy to improve policies, decaying from 1.0 to 0.3 in roughly 20,000 steps.
- Use a replay buffer with 320 samples min, 50,000 max, and a mini-batches of 64.
- Use a target network that updates every 15 steps.

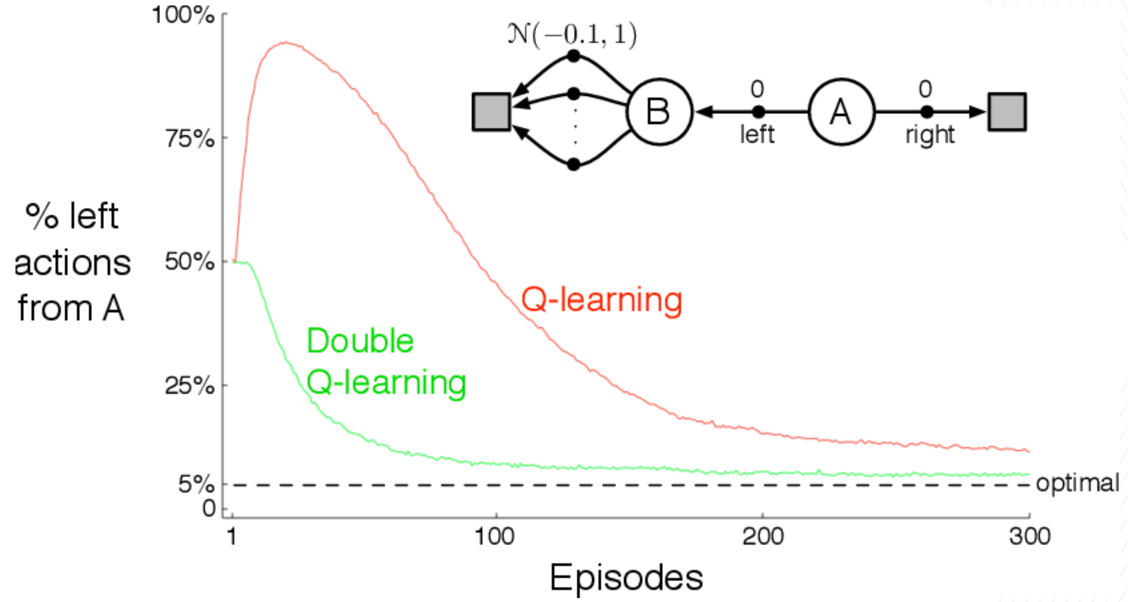
DQN has 3 main steps:

1. Collect experience: (S_t, A_t, R_{t+1}, S_{t+1}, D_{t+1}), and insert it into the replay buffer.
2. Randomly sample a mini-batch from the buffer and calculate the off-policy TD targets for the whole batch: *r + gamma*max_a'Q(s',a'; θ)*.
3. Fit the action-value function *Q(s,a; θ)*: Using MSE and RMSprop.

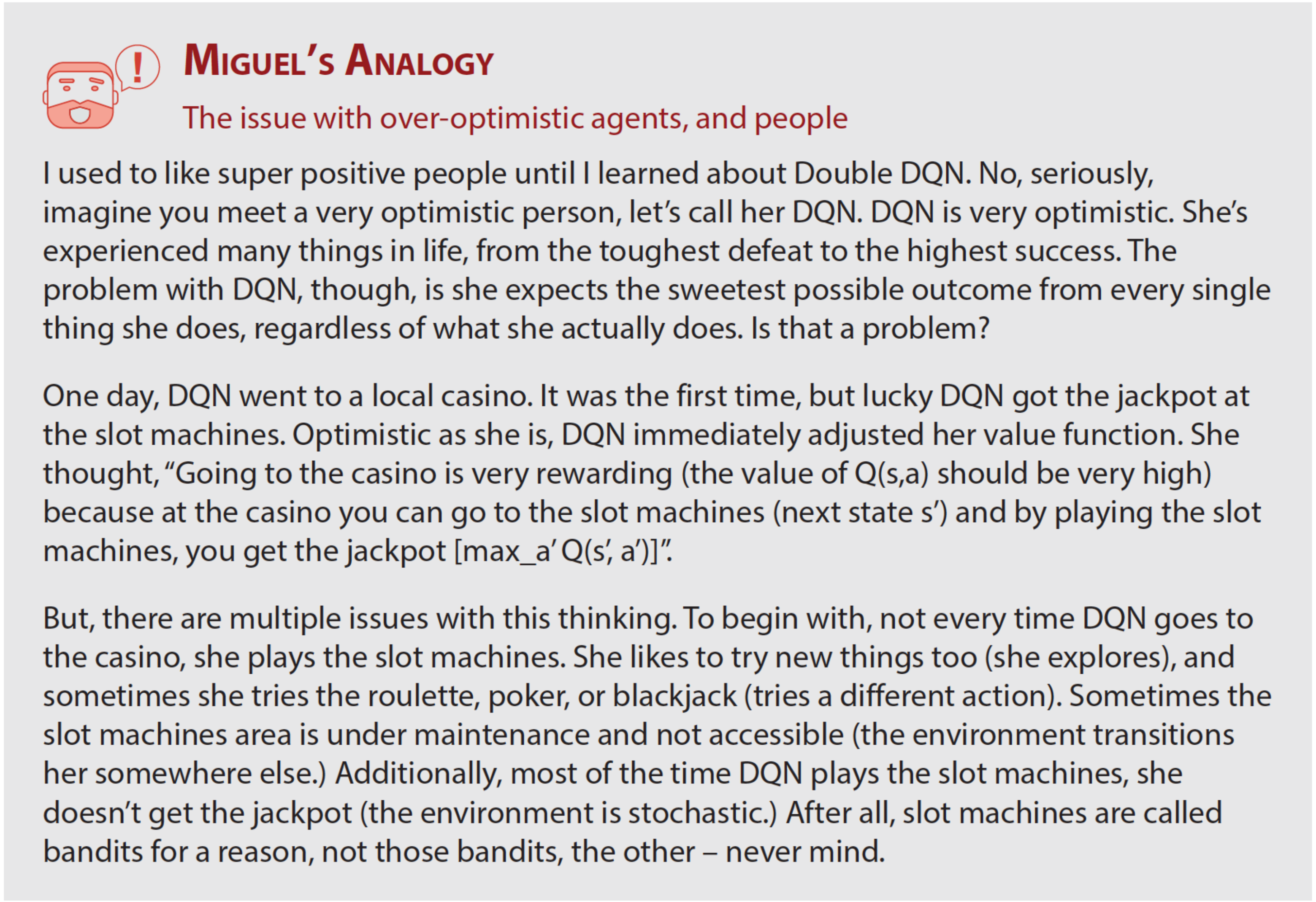
Georgia Tech
CREATING THE NEXT

Issue #1

- DQN overestimates the targets. This algorithm is biased towards positive values because we use the max all the time.
- The crux of the problem is very simple: We are taking the max of estimated values. Estimated values are often off-center, some higher than the true values, some lower, but the bottom line is they are off. Now, the problem is that we are always taking the max of these values.
- DQN prefers higher values, even if they are not correct. Meaning it shows a positive bias, and performance suffers.



Illustrating the problem of overestimation



Miguel’s Analogy

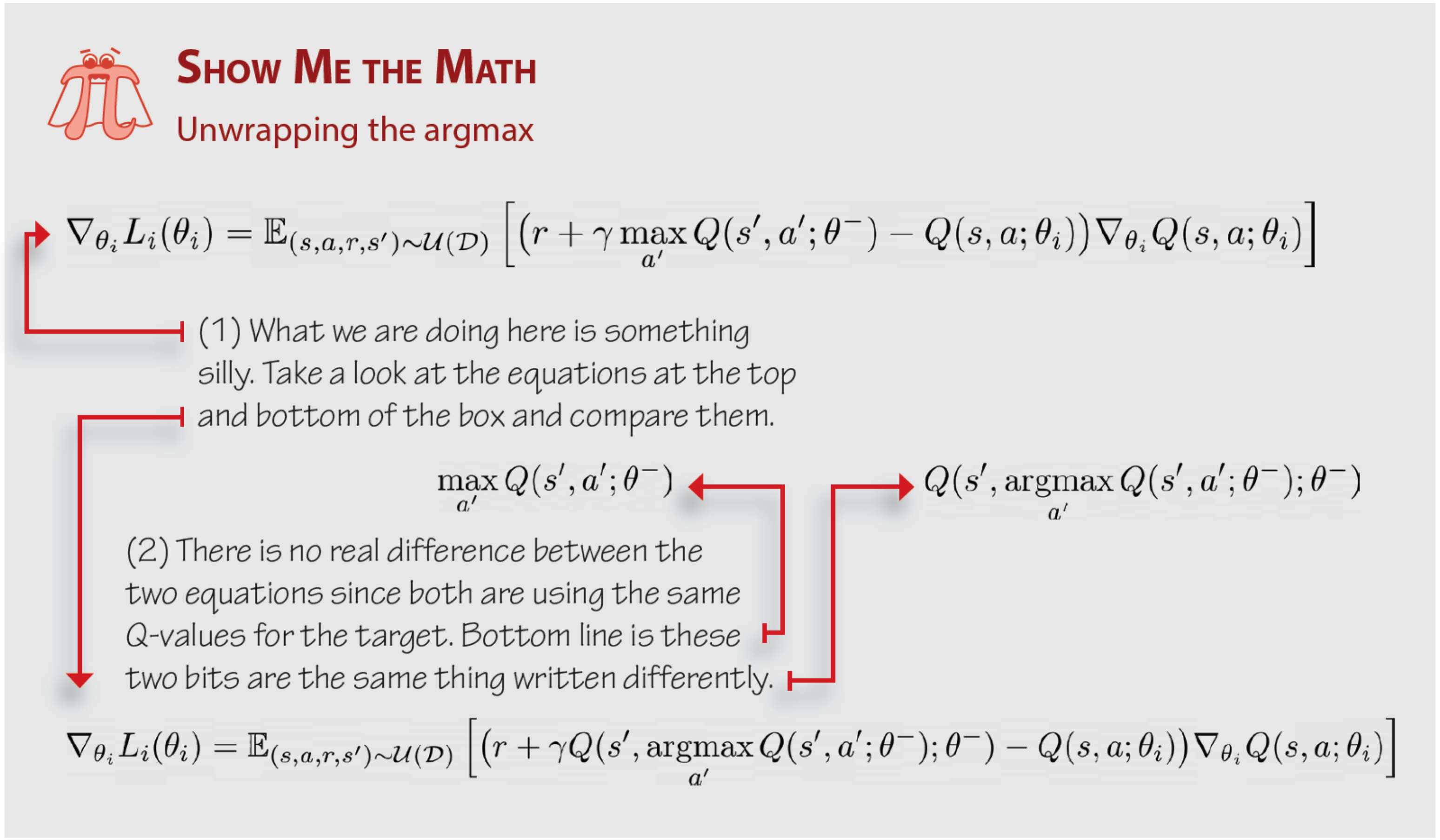
The issue with over-optimistic agents, and people

I used to like super positive people until I learned about Double DQN. No, seriously, imagine you meet a very optimistic person, let’s call her DQN. DQN is very optimistic. She’s experienced many things in life, from the toughest defeat to the highest success. The problem with DQN, though, is she expects the sweetest possible outcome from every single thing she does, regardless of what she actually does. Is that a problem?

One day, DQN went to a local casino. It was the first time, but lucky DQN got the jackpot at the slot machines. Optimistic as she is, DQN immediately adjusted her value function. She thought, “Going to the casino is very rewarding (the value of Q(s,a) should be very high) because at the casino you can go to the slot machines (next state s’) and by playing the slot machines, you get the jackpot [max_a’ Q(s’, a’)]”.

But, there are multiple issues with this thinking. To begin with, not every time DQN goes to the casino, she plays the slot machines. She likes to try new things too (she explores), and sometimes she tries the roulette, poker, or blackjack (tries a different action). Sometimes the slot machines area is under maintenance and not accessible (the environment transitions her somewhere else.) Additionally, most of the time DQN plays the slot machines, she doesn’t get the jackpot (the environment is stochastic.) After all, slot machines are called bandits for a reason, not those bandits, the other – never mind.

DDQN: What about that max?



Show Me the Math

Unwrapping the argmax

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(\mathcal{D})} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

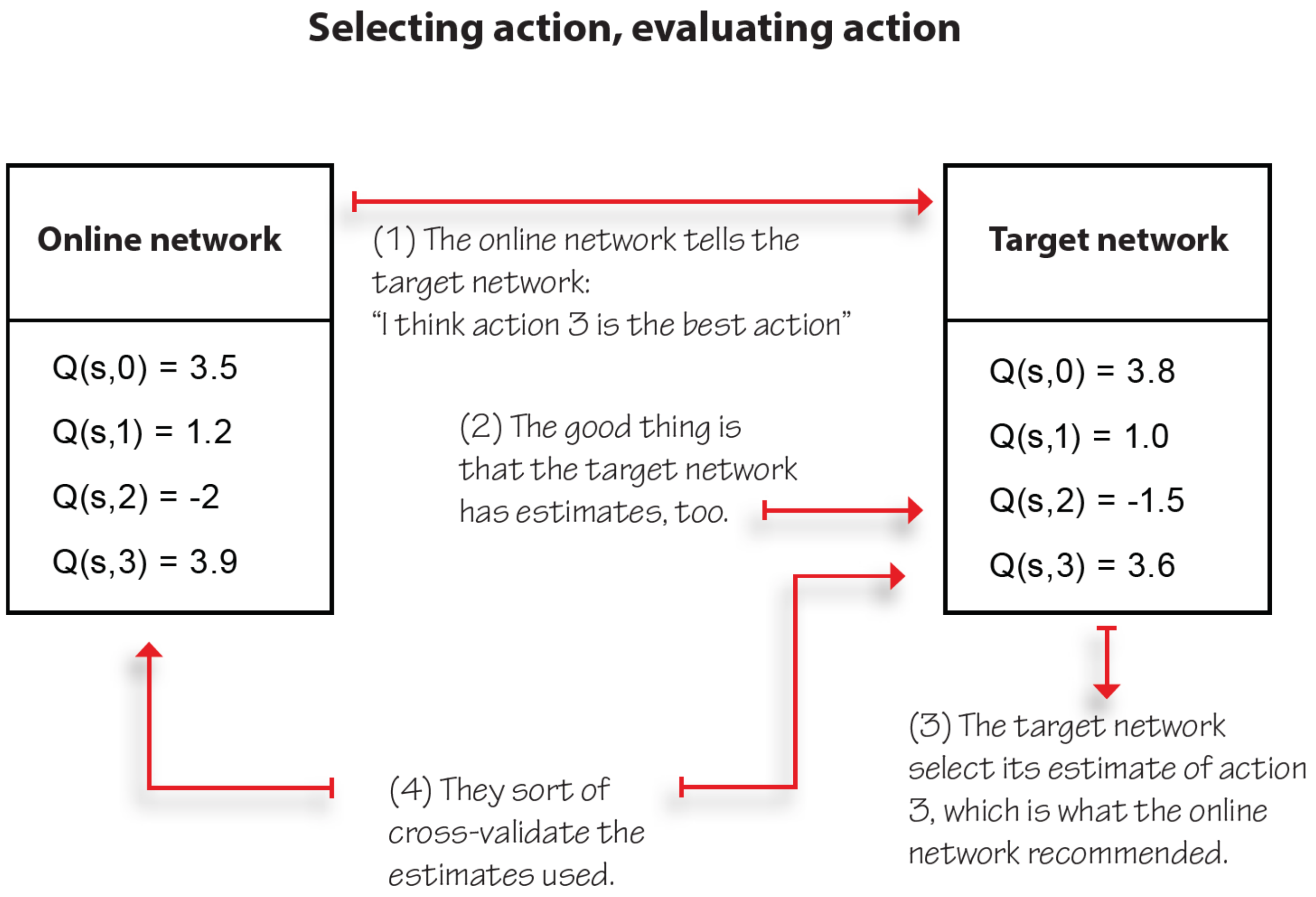
(1) What we are doing here is something silly. Take a look at the equations at the top and bottom of the box and compare them.

$$\max_{a'} Q(s', a'; \theta^-) \qquad Q(s', \operatorname{argmax}_{a'} Q(s', a'; \theta^-); \theta^-)$$

(2) There is no real difference between the two equations since both are using the same Q-values for the target. Bottom line is these two bits are the same thing written differently.

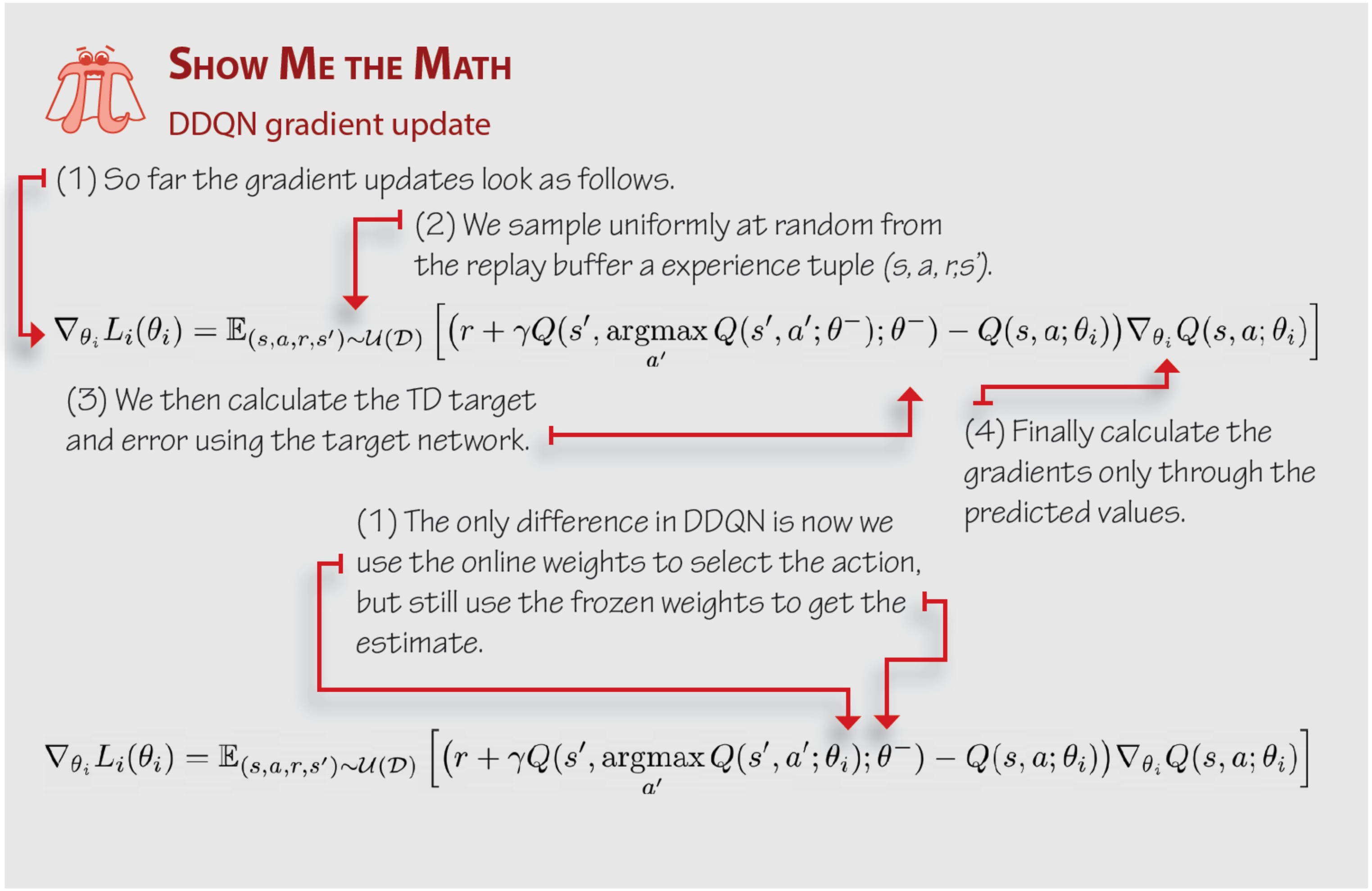
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(\mathcal{D})} \left[\left(r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s', a'; \theta^-); \theta^-) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

DDQN: Double Learning



Selecting action, evaluating action
Online network
Q(s,0) = 3.5
Q(s,1) = 1.2
Q(s,2) = -2
Q(s,3) = 3.9
(1) The online network tells the target network: "I think action 3 is the best action"
Target network
Q(s,0) = 3.8
Q(s,1) = 1.0
Q(s,2) = -1.5
Q(s,3) = 3.6
(2) The good thing is that the target network has estimates, too.
(3) The target network select its estimate of action 3, which is what the online network recommended.
(4) They sort of cross-validate the estimates used.

DDQN equations



Show Me the Math

DDQN gradient update

(1) So far the gradient updates look as follows.

(2) We sample uniformly at random from the replay buffer a experience tuple (s, a, r,s').

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s',a';\theta^-);\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

(3) We then calculate the TD target and error using the target network.

(4) Finally calculate the gradients only through the predicted values.

(1) The only difference in DDQN is now we use the online weights to select the action, but still use the frozen weights to get the estimate.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s',a';\theta_i);\theta^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Training DDQN in the cart-pole environment

It's In The Details

The full Double Deep Q-Network (DDQN) algorithm

DDQN is almost identical to DQN, but there are still some differences. We still:

- Approximate the action-value function *Q(s,a; θ)*.
- Use a state-in-values-out architecture (nodes: 4, 512,128, 2).
- Optimize the action-value function to approximate the optimal action-value function *q*(s,a)*.
- Use off-policy TD targets (*r + gamma*max_a'Q(s',a'; θ)*) to evaluate policies.

Notice that we now:

- Use an adjustable Huber loss, which since we set the 'max_gradient_norm' variable to 'float('inf')', we are effectively just using mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0007. Note that before we used 0.0005 because without double learning (vanilla DQN) some seeds fail if we train with a learning rate of 0.0007. Perhaps stability? In DDQN, on the other hand, training with a higher learning rate works best.

In DDQN we are still using:

- An exponentially decaying epsilon-greedy strategy (from 1.0 to 0.3 in roughly 20,000 steps) to improve policies.
- A replay buffer with 320 samples min, 50,000 max, and a batch of 64.
- A target network that freezes for 15 steps and then updates fully.

DDQN, just like DQN has the same 3 main steps:

1. Collect experience: (S_t, A_t, R_{t+1}, S_{t+1}, D_{t+1}), and insert it into the replay buffer.
2. Randomly sample a mini-batch from the buffer and calculate the off-policy TD targets for the whole batch: *r + gamma*max_a'Q(s',a'; θ)*.
3. Fit the action-value function Q(s,a; θ): Using MSE and RMSprop.

The bottom line is the DDQN implementation and hyperparameters are *identical* to those of DQN, *except* that we now use double learning and therefore train with a slightly higher learning rate. The addition of the Huber loss does not change anything because we are "clipping" gradients to a max value of infinite, which is equivalent to using MSE. However, for many other environments you will find it useful, so tune this hyperparameter.

Georgia Tech
CREATING THE NEXT

Recommended Readings

- DQN v1: https://arxiv.org/abs/1312.5602
- DQN v2: https://www.nature.com/articles/nature14236
- DDQN: https://arxiv.org/abs/1509.06461
- Dueling Networks: https://arxiv.org/abs/1511.06581
- Prioritized Experience Replay: https://arxiv.org/abs/1511.05952
- Rainbow: https://arxiv.org/abs/1710.02298

Thank you!